b
UNIVERSITÄT
BERN

Issue Report Assessment

Assessment of Issue Report Quality and Class through Natural Language Processing

Bachelor Thesis

Simon Curty
from
Bern, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

31. Januar 2018

Prof. Dr. Oscar Nierstrasz
Dr. Haidar Osman
Software Composition Group

Institut für Informatik
University of Bern, Switzerland

Abstract

Issue reports are a crucial aspect of software development. They enable collaboration and communication of faults and task. However, issue reports are only helpful if they provide meaningful information, which is often not the case. Such low-quality issue reports induce a multitude of problems: Developers lose time figuring out the issue. Both time needed to complete the issue and its priority are harder to estimate. Assigning issues to the right developer is more difficult. To alleviate these negative impacts, commonly a triager is taking the responsibility to assess issue reports. However, this process is time consuming and cannot fill in missing information. Since issue reports provide insight into a projects changes and faults over time, they are an often used data source for bug prediction, that is the inference of knowledge about past and future bugs. But the quality of such predictions depends on the quality of the source data.

We address two factors strongly affecting the quality of issue reports: (1) issue reports have an assigned type. When the assigned type does not reflect the true nature of the report, it is said to be misclassified. An issue report classified as bug may in reality be a feature request. This *Misclassification* introduces bias in bug prediction and makes it harder to assign an issue to the right developer. (2) Bug reports are commonly written by hand and the reporter does not always include important information. The *quality of the content* of a bug report impacts the time a developer needs to spend identifying the actual issue.

To tackle the misclassification problem, we propose an approach to categorize issue reports. Our classifier achieves an accuracy of 82.9% when classifying issue reports into *bugs* and *non-bugs*. For multiclass classification of issue reports by type, our model achieves an accuracy of 74.4%. Thus, our model can reliably validate datasets used for bug prediction and distinguish between issue types. To address the second problem, we propose an approach to estimate the quality of bug reports and assign them a score. This *Quality Estimator* is capable of giving improvement suggestions, potentially helping reporters to write more helpful bug reports. To showcase real world

applications of our models we integrated both the classifier model and the *Quality Estimator* into a tool for issue report assessment. The tool is implemented as browser extension and allows the user to get feedback about the type of an issue report with one click. The tool is also capable of providing suggestions on how to improve a bug report based on the information already provided.

Contents

1 Introduction **1**

2 Related Work **3**
2.1 Issue Report classification 3
2.2 Bug Report Quality 4

3 Issue Report Assessment with Machine Learning **6**
3.1 Dataset 6
3.2 Issue Report Anatomy 9
3.3 Issue Reports Classification 10
3.3.1 Deep Learning 10
3.3.2 Issue Vectorization 10
3.3.3 Feedforward Neural Network 14
3.4 Bug Report Quality Assessment 15

4 Results **19**
4.1 Deep Learning 19
4.2 Issue Vectorization 20
4.3 Feedforward Neural Network Classifier 20
4.4 Bug Report Quality 22

5 Conclusions and Future Work **26**

6 Appendix **28**
6.1 Additional results 28
6.2 Code repositories 29

7 Anleitung zu wissenschaftlichen Arbeiten **30**
7.1 Issue Vectorization module 30
7.2 Issue Analysis module 32
7.3 Issue Analysis browser extension 34

1
Introduction

In software development, issue reports provide information about the software's state and both completed and planned tasks. Issues are commonly categorized and managed in an issue tracking system. Specific categories vary from one project to another. However, issues can be separated into bug reports and non-bug reports. The former describes software failures, which require corrective code maintenance and the latter may refer to refactoring, documentation, adaptive maintenance, feature requests, and so on. Bug reports in particular contain information useful for maintenance activities, time and cost estimations and bug prediction. Unfortunately, not all bug reports are equally useful; they vary in their informational quality. Crucial information may be missing or incorrect. Thus, low-quality bug-reports slow down developers, since they need to spend more time identifying the problem.

Sometimes an apparent bug report is not describing a bug, but, for example, requesting a feature. This brings on another problem: Misclassification of issue reports. That is, a report classified as *bug* is in reality a *non-bug* and vice-versa.

In 2013, Herzig *et al.* [6] manually examined more than 7,000 issue reports of open-source projects and found 33.8% of bug reports to be misclassified. As a consequence, a bug prediction model, for instance, may predict locations which are likely to change more often instead of locations that are prone to errors. Thus, the quality of the data needs to be verified before being used in data mining and bug prediction models. But how can data quality be ensured? Classification of large datasets in a manual fashion is very time consuming and prone to human error. It would be desirable to solve these problems on the issue tracking systems themselves. Sometimes a middleman between reporter and developer called a bug triager is employed to resolve any misclassification

and to provide some quality control. But this process is time consuming as well and cannot provide information that is missing in a bug report. Therefore, an automated solution is needed to resolve misclassification and provide assistance in quality control.

In this thesis we address the two data-quality issues, namely the quality of bug reports and the misclassification of issue reports. We propose two approaches to tackle each problem:

Misclassification of issue reports: With the dataset provided by Herzig *et al.* [6], which contains both the original classification and the classification from their manual inspection, we trained neural networks in several variations: Binary classifier, multi-class classifier and a deep learning approach. Both the binary and the multi-class classifier were implemented as feedforward neural nets and outperform the deep learning approach. Out of all the variations, the binary classifier performed best with an accuracy of 82.9%.

Quality of bug reports: In 2010, Zimmermann *et al.* [14] investigated what composes a good bug report by conducting a questionnaire among developers. We use their findings to create an Bayesian expert system that can give a score from 1 to 5 to bug reports. The score indicates how useful the report in question is to a developer; a good bug report should provide all information that is useful for fixing the bug. To showcase a real world application, we implemented the *Quality Estimator*, a tool capable of rating bug reports and giving the user suggestions on how to improve the report.

Both the *Quality Estimator* and the feedforward classifiers are empowered by our *Issue Vectorization*, a heuristic model to create a vector representation of an issue report. Ultimately, we integrated the *Quality Estimator* and the binary classifier into a browser extension for Chrome and Firefox using the WebExtension API. The extension is primarily meant for reporters to help them write good bug reports. It provides insight into the issue type and improvement suggestions with just a single click.

In chapter 2 we take a look at related work. In chapter 3 we discuss the datasets and the models in detail. In chapter 4 we present our experimental setup and the results. And finally, in chapter 5 we discuss the results and conclude.

2 Related Work

2.1 Issue Report classification

In 2013, Herzig *et al.* [6] manually examined a large dataset of issue reports for misclassification to statistically analyze how misclassification could affect empirical bug prediction models. Their paper highlights the need for validation of such datasets, as their findings suggest, that about two fifths of issue reports are falsely classified and that every third bug report actually does not describe a bug. Our issue report classification approach is motivated by their work and uses the provided dataset.

Dommati *et al.* [4] presented an approach for both binary and multiclass classification of issue reports by type in their paper from 2012. They focused on networking bugs and applied domain specific feature-extraction. Classification was done using a naïve Bayes approach. Their two class classification model reached an accuracy of about 42% whilst the multiclass classifier only reached 15%. The accuracy was calculated by using the confusion matrix.

One of the older papers investigating different approaches to classify issue reports into bugs and non-bugs was written by Antoniol *et al.* [1]. The authors compared the performance of a naïve Bayes classifier, logistic regression and alternating decision trees in a classification task. One of the main objectives of the authors was to find out whether or not the text content of an issue report is enough to classify it. According to their findings it is for a binary classification task. For training, they randomly selected issues from the three open source projects Mozilla, JBoss and Eclipse and manually classified them.

