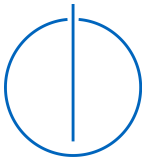
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Support Scrub Meetings in Distributed Teams by Detecting Duplicates of Software Defect Reports in Issue Management Systems

Maximilian Flis



DEPARTMENT OF INFORMATICS

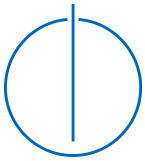
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Support Scrub Meetings in Distributed Teams by Detecting Duplicates of Software Defect Reports in Issue Management Systems

Unterstützung von Scrubmeetings Verteilter Teams durch Erkennung von Doppelten Softwarefehlerberichten in Fehlerverwaltungssystemen

Author:	Maximilian Flis
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Christof Tinnes
Submission Date:	May 15th, 2020



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, May 15th, 2020 Maximilian Flis

Acknowledgments

I would first like to thank my colleagues at Sky for making this work possible and for aiding me with their expertise in defect management. Their continuous support during this work was greatly appreciated. Especially, I want to thank my advisors Bertwin Wolf (Sky) and Christof Tinnes (TUM). I am grateful for the weekly syncs with Christof throughout this work where he provided me with guidance and many insightful comments and feedback. He always showed me a way to proceed when I encountered obstacles or struggled with certain topics. Additionally, I want to express my gratitude to my supervisor Prof. Dr. Florian Matthes. Furthermore, I want to thank all my friends and family for their tremendous support while I was focused on the research for this work. And finally, a huge shout out to Markus Hoppe for his awesome brewing skills resulting in the best beer in the world.

Abstract

Developing software is a complex task that is prone to errors which cause faulty behaviour of a piece of software. In technical terms, these errors are referred to as software defects or bugs and need to be tracked to reflect the software's state of quality.

It is state of the art to use issue tracking systems like JIRA to submit and track reports based on bugs. At Sky who is our industry partner, JIRA is used to track bugs. Defect reports have to be reviewed before faulty software can be fixed. The process of reviewing and assigning defect reports to developers is generally called defect triage or defect *scrub* at Sky and happens usually multiple times a week. Nonetheless, no system is perfect. Tools like JIRA do not prevent defect reports focusing on the same problem only differing in the words used to describe the defect. Duplicate defect reports waste valuable time of *reporters*, *triagers* and *fixers* which could be invested in tasks like fixing high priority bugs, for instance.

With *Defejavu*, we introduce a recommender system leveraging word embeddings and **NLD**, a novel way of expressing document vector similarities, paired with **RankNet**, a learning-to-rank algorithm aiding to rank similarity vectors obtained from textual, categorical and time-based features of bug reports (BRs), to retrieve a list of top-k similar BRs for a query BR q.

Additionally, we show that expert triagers in closed source environments maintain a throughput-optimized defect management process where similar BR recommenders add the most value at the time of BR creation.

Contents

1 Introduction

1.1 Context and Motivation

In bug management or more specifically bug triage which is a part of a software development life cycle (SDLC) there are usually two types of duplicate bug reports (BRs) as Sun et al. [1, p. 46] stated where "[o]ne describes the same failure [TYPE I], and the other depicts two different failures both originated from the same root cause [TYPE II]". In 2005, Anvik et al. [2, p. 38–39] mentioned, “everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle”. At Sky, there are roughly 1000 updates to tickets daily. Unfortunately, in spite of advances in terms of automating bug triage, there is still room for improvement with respect to bug management. As Anvik et al. [2, p. 35] further stated, ”humans must read the bugs and decide upon whether they are duplicates, and to whom they should be assigned”. Similarly to the focus of Sun et al. [1], we target TYPE I but we do not classify query bug reports into duplicate or non-duplicate. We seek to support the triage process without introducing additional restrictions by automatically filtering duplicates or patronizing the experts with comprehensive domain knowledge. Therefore, it is crucial to understand what bug triagers are actually looking for in order to simplify their bug triaging processes.

Open bug repositories render themselves as a target for increased duplicate bug report rates [3, p. 53] as opposed to closed ones like the one at Sky with highly optimised BR management backed by an advanced bug *scrub* process and reporters with tremendous domain-knowledge. Nonetheless, memorizing details of large amounts of BRs is far from justifiable. Therefore, duplicate BRs are reported far more frequently than desired. Here is where we see potential for improvement.

1.2 Contributions

Because of the identified potential for improving bug triage, there are mainly two goals we want to contribute to:

1. In the past, researches applied common information retrieval (IR) techniques like *TF-IDF* and similarity measures like cosine similarity [4]. In chapter 3, we refer to a non-exclusive set of these contributions. However, due to more recent advances in this domain, we want to experiment with word embeddings and Word2vec models and a novel similarity measure based on the idea of a non-linear space in which the document vectors are situated as opposed to the common cosine similarity in a linear space.
2. We closely work with Sky, our industry partner, to learn about defect management in closed source environments tracking bugs in closed bug repositories leveraging Atlassian Jira[1] (JIRA), an issue tracking system (ITS). Especially the impact of tool assisted duplicate detection with respect to the *scrub* process at Sky are of interest to us.