Sohrawadi *et al.* [11] conducted a comparison of five different machine learning algorithms to distinguish bugs from other types of issues: Naïve Bayes, kNN, Pegasos, Rocchio and Perceptron. Of most interest to us is Perceptron, as it is a classifier using neural networks. The authors used the dataset from [6], same as we do but consider only the two classes “bug” and “no bug”. The text was extracted from the issue reports and stop words were removed. With stemming they generated a bag of words which was used for training. This differs from our approach as we use a heuristic model for feature extraction. Of the five algorithms, Perceptron performed the best with an accuracy of about 72% at average. This is close to the performance of our best multiclass model, which has an average accuracy of 74%. However, our best model for binary classification achieves an average accuracy of 83%.

2.2 Bug Report Quality

Surprisingly, not much work has been done on assessing the quality of issue reports.

In 2007, Bettenburg *et al.* [2] conducted a survey among developers of Eclipse to determine the information in bug reports that is considered the most useful. The authors implemented a prototype of a tool to automatically assess the quality of a bug report. Their tool determines the score based on the Eclipse guidelines on how to write bug reports. However, the tool considers only the text in a bug report.

Zimmermann *et al.* continued on the works of Bettenburg *et al.* [2] and performed a questionnaire not only among developers but also among users of Apache, Eclipse and Mozilla “to find out what makes a good bug report” [14]. The results showed that users do not provide as information what is most helpful for developers. There is a discrepancy between what users consider to be important and what developers want. This results in bug reports with missing or inaccurate information. On the basis of the questionnaire, the authors created a list of items of a bug report and how helpful the they are for developers. This work is the basis of our bug report quality model.

Schugler *et al.* [10] expanded upon the quality indicators identified by Bettenburg *et al.* [2] and trained a naïve Bayes classifier to assign one of five quality levels to a bug report. Their model performed very well when classifying a bug with a high quality. However, the performance was considerably worse in the rest of the cases. Our tool has the advantage of being able to give improvement suggestions to the user.

A common practice to process bug reports on tracking systems is to employ a triager taking the responsibility to decide the priority of bugs and assign them to the developers. A good written bug report is easier to assign to the right developers. An important factor influencing the priority is the severity of the bug report [7][13], that is, the impact a bug is expected to have on a software project. Prediction of the severity of a bug is a well studied subject [7][8][9][13]. Yang *et al.* [12] explored the use of quality indicators to improve the performance of severity prediction, namely stack traces, steps to reproduce,

attachments, and the length of the report. All but the last indicator are used by us as well to assess bug report quality. Thus, we believe that our approach to assess report quality would benefit severity prediction as well.

3

Issue Report Assessment with Machine Learning

In this chapter we will introduce the different models created for classification and issue report quality assessment. But first, we will take a look at the used data and the anatomy of issue reports.

3.1 Dataset

Herzig *et al.* [6] provide a dataset with over 7'000 issues from 5 open-source Java projects. Each issue was manually examined and assigned an issue type, that is, the classification or category of the issue report. Each issue report also has the issue type from the issue tracking system. To avoid confusion, we will refer to the issue type from the paper as classified issue type and to the issue type from the issue tracker as original issue type. The issue reports taken from the five open-source projects are shown in Table 3.1.

When doing any sort of classification one must decide on a set of categories, or issue types in our case. We already introduced a coarse classification of issue reports into bugs and non-bugs. But often it is useful to have a finer distinction since that helps maintainers to manage a software project. For this reason, many issue tracking systems allow for user defined sets of issue types on a project basis. This also means there is no universal definition of a certain issue type. To overcome this challenge and the fact that the projects in Table **??** do not all have the same set of issue types, Herzig *et al.* [6] defined their

Project	Number of reports	BUG	RFE	IMPR	DOC	REFAC	Other
HTTPClient	**746**	304	159	126	68	15	74
Jackrabbit	**2,402**	938	390	496	65	77	418
Lucene-Java	**2,443**	697	515	430	151	152	498
Rhino	**584**	302	66	60	0	1	155
Tomcat5	**1,226**	673	91	160	138	4	160

Table 3.1: Software projects and the contributed number of issue reports per type, as assigned by Herzig *et al.* [6]. See Table 3.2 for the type definitions.

own set of issue types: Bug, Feature Request, Improvement Request, Documentation, Refactoring and Other shown in Table 3.2.

While inspecting some issues ourselves we found that we do not always agree with the issue type assigned by Herzig *et al.* [6]. We manually classified 60 issue reports from the dataset and disagreed on 10% with the classification in the paper. We are positive this is due different interpretations of the reports and the issue types themselves. The issue HTTPCLIENT-630, for example, has the issue type *BUG* assigned on the issue tracking system. This issue reports wrong access modifiers that prevent methods from being invoked. It was classified as *Other* in the paper, whilst we agree with the classification on the tracker. There are many reasons why misclassification can happen. Herzig *et al.* [6] inspected the sources of misclassification in detail. Some of them reflect our own findings from inspecting the reports:

- Different interpretations and definitions of issue types.
- Default (pre-set) issue type of the issue tracking systems. The reporter does not always bother to change the issue type or overlooks the option.
- Actual issue type becomes clear only during resolution but does not get changed.
- A suitable issue type is not available; the reporter assigns the issue type according to best guess.

We experimented with the dataset in various partitions shown in Table 3.3. The issue type *Other* was excluded from all variations because it is ambiguous and can be expected to confuse the neural network during training. *DOC* and *REFAC* were excluded as well; there were too few issue reports of these types. In order to further minimize noise in the dataset we decided to a consider a variation with only the issue reports included where the issue type from [6] matches the original one from the tracking system; in other words: only the issue reports with a type everyone agrees on.

Issue Type	Description
BUG	Issue reports documenting corrective maintenance tasks that require semantic changes to source code.
RFE	Issue reports documenting an adaptive maintenance task whose resolving patch(es) implemented new functionality (request for enhancement; feature request).
IMPR	Issue reports documenting a perfective maintenance task whose resolution improved the overall handling or performance of existing functionality.
DOC	Issue reports solved by updating external (e.g. website) or code documentation (e.g. JavaDoc).
REFAC	Issues reports resolved by refactoring source code. Typically, these reports were filed by developers.
Other	Any issue report that did not fit into any of the other categories. This includes: reports requesting a backport (BACKPORT), code cleanups (CLEANUP), changes to specification (rather than documentation or code; SPEC), general development tasks (TASK), and issues regarding test cases (TEST).

Table 3.2: Issue types defined by Herzig *et al.* [6] and used for manual classification. The table is adapted from Herzig *et al.* [6]. The issue types *DOC*, *REFAC* and *Other* were excluded from the dataset: *Other* is ambiguous and can be expected to confuse the neural network; *DOC* and *REFAC* were excluded because there were too few issue reports of these types.

	Description	Partition size	Source of classification
P1	All issue reports (excluding *Other*, *DOC* and *REFAC*) with the classification of [6].	5655	Classified issue type
P2	All issue reports (excluding *Other*, *DOC* and *REFAC*) with the original classifications from the issue tracking system.	6956	Original issue type
P3	Only issue reports whose original issue type matches the type assigned by [6]. This restricts the dataset to the issue types BUG, IMPR and RFE.	4072	Both

Table 3.3: Partitions of the dataset.

3.2 Issue Report Anatomy

The dataset of Herzig *et al.* [6] provides us with a list of issue identifiers and both the classified and original issue type. However, it does not contain any information about the content of the issue reports. All projects use either Jira or Bugzilla as an issue tracking system and provide public access to the issue database. Using the issue identifiers we queried the issue reports from these systems, either using REST (Jira) or XML-RPC (Bugzilla). The reports were stored as JSON files in a structure that suits our needs. Depending on the issue tracking system the reports can vary in their structure but share common informational elements:

- **Summary**: A short summary of the report. It often acts as the issue title.
- **Description**: Description of the problem.
- **Comments**: Discussions between developers and reporters. They often contain additional information.
- **File attachments**: Source code patches, text files or screenshots.
- **Meta information**: Metadata about the issue report.
  - Software version
  - Used hardware and operating system

- Severity and priority of the issue
- Issue identifier
- Software product and component
- Issue type
- Status of the issue: Open, closed, resolved etc.

3.3 Issue Reports Classification

To avoid bias, issue report datasets used for studies that rely on the categories in issue trackers, should be validated. This can be done through manual inspection in the manner Herzig *et al.* [6] did . However, manual inspection is time consuming and expensive. Given the large size of the datasets an automatic validation would be useful. This can be achieved with a classification model. But to be of any use, the model must surpass human performance. According to Herzig *et al.* 33.8% of bug reports were not filed as bugs, thus the reporters filed the bug reports with a precision of 66.2% [6]. The precision of reporters will be used as baseline and our goal is to achieve a significantly better performance. In this section we explore different approaches to solve this classification problem. Since we have a labeled dataset, a suitable method is to train neural networks for the classification problem in a supervised manner.

3.3.1 Deep Learning

In the past, deep learning has proven to be capable of outperforming human experts in tasks such as natural language processing. Using DL4j[1], a deep learning framework for Java, we implemented a multiclass classifier. We have chosen Dl4j since it already ships with means to classify labeled documents: a neural net called Doc2Vec[2]. Conveniently, feature extraction and vector creation are handled by the framework. To accommodate to the fact that Doc2Vec is designed to classify text documents, a textual representation of the reports was created, including summary, description and comments. Meta information was left out to avoid confusing the network. However, this approach proved to be unsuccessful for classifying issue reports; we were unable to achieve satisfactory results.

3.3.2 Issue Vectorization

Natural language must be transformed into a form neural nets can understand. Doc2Vec handles this internally, but since this approach was not successful we decided to devise

[1]Deep Learning for Java, 12.12.2017, https://deeplearning4j.org/

[2]Doc2Vec, or Paragraph Vectors, in Deeplearning4j, 12.12.2017, https://deeplearning4j.org/doc2vec

our own system for feature extraction. To find the features, we manually examined 50 issue reports (10 for each issue type). While manually inspecting the issue reports we kept the question in mind: “From the perspective of a human, what reveals that this report is of a particular type”. This process has yielded a list of common items for each issue type. An item can be a simple keyword or an *informational element* such as a code snippet or a stack trace. There is an overlap between this list and the items that indicate an issue type at various strengths. For example, the presence of a stack trace is a strong indication that the issue in question describes a bug. The items were sorted thematically and based on their correlation into groups. Such a group composes a feature and we say that an issue report has a particular feature, if one or more of the items of the underlying group is present. Sorting items in such manner instead of using the lists as groups has the advantage that it yields more features. We expanded our feature set with informational elements from the paper by Zimmermann *et al.* [14] for content quality assessment. Some features are composed entirely of keywords. The chosen keywords were not processed, that is, neither stemming nor automatic pluralization was applied. The keywords are detected only when present as whole words, but are case insensitive. In the following list all features and their detection heuristics are described. Features taken from the paper by Zimmermann *et al.* [14] are marked with (*).

Error This feature indicates a software failure. It is detected by stack traces, references to exception classes and keywords related to software failure.

- **Keywords** *exception* and *failure*
- **Reference to exception classes** In Java, the names of exception classes are generally of the form “NameOfAnException” and thus, can easily be detected by using regular expressions. Example: `ClassCastException`
- **Stack trace** A stack trace always contains the name of an exception followed by multiple lines starting with the word “at” and are detected using regular expressions as well. Example:

```
Exception in thread "main" java.lang.ClassCastException: ...
      at org.mozilla.javascript.ScriptRuntime. ...
```

Operational element This feature is said to be present, if the text contains a reference to a specific location in the code or a snippet of a log.

- **Reference to code** We define a reference to a code location to be either the name of a method, or a the name of class. These are detected using regular expressions. However, exception classes are excluded as these are already covered by the *Error* feature. Methods are detected by looking for a word of the form “nameOfAMethod” followed by opening and closing parenthesis. Examples: `PrintWriter`, `checkCompile()`, `equals (Object obj)`, `org.apache.catalina.startup.Bootstrap`

Log excerpt Log messages usually contain a timestamp, the log level, the class name and the message. However, the format can vary from application to application. Nevertheless it is possible to detect log lines using regular expressions. Example:

```
12:35:03.076 [main] INFO ch.unibe.classifier.IssueClassifier
```

Code snippet We defined three rules to identify code examples: An opening curly bracket followed by a semicolon, assignment of a value to a variable and object instantiation. The patterns are detected using regular expressions. Examples:

```
new URIBuilder("http://www.example.com").addParameter("foo", "bar"),
try { ctxt.compile(); ,Matcher matcher = pattern.matcher(text);
```

Documentation A keyword-only feature indicating the mentioning of documentation. The keywords are: *doc*, *document*, *documentation*, *logo*, *image*, *link* and *javadoc*.

Enhancement A keyword-only feature indicating the task of improving or optimizing existing functionality. The keywords are: *improvement*, *improve*, *optimization*, *optimize*, *performance*, *overhead*, *logging*, *speedup*, *easier* and *useful*.

Request A keyword-only feature indicating feature requests. The keywords are: *feature*, *features*, *support*, *implement* and *provide*.

Visibility A keyword-only feature indicating a discussion about Java access modifiers. The keywords are: *visibility*, *private*, *public*, *protected* and *package private*.

Action A keyword-only feature indicating a refactoring task. The keywords are: *refactor*, *refactoring*, *rename*, *renaming*, *extract*, *extracting* and *remove*.

Naming A keyword-only feature indicating a discussion about naming. The keywords are: *name*, *naming*, *naming convention* and *simple name*.

Implementation A keyword-only feature with keywords commonly used when discussing implementation details of Java programs. The keywords are: *subclass*, *subclassing*, *extend*, *extends*, *overload*, *overloading*, *overloaded* and *interface*.

Test case A keyword-only feature indicating the discussion or presence of test cases. The keywords are: *test case*, *test cases*, *testcase*, *testcases* and *@Test*.

Reproduction This feature indicates the presence of instruction on how to reproduce a fault. It only uses keywords. We experimented with the detection of the feature by the presence of lists, but this has yielded many false positives. The keywords are: *steps to reproduce*, *how to reproduce* and *reproduction*.

Link This feature indicates the presence of a hyperlink. We are using regular expressions to detect hyperlinks and the file extension ".html" in the text body.

Patch This feature indicates the presence of a software patch as attachment. Some projects include this information in the metadata of the report. If this is not the case, the attachments of the issue report need to be inspected. This was done while querying the reports from the issue tracking systems.

Screenshot This feature indicates the presence of a screenshot as attachment. The attachments of a report where examined for image files. This was done while querying the reports from the issue tracking systems.

System specification A feature indicating the presence of a information about the platform, operating system or hardware. System specification are sometimes in the metadata of the report, but can also be specified in the text body. The keywords are: *windows*, *win10*, *win7*, *xp*, *linux*, *ubuntu*, *debian*, *fedora*, *mac*, *nvidia*, *amd*, *ati*, *intel*, *os* and *operating system*.