1.3 Research Questions

This section is dedicated to emphasize the research areas of this work. We compiled a total of six research questions. The motivation and methodology are described in each research question section within this chapter. All the results are covered in chapter 6.

RQ1 Do duplicate defect reports pose problems in a software development life cycle?

Motivation. This question forms the base of the research conducted and presented in this work. Since we cooperated with Sky before we started this research, we had the opportunity to participate in projects there. We learned that ITSs are very useful for a SDLC but with the state-of-the-art ITSs, best to our knowledge, there is no native functionality to prevent duplicate BRs. Furthermore, we experienced an

[1] `https://www.atlassian.com/software/jira` (verified 2020-02-04)

increased amount of duplicate BRs during phases of projects where the ITS was heavily frequented. Therefore, we wanted to learn if duplicate BRs pose problems in a SDLC.

Methodology. There are several ways to elaborate on the topic of duplicate defect detection. In the past (chapter 3), the focus was mainly on looking into data sets of open bug repositories. We believe that personal communication is key to determine problems of duplicates in bug repositories. Hence, we directly consulted five expert *triager*s at Sky in individual qualitative interviews to get first hand impressions of daily reoccurring bug triage. The group of experts consisted of three System Integration Engineers (SIEs), one Scrum Master (SM) and one Defect Manager (DM) who can be considered as the interface between raised bug reports and the triage team. See Table 1.1 for more information about the experts.

Table 1.1: Group of expert triagers and their experience.

Expert	Role	Defect Management (years)	JIRA (years)
SIE1	System Integration Engineer	19	6
SIE2	System Integration Engineer	10	10
SIE3	System Integration Engineer	8	5
SM	Scrum Master	4.5	4.5
DM	Defect Manager	20	8.5

Insights into the problems of duplicate defect reports were obtained by discussing the topic *Duplicate Reports* in the interviews. Additionally, we reviewed previous work to reason about RQ1.

RQ2 What are the reasons for not incorporating tools aiming at detecting duplicate reports?

Motivation. In chapter 3, previous work regarding bug triage supported by tool-based approaches is outlined. However, no automated way of similar issue detection is implemented at Sky. We try to find the reasons for the lack of automation supporting the bug triage tasks.

Methodology. Similarly to the methodology of obtaining results for RQ1, we dedicated questions within our qualitative interviews to find an answer for this research question. The questions were chosen to obtain sentiments about these topics:

- *Lack of Native Similar Issue Detection* (JIRA)
- *Preventive Measures against Duplicate Reports*
- *Tool Assisted Similar Issue Detection*

RQ3 Does tool assisted detection help in highlighting duplicates before the *scrub* meeting?

Motivation. One of the tasks in the defect management workflow at Sky (section 2.2) is preparation of the triage meeting. Our goal is to support this heavily on human memory relying task by automating the detection of similar issues.

Methodology. To help better understanding the process of the *scrub* meeting at Sky, we attended 15 sessions over the course of 2 months and tracked the discussion time of each ticket in the list of pending tickets for each day (section 2.2). Furthermore, some questions of the interview were dedicated to answer this research question. Since this research question is targeting the preparation of the triage meeting, the DM's answers are considered to be most valuable (see Table 1.1 and RQ1 for more methodology details). These questions covered the following topics:

- *Triage Preparation*
- *Triaging Challenges*
- *In-depth Issue Discussion*
- *Linking Policy*
- *Tool Assisted Similar Issue Detection*

Regarding *In-depth Issue Discussion* one can imagine the triage meeting at Sky to move very quickly. Hence, in this meeting, defects usually are assigned to an appropriate development team without a profound analysis. From time to time, *triager*s take a closer look at the discussed ticket. For this reason, it seemed interesting to ask the *scrub* participants about the causes for the longer discussion especially when considering the

relation between profound analysis of the BR and it being resolved as duplicate at the end of the discussion.

Additionally to the questions asked in the interview, our approach (chapter 4) was evaluated shortly. However, due to a misconfiguration of our tool causing too much noise it had to be disabled (section 6.2).

RQ4 Does tool assisted detection affect finding duplicate defect reports during the *scrub* meeting?

Motivation. Apart from the preparation task, new defects have to be triaged, i.e., ideally assigned to the correct development team which is in charge for further analysis and eventually delivering a fix.

Triaging is a hard task. There are several challenges that can cause trouble to the triage team like poor quality of the BR's description, duplication or incorrect assignment.

We aim to find similar issues with our approach to help reduce duplication not only while preparing the triage meeting but also during the meeting itself.