Version This feature is considered present if the metadata of the report contains information about the software versions affected by the issue.

High priority This feature is considered present if the metadata of the report contains information about the severity and/or priority of the issue. Since some issue trackers combine priority and severity, we decided to do the same. Examples: An issue with priority "highest", "high" or severity "critical".

Low priority * Same as the high priority feature, but indicates low or normal priority or severity.

Expected behavior * A keyword-only feature a description of how the software is expected to behave. The keywords are: *expected*, *expected behavior* and *expected behaviour*.

Observed behavior * A keyword-only feature a description of how the software is actually behaving. The keywords are: *observed*, *observed behavior*, *observed behaviour*, *actual behaviour*, *linux*, *ubuntu*, *debian*, *fedora*, *mac*, *nvidia*, *amd*, *ati*, *actual behavior*, *actual result* and *actual results*.

Component * This feature is considered present if the metadata of the report contains information about the software component affected by the issue.

Product * This feature is considered present if the metadata of the report contains information about the software product affected by the issue.

In order to extract these features from an issue report, the meta information as well as the textual information must be processed. To do this, we created a heuristic model in which each feature is detected by a dedicated heuristic, shown in Figure 3.1. The feature

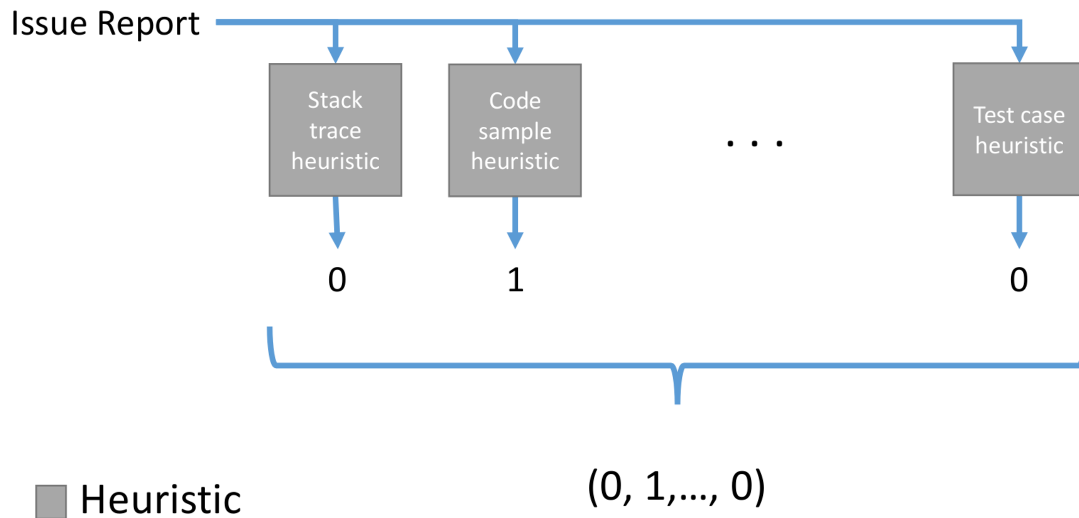


Figure 3.1: Heuristic model for vectorizing an issue report.

detection is done in a binary fashion; ether the feature is present or not. So the heuristic model takes an issue report as input and outputs a binary vector representation, hence the name *Issue Vectorization*.

3.3.3 Feedforward Neural Network

Having the means to extract features from issue reports, we created a simple feedforward neural network. We used again DL4j to implement the model and trained it on the dataset partitions shown in Table 3.3. The network has two hidden layers with 25 nodes each. The input layer has a 23 nodes (equals the length of the feature vector). As activation function, that is, the function that defines the output [5], we used *the hyperbolic tangent* (tanh) for the input and hidden layers. The output layer was set to use *softmax*, also known as *normalized exponential* [3] as activation function. As a loss function we used *negative log likelyhood*. The loss function is used as a way to penalize the output, meaning it indicates the magnitude of error the network made on its prediction [3]. We trained the network in a supervised manner using the issue types as labels (the available issue types vary depending on the chosen dataset partition). We also experimented with just the distinction between bugs and non-bugs, that is, all issue types other than bug were mapped to non-bug (see Figure 3.2). This mapping can be applied to all dataset partitions.

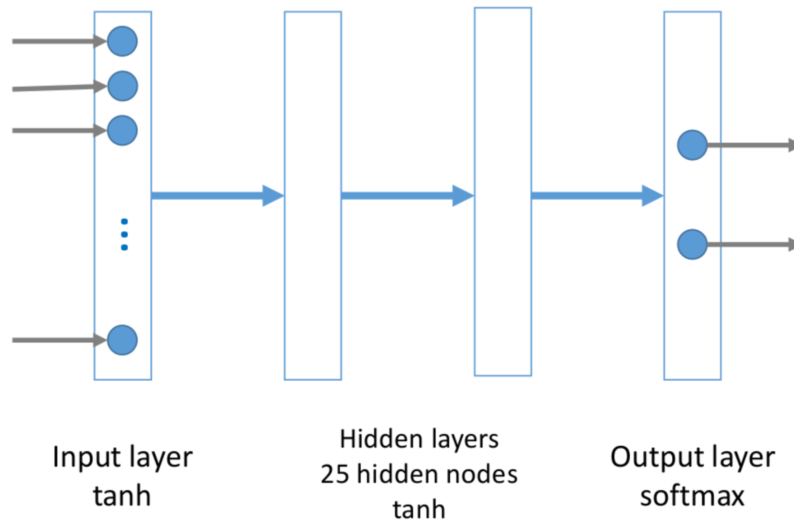


Figure 3.2: Architecture of our feedforward neural network in a configuration for binary classification. The dataset is mapped to two labels: Bugs and non-bugs.

3.4 Bug Report Quality Assessment

Not only misclassification of issues but also the quality of their content has an impact. Developers have to spend more time finding the issue, if confronted with low quality issue reports, ultimately increasing the time needed to fix the issue. But low content quality also negatively affects bug triaging, datasets and so on. Of special interest to us is the quality of bug reports. From the perspective of a developer, a good bug report is a useful one, that is, the bug report provides enough and correct information about the problem, how to reproduce it and possibly even its cause. Thus, developers are not slowed down by misleading or missing information. In this section, we present a model capable of assessing the quality of bug reports. The model assigns a 1 to 5 rating to a bug report and can provide a list with missing informational elements. In 2010, Zimmermann *et al.* [14] conducted a questionnaire among both developers and reporters of open source projects by Apache and Mozilla (HTTPClient, Jackrabbit, Lucene-Java, Rhino and Tomcat), to find out what information a good bug report should contain. We are especially interested in the developer's point of view, since they are the ones confronted with bad quality bug reports the most. In one of the questionnaires the developers were asked to name information provided by reports that they found most helpful. In Table 3.4 the items are sorted according to how helpful for developers they are when fixing bugs. All items correspond to one or more features (see Section 3.3.2) and thus, to the respective heuristics of the issue vectorization. "Summary" and "build information" was

Informational element	Importance
Steps to reproduce	83%
Stack traces	57%
Test cases	51%
Observed behavior	33%
Screenshots	26%
Expected behavior	22%
Code examples	14%
Summary	13%
Version	12%
Error reports	12%
Build information	8%
Product	5%
Operating system	4%
Component	3%
Hardware	0%
Severity	0%

Table 3.4: Results from the questionnaire conducted by Zimmermann *et al.* [14]. Information most helpful for developers. The right column represent the importance of an item for developers when fixing a bug.

not used by us since the first is always present in our dataset and we could not find any occurrence of the latter.

We can consider developers to be experts concerning the quality of the content of a bug report. Thus, we decided to emulate their quality rating by creating an expert system with the knowledge derived from Table 3.4. We modeled this expert system as a Bayesian probabilistic model, similar to a Cause-and-Effect network. The items in Table 3.4 are represented by item nodes with the respective importance from the right column as value. All binary nodes are connected to a central node whose calculated value is the output of the network. An item node is mapped to one or more components of the feature vector and can adopt one of two states: active or inactive. A node is activated when at least one of the mapped vector components is equals 1, that is, the underlying feature of the issue under inspection is present. Inactive item nodes do not contribute to the calculation of the central node's value: Let N be the set of all item nodes and v be the feature vector, then we have $A = \{n \in N : n \textit{ is active under } v\}$ the set of all active item nodes. The value of the central node is calculated as follows:

$$Usefulness(v) = 1 - \prod_{n \in A}(1 - Importance(n))$$

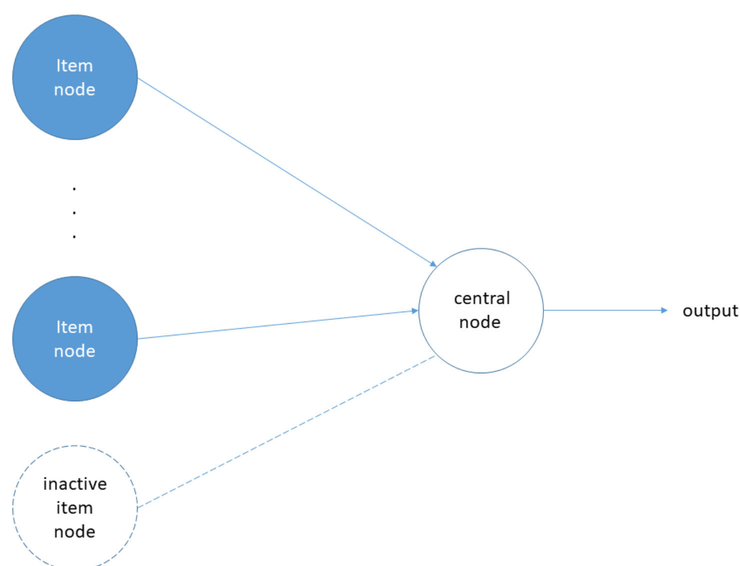


Figure 3.3: Architecture of our expert system. Each informational element is modeled as item node. Activated item nodes contribute to the ouput calculation. A node is activated when at least one of the mapped features is present.

Where *Importance(n)* is the importance of the item represented by node *n* for developers. All in all, the network takes a vector representation of a bug report as input and outputs a value in $(0, 1)$ representing the *usefulness* of the bug report to developers.

However, without context the output of the network does not reveal much information about the actual quality of the report. In addition to the questionnaires to identify the importance of informational elements, Zimmermann *et al.* [14] asked developers to rate specific bug reports by assigning a discrete score from one to five (see Figure 3.4). From their work, we derived the distribution of the number of bug reports over the set of scores. To create a mapping function, we calculated the *usefulness* of all bug reports from the dataset partition with matching issue types (see Table 3.3) and applied the distribution (see Figure 3.4). This has yielded the thresholds for the mapping function

$$Score(u) = \begin{cases} 1 & \text{if } u < 0.190 \\ 2 & \text{if } 0.190 \leq u < 0.603 \\ 3 & \text{if } 0.603 \leq u < 0.933 \\ 4 & \text{if } 0.933 \leq u < 0.988 \\ 5 & \text{if } u \geq 0.988 \end{cases}$$

where u is the *usefulness*. A score of 1 corresponds to a “very bad” bug report and a score of 5 to a “very good” one.

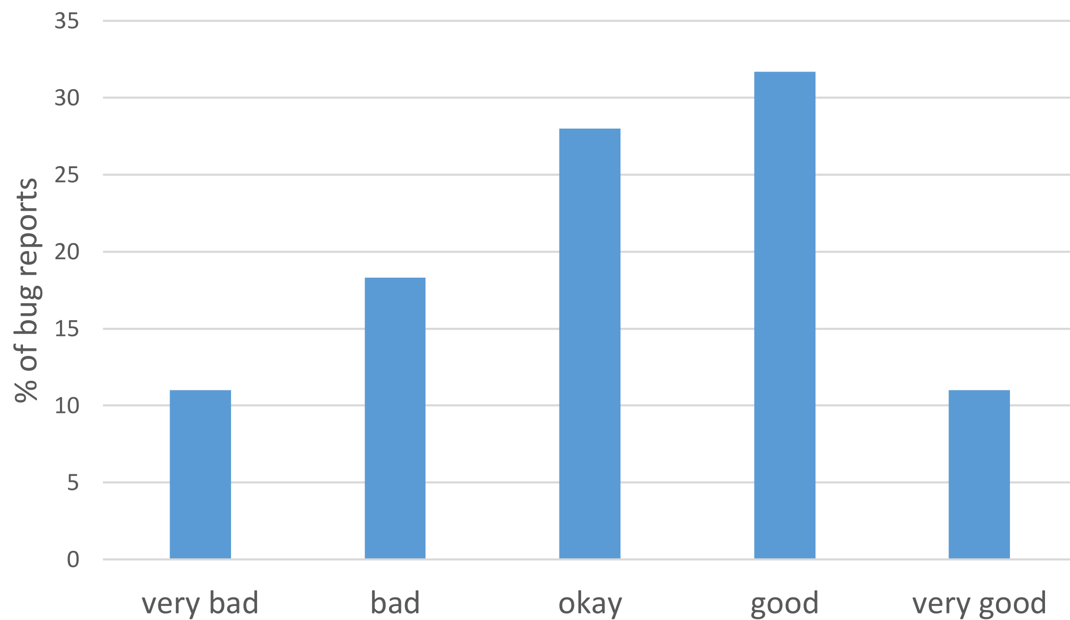


Figure 3.4: Zimmermann *et al.* [14] asked developers from the Apache, Mozilla and Eclipse projects to rate bug reports by giving them a score from 1 to 5, where 1 is "very bad" and 5 "very good". 289 randomly selected bug reports were rated with a total of 1,244 votes. This figure shows the distribution of the votes by developers. We calculated the distinct *usefulness* of all bug reports in the dataset partition P3. The elements of the resulting set can be sorted by value. The 11% smallest values correspond to the quality "very bad" and so on. By splitting the set in this manner we obtain the thresholds of the mapping function.

4 Results

We conducted a series of experiments for our models to validate their performance. In this chapter we will take a look at the results and elaborate on the experimentation setups. All training of the machine learning models was performed on an Intel i7 Skylake @2.6GHz CPU. Information about experiment duration therefore has to be interpreted accordingly. The four standard measures precision, recall, accuracy and f1 score were calculated by using the respective confusion matrix of the model.

4.1 Deep Learning

For validation purposes, the classifier was trained on the dataset partition containing only matching issue types (see Table 3.3). 25% of the reports of each issue type were reserved for testing. The dataset was split randomly for each run. We trained the classifier for 100 epochs and 1000 epochs with a batch size of 1000 samples. One epoch consists of one full training cycle. The average duration of one training run with a cycle of 100 epochs was 4 minutes and for 1000 epochs 42 minutes. The initial learning rate was set to 0.15 with a minimum value of 0.1.

As seen in Table 4.1, this model does not perform well in a classification task. Almost all issue report were classified as bugs. We abandoned this approach in favor of our feedforward network.

	Precision	Recall	f1 Score
BUG	**0.741** / 0.711	0.986 / **0.999**	0.819 / **0.822**
RFE	**0.095** / 0.000	**0.003** / 0.000	**0.005** / 0.000
IMPR	**0.127** / 0.033	**0.011** / 0.000	**0.018** / 0.001
mean	0.348 / 0.459	0.335 / 0.333	0.335 / 0.374

Table 4.1: Average results of 10 independent runs of the Doc2Vec multiclass classifier. The left value refers to results for 100 and the right for 1000 epochs. Changing the number of epochs didn't improve the results significantly. Trained on the dataset partition P3. Whilst the results for bug reports are good, the model failed to reliably classify non-bugs.

4.2 Issue Vectorization

To validate our heuristic vectorization model, we randomly selected 50 issue reports, 10 from each software project. We manually inspected the selected issue reports and created the feature vectors. Table 4.2 shows the accuracy, precision and recall of the feature heuristics. Of 23 feature heuristics, 8 of them are based on keywords only.

The manual inspection was done online on the respective issue tracking system and not with the version in our database, since the online report is the one the developers actually use. This explains the poor result for detecting patches; it was not possible to automatically and reliably retrieve information about patches in all instances. Overall, our issue vectorization model reaches an accuracy of 0.988. However, it needs to be noted that the model is tailored specifically to the five projects and all of them use Java as main language. We expect the model to perform worse when applied to projects using other languages, since the heuristics are implemented using regular expression specifically tailored to Java.

We provide a Java library[1] so that our model can be adapted for other projects.

4.3 Feedforward Neural Network Classifier

Validation of the feedforward neural network was done in a similar fashion as the deep learning approach. We trained the model either as binary or multiclass classifier on each of the dataset partitions (see Table 3.3), resulting in 6 combinations. The dataset was balanced and randomly split into test and train set for each run. The test set contains 25% of each issue type (after balancing), that is, the set contains the same amount of reports of each issue type. To prevent overfitting, early stopping[2] was used with a maximal amount

[1] Issue Vectorization library, https://github.com/curtys/issue-vectorization
[2] Early Stopping, 12.12.2017, https://deeplearning4j.org/earlystopping

Heuristic	Accuracy	Precision	Recall
Patch	0.907	1.0	0.848
Code snippet	0.943	0.923	0.962
Operational element	0.963	0.962	1.0
Reproduction	0.963	0.75	1.0
Expected behavior	0.981	0.9	1.0
Error	0.981	0.952	1.0
Link	1.0	1.0	1.0
System specification	1.0	1.0	1.0
High priority	1.0	1.0	1.0
Low priority	1.0	1.0	1.0
Screenshot	1.0	1.0	1.0
Version	1.0	1.0	1.0
Component	1.0	1.0	1.0
Product	1.0	1.0	1.0
Test case	1.0	1.0	1.0
Naming	1.0	1.0	1,0
Request	1.0	1.0	1.0
Documentation	1.0	1.0	1.0
Action	1.0	1.0	1.0
Enhancement	1.0	1.0	1.0
Implementation	1.0	1.0	1.0
Visibility	1.0	1.0	1.0
Observed behavior	1.0	1.0	1.0

Table 4.2: Accuracy, precision and recall of the feature heuristics. Across all 23 heuristics the issue vectorization achieves an accuracy of 0.988.

of 1000 epochs and 1 iteration. The learning rate was fixed to 0.1. The training set was not split into batches. The training was repeated for 100 independent runs, that is, the dataset was shuffled and split anew after each run. The duration of one run is 90s at average.

Table 4.3 shows the average results of 100 independent runs of the multiclass classifier trained on the dataset partition P3 (Table 3.3). Though we see a decrease in recall compared to the deep learning model (see Table 4.1), the f1 scores of all classes have improved. Whilst the results for BUG are above the baseline, the two other classes are not. In order to increase the performance of the classification further, we trained a model for binary classification. In this configuration the neural network has the same architecture, particularly the output layer is uses still softmax as activation function. Training was also done on the dataset partition P3. However, by mapping all issue types other than BUG to the synthetic type NONBUG, the dataset is restricted to two classes. As we see in Table 4.4, whilst the precision of the class BUG is lower, the overall model performance is increased. In this configuration the classifier looses its ability to distinguish non-bug issue types. Since our main interest lies on the issue type BUG, we consider this an acceptable tradeoff.

On all three dataset partitions in Table 3.3 a model was trained with and without the binary mapping; resulting in six variations. Table 4.5 shows the average results of all models, inclusive the model of the deep learning approach. Of all models, the binary classifier trained on P3 achieved by far the best performance. Compared to its multiclass counterpart (see *Multiclass P3*), the binary mapping resulted in a 17% increase of the f1 score. Interestingly, the multiclass model trained on P1 (classification by Herzig *et al.* [6]) scored lower than the multiclass model trained on P2 (original classification from the tracking systems) with a difference of 6.6% in respect to the f1 score. Chapter 6 contains a detailed listing of the results of all models not shown in this chapter.

Both, *Binary P3* and *Multiclass P3* classifiers were integrated into our *Issue Analysis module*[3] , that is, a Maven module providing an API for the assessment of issue reports. We implemented a browser extension that makes use of the module (see section 4.4).

4.4 Bug Report Quality

We did not perform an empirical validation of our model for quality assessment since we could not retrieve the original dataset from Zimmermann *et al.* [14]; our attempts to contact the authors failed. The *Quality Estimator* is part of our *Issue Analysis module* as well, as the classifier. However, we also implemented a *Issue Analysis browser extension*[4] (see Figure 7.3) for qualitative validation and to showcase a real world application. When activated on an online bug report, the browser extension displays a score from 1 to 5 and

[3]Issue Analysis module, https://github.com/curtys/issue-analysis

[4]Issue Analysis browser extension, https://github.com/curtys/issue-analysis-webextension

	Precision	Recall	f1 Score	AUROC
BUG	0.908	0.803	0.851	0.884
IMPR	0.533	0.696	0.6	0.867
RFE	0.507	0.534	0.489	0.857

Table 4.3: Results of the multiclass classifier trained on the dataset partition with matching issue types (P3, 3.3). The model performs well at identifying bug reports. Classification of improvement and feature requests performed much better compared to the deep learning model.

	Precision	Recall	f1 Score	AUROC
BUG	0.886	0.87	0.87	0.89
NONBUG	0.753	0.747	0.747	0.89

Table 4.4: Results of the binary classifier trained on the dataset partition with matching issue types (P3, 3.3).

Model	Accuracy	Precision	Recall	f1 Score
Deep learning 100	0.693	0.348	0.335	0.335
Deep learning 1000	0.698	0.459	0.333	0.374
Multiclass P1	0.573	0.49	0.579	0.531
Multiclass P2	0.686	0.58	0.615	0.597
Multiclass P3	0.744	0.632	0.65	0.64
Binary P1	0.753	0.752	0.753	0.752
Binary P2	0.773	0.77	0.773	0.771
Binary P3	**0.829**	**0.812**	**0.809**	**0.81**

Table 4.5: Results of all classification models.

gives improvement suggestions to the reporter in the form of a list of absent informational elements that could be added to improve the score. During manual examination, the browser extension rated the bug reports similar or equal to our own assessment. The browser extension is implemented in TypeScript[5], which compiles to Javascript. It uses the WebExtension API[6] and is compatible with Firefox and Chrome. Currently, the issue tracking systems Jira and Bugzilla are supported. The browser extension itself only collects the information from the page and sends it to a server running the *Issue Analysis module*. We also provide webapps implementing the required service[7] [8]. The browser extension was implemented in this manner, because modern browsers do not support execution of Java code natively.

[5]TypeScript, 20.01.2018, https://www.typescriptlang.org/
[6]WebExtension API, 20.01.2018, https://developer.mozilla.org/en-US/Add-ons/WebExtensions
[7]Issue Analysis service, https://github.com/curtys/issue-analysis-service
[8]Issue Analysis service embedded, https://github.com/curtys/issue-analysis-service-embedded

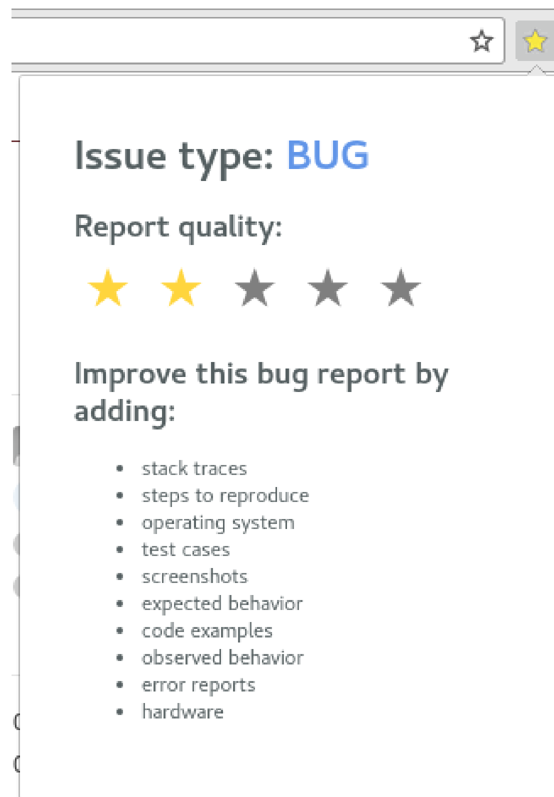


Figure 4.1: The *Issue Analysis browser extension* is activated with a click on the star symbol. A popup appears, showing the assessment results of the issue report in question. The extension makes use of the binary classifier from the *Issue Analysis module* (running on a server). When activated on a bug report, a score and a list of missing elements is given.

5 Conclusions and Future Work

Issue reports are used by many peoples for many different purposes, from project management to bug prediction. No matter the use case, low quality issue reports have the potential to negatively affect the process in which they are involved. We examined two problems concerning issue reports: Misclassification and content quality. Misclassification of issue reports introduces bias in studies relying on the issue type, but also impairs assignment of the report to the right developer and hinders finding duplicates. The quality of bug report directly affects the process of fixing the respective bug. Crucial information may be missing from the report or be misleading. The developer then needs to spend more time identifying the actual issue. For each of the underlying problems we propose an approach to mitigate the negative impacts.

Misclassification of issue reports Using a labeled dataset we trained a neural network to classify issue reports by type. Our goal was to top the performance of human reporters, which have a precision of about 66% when deciding if a issue is a bug or not. The best model, trained to classify issue reports as *bugs* or *non-bugs*, achieves a precision of 81.2%. Thus, our model outperforms reporters in that task. We also trained a multiclass classifier, capable of classifying issues into the three classes *bug*, *feature request* and *improvement*. This model did not match the performance of the binary classifier; but multiclass classification is also a more complex task.

Quality of bug reports Using the knowledge of developers about the importance of specific elements of a bug report, we created the *Quality Estimator*. Our goal was to provide a tool capable of rating the bug report and giving improvement suggestions. The *Quality Estimator* does that, but unfortunately we were unable to

empirically evaluate how accurately the rating is. However, this does not affect the capability to give improvement suggestions.

The *Quality Estimator* as well as the classification models use the *Issue Vectorization* to create vector representations of issue reports. The *Issue Vectorization* is specifically tailored to Java and the five open-source projects. One can expect it not to work as well when applied to other languages. However, we are positive that the concept is solid and can be adapted to other programming languages by changing the detection heuristics. We did not investigate what effects cultural differences of projects have on the vectorization and any of the models. Though the vectorization performed well in the evaluation, it is possible that some heuristics do not detect their feature accurately. Features like "observed behavior" are difficult to detect, since they can be in a form, which requires some interpretation of the text. Despite its limitations, we believe that our *Issue Analysis* tool would be of help for bug triaging and quality control on issue tracking systems.

The contributions of this thesis are as follows: We have shown that it is possible to extract features from issue report text and meta information with minimal preprocessing using a heuristic model. We compared models trained on the same dataset but with different labeling, illustrating the importance of proper labeling for supervised learning. We provide viable solutions for both the automatic validation of large datasets on misclassification and content quality assessment for bug reports.

Extending the proposed approaches to other languages and projects remains a topic for future research, as well as exploring the possibility to integrate the provided tools into issue tracking systems.

6 Appendix

6.1 Additional results

	Precision	Recall	f1 Score	AUROC
BUG	0.778	0.678	0.724	0.825
IMPR	0.429	0.441	0.434	0.706
RFE	0.5	0.442	0.468	0.751
REFAC	0.252	0.757	0.368	0.905

Table 6.1: Results of the *Multiclass P1* model.

	Precision	Recall	f1 Score	AUROC
BUG	0.86	0.753	0.801	0.852
IMPR	0.591	0.597	0.593	0.82
RFE	0.288	0.495	0.35	0.8

Table 6.2: Results of the *Multiclass P2* model.

	Precision	Recall	f1 Score	AUROC
BUG	0.775	0.749	0.762	0.826
NONBUG	0.73	0.756	0.743	0.826

Table 6.3: Results of the *Binary P1* model.

	Precision	Recall	f1 Score	AUROC
BUG	0.823	0.773	0.797	0.851
NONBUG	0.716	0.774	0.743	0.851

Table 6.4: Results of the *Binary P2* model.

6.2 Code repositories

Issue report dataset Dataset of all issue report queried from the issue tracking systems.
`https://github.com/curtys/issue-report-dataset`

Issue Vectorization Issue Vectorization library as Maven module.
`https://github.com/curtys/issue-vectorization`

Issue Analysis Quality Estimator and issue classifier as Maven module.
`https://github.com/curtys/issue-analysis`

Issue Analysis service A Tomcat webapp providing the functionality of the Issue Analysis as service.
`https://github.com/curtys/issue-analysis-service`

Issue Analysis embedded service Same as above but with an embedded Jetty server.
`https://github.com/curtys/issue-analysis-service-embedded`

Issue Analysis browser extension Browser extension for Chrome and Firefox. The extension needs a web service (one of the above).
`https://github.com/curtys/issue-analysis-webextension`

7 Anleitung zu wissenschaftlichen Arbeiten

7.1 Issue Vectorization module

The Issue Vectorization module provides functionality to create a vector representation of issue reports. This section will give a brief tutorial on how to use it. The code repository can be found here:
`https://github.com/curtys/issue-vectorization`

Create a vector representation of a single or multiple issue reports using the default configuration:

```
VectorizationEngine engine = VectorizationEngine.builder()
            .useDefaults()
            .build();

// Create a vector from a single issue report
Vector vector = engine.vectorize(issue);

// Batch processing is supported as well
List<Vector> vectors = engine.vectorize(listOfIssues);
```

In the above code example the vectorization will be done using the default vector components. However, it is possible to provide custom vector components:

```
VectorizationEngine.builder()
      .useComponents(listOfComponents)
      .build();
```

Sometimes it is necessary to prepare the issues before vectorization. This can be done using preprocessors and is especially useful for batch processing:

```
// Preprocessor to eliminate duplicate issues
ChainPreprocessor firstPreprocessor = new DuplicateFilter();
firstPreprocessor.addNext(secondPreprocessor);

VectorizationEngine.builder()
      .useDefaults()
      .preprocessor(firstPreprocessor)
      .build();
```

It can be useful to integrate the label, that is, the issue type (as integer value), into the resulting vector as well. For example, when exporting the results to CSV. In such a case, a label mapper must be provided, taking the responsibility to assign a integer value to label:

```
LabelMapper labelMapper = new DefaultLabels();

// add a label mapper and configure the engine
// to integrate the label into the vector
VectorizationEngine.builder()
      .useDefaults()
      .labelMapper(labelMapper)
      .integrateLabels(true)
      .build();
```

Full example: Issues in JSON format are loaded from the classpath. In this example, we want to use the issue type from the issue tracking system as ground truth. The first preprocessor does this. The second preprocessor removes all loaded issues which have non-matching labels (e.g. issue type from on the tracker is “improvement”, but the issue was classified as “bug”).

```
// Load issues from file
URI inPath = VectorizationExample.class
      .getResource("/example").toURI();
List<Issue> issues = JsonIO.readJsons(inPath);

// Setup preprocessors
// Set the label to use as ground truth
ChainPreprocessor preprocessor =
      new TrueLabelPreprocessor(Issue.LABEL_SOURCE_TRACKER);
// Remove all issues with non-matching labels
preprocessor.addNext(new MatchingLabelFilter());
LabelMapper labelMapper = new SimpleLabels();

// Configure engine
VectorizationEngine engine = VectorizationEngine.builder()
      .verbose()
```

```
        .integrateLabels(true)
        .preprocessor(preprocessor)
        .issues(issues)
        .useDefaults()
        .labelMapper(labelMapper)
        .build();

// Pre-process issues
engine.prepareIssues();

// Create vectors
List<Vector> vectors = engine.vectorize();
System.out.println(vectors);
```

7.2 Issue Analysis module

The Issue Analysis module provides two main functionalities:

- Issue report classification: A binary and a multiclass classifier are integrated. The binary classifier can distinguish bugs and non-bugs. The multiclass classifier supports the types *bug*, *improvement* and *feature request*.

- Bug report quality estimation: For a given bug report, a score can be calculated based on its content.

This section will give a brief tutorial on how to use it. The code repository can be found here:
`https://github.com/curtys/issue-analysis-webextension`

Use classifiers to predict the type of an issue report:

```
// instantiate a binary classifier
Classifier binaryClassifier = new BinaryClassifier();

// or a multiclass classifier
Classifier multiclassClassifier = new MulticlassClassifier();

Prediction prediction = binaryClassifier.query(issue);

// get the label with the highest probability
String type = prediction.getBestClassLabel();

// get the probability of a given label
float probability = prediction.probabilityOf(label);

// get all class labels
Set<String> labelSet = prediction.labels();

// get a map with all labels and their probabilities
Map<String, Float> propabilitiesMap = prediction.getProbabilities();
```

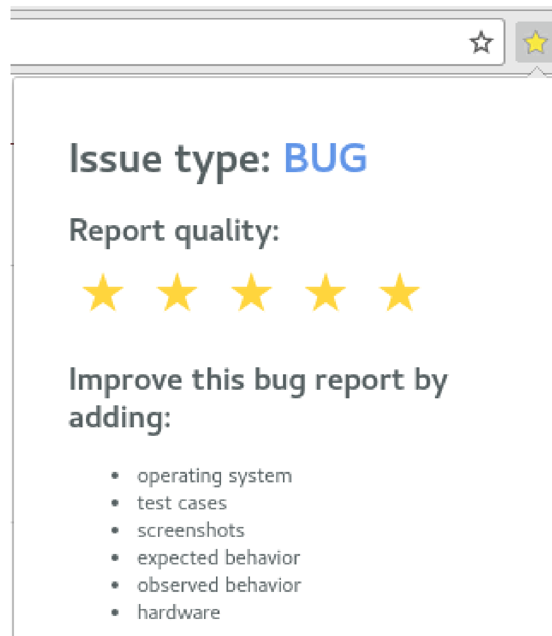
Use the quality estimation to calculate a score for a bug report:

```
QualityEstimator estimator = new QualityEstimator();

// calculate the score
int score = estimator.score(issue);

// get a map with the features as keys and a value
// of either 1 (if feature is present) or 0 (otherwise)
Map<String, Integer> featureMap =
      estimator.activationFeatures(vector);
```

7.3 Issue Analysis browser extension



The Issue Analysis browser extension is a tool to assess issue reports. It distinguishes bugs and non-bugs and can provide a quality rating as well as improvement suggestions for bug reports to the user. After the installation, a star icon appears next to search bar. A click on the icon shows a popup with the assessment results. The code repository can be found here:
`https://github.com/curtys/issue-vectorization`

The extension is compatible with Firefox and Chrome and requires a server running the Issue Analysis module:

- Tomcat webapp:
  `https://github.com/curtys/issue-analysis-service`
- Embedded Jetty server:
  `https://github.com/curtys/issue-analysis-service-embedded`

It may be necessary to configure the service URL. This can be done on the settings page of the add-ons:

- In Firefox: Menu - Add-ons - Issue Analysis - Preferences
- In Chrome: Menu - More tools - Extensions - Issue Analysis - Options

Bibliography

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guhneuc. Is it a bug or an enhancement? A text-based approach to classify change requests. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pages 304–318. ACM, 2008.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann. Quality of bug reports in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, pages 21–25, New York, NY, USA, 2007. ACM.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer Science+Business Media, LLC, 2006.

[4] S. J. Dommati, R Agrawal, R. M. R. Guddeti, and S. S. Kamath. Bug classification: Feature extraction and comparison of event model using naive Bayes approach. *CoRR*, abs/1304.1677, 2012.

[5] Simon Haykin. *Neural Networks and Learning Machines*. Pearson Education, Inc., 2009.

[6] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 international conference on software engineering*, pages 392–401. IEEE, 2013.

[7] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 1–10, May 2010.

[8] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *2008 IEEE International Conference on Software Maintenance*, pages 346–355, Sept 2008.

[9] M. N. Pushpalatha and M. Mrunalini. Predicting the severity of bug reports using classification algorithms. In *2016 International Conference on Circuits, Controls, Communications and Computing (I4C)*, pages 1–4, Oct 2016.

[10] P. Schugerl, J. Rilling, and P. Charland. Mining bug repositories–a quality assessment. In *2008 International Conference on Computational Intelligence for Modelling Control Automation*, pages 1105–1110, Dec 2008.

[11] S. J. Sohrawardi, I. Azam, and Hosain S. A comparative study of text classification algorithms on user submitted bug reports. In *Proceedings of the 2014 Ninth International Conference on Digital Information Management (ICDIM)*, pages 242–247. IEEE, 2014.

[12] C. Z. Yang, K. Y. Chen, W. C. Kao, and C. C. Yang. Improving severity prediction on software bug reports using quality indicators. In *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pages 216–219, June 2014.

[13] C. Z. Yang, C. C. Hou, W. C. Kao, and I. X. Chen. An empirical study on improving severity prediction of defect reports using feature selection. In *2012 19th Asia-Pacific Software Engineering Conference*, volume 1, pages 240–249, Dec 2012.

[14] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schrter, and Weiss C. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36:618–643, 2010.