Methodology. To answer this research question, we use the same methods as for finding the answer to RQ3. However, this time the answers from the experts other than the DM are considered to be more valuable (see Table 1.1 and RQ1 for more methodology details).

RQ5 How generalizable is the proposed duplicate detection tool to support bug triage processes in other domains?

Motivation. The goal of this research question is to assess the tool's extensive operational capability. We do not aim for creating a perfect fit just for one specific domain at one specific business. Bug triage is a task performed within closed source and open source projects. Considering previous work (chapter 3), there has to be high demand for automated workflows to cope with the high amount of BRs filed daily.

Methodology. Because of the importance of a general approach to the problem of duplicate bug report detection, we allocated more time in the interviews to get an impression of the inner workings of a closed bug repository triage process and compared it with processes from open bug repositories. This portion of the interviews covered the following categories:

- *Triage Process Description*
- *Triage Process Classification*
- *Perks of the Triage Process*
- *Triaging Improvements*
- *Rapid Topic Understanding*
- *Tool Assisted Similar Issue Detection*

RQ6 Are there any measures in place to cope with a total loss of domain knowledge of the defect *scrub* team?

Motivation. While participating in the *scrub* meetings, we observed rather short discussion durations for each pending BR (see section 6.1). Based on our assumptions, this is due to broad domain knowledge provided by the *scrub* participants. We wanted to learn about the measures in place to avoid a reduced throughput in or even complete halt of the *scrub* meeting.

Additionally, while conducting the research for this work, we stumbled upon the MicroPython[2] project on Github where domain knowledge is densely bundled because of a single person contributing the most changes as it seems.

Methodology. With this in mind, we forwarded this research question directly to the experts in our interviews.

[2] `https://github.com/micropython/micropython` (verified 2020-02-21)

2 Software Development Process

This chapter will provide a short overview of the SDLC (ISO/IEC/IEEE 12207:2017 [5]) and models describing its concepts in a software development process. Afterwards, an insight into defect management at Sky will be given to highlight the challenging tasks of BR assignment by defect *triager*s in software projects.

2.1 Overview

Software Development Life Cycle

According to MacKay [6] and *SDLC - Overview* [7], there are six stages in the SDLC. We consolidated deployment and maintenance into one stage.

1. Planning and Requirement Analysis
2. Defining Requirements
3. Designing the Product Architecture
4. Building or Developing the Product
5. Testing the Product
6. Deployment in the Market and Maintenance

Software development is not just a matter of writing several lines of code to obtain a working product. A software project should be planned appropriately to avoid surprising results or even failure. It involves a sophisticated process which is split up into the aforementioned stages. Each stage is associated with planned tasks producing a result that will be used in another stage, i.e., writing code is done in the software development stage which is then verified in the testing stage. Without the code, testing cannot be completed.

Over the past couple decades major models have evolved from the SDLC organizing its stages in different orders including all sequential and even parallel arrangements of distinct stages. Common models used in the industry are the waterfall model and the agile model.

Software Development Life Cycle Models

Waterfall Model

As Ruparelia [8] claimed, the foundation of all SDLC models is the waterfall model. Initially presented by Benington [9] and revised by Royce [10], the waterfall model arranges the SDLC stages in this sequential order [8, p. 9]:

1. Evaluation
2. Requirements
3. Analysis
4. Design
5. Development
6. Validation
7. Deployment

This model stands out through its simplicity because of the sequential order of the SDLC stages. Each stage has to be completed before the next stage can commence. For specific stages, certainly, this is reasonable.

To seize the idea of the previous example, one cannot start testing software without development which seems to be meaningful at first glance, but can be considered to be too simple for software projects. Thinking about it, reveals the possibility of starting to test certain features of a software product before everything is in place.

Additionally, requirements are bound to change more often than the waterfall model allows to [11], i.e., after the requirements stage, they are fixed and cannot be changed anymore. Because of this, the model is often criticized of being too strict and too simple for the purpose of software development [12, 13].

Agile Model

Due to the limitations of the waterfall model, software development practitioners aimed for more flexible models or even get rid of a life cycle altogether [12]. The agile model is well suited for frequently changing requirements [14] considering the nature of the arrangement of the SDLC stages [15].

At the beginning, i.e., the initial exploration, the initial set of requirements needs to be identified [16]. When this is in place, the actual agile SDLC can begin. It is a concatenation of iterations, each of which consisting of most of the SDLC stages typically lasting for two weeks [16]. Each iteration is kicked off by the planning stage. Requirements should be fixed for the upcoming iteration at this point in time, however, while this iteration is in progress, the requirements for the following iteration are gathered, refined and fixed so that the next iteration can progress without impediments [16].

After the planning is complete, the developers start to implement the planned scope of work for the given iteration. In general, the iteration's scope is divided into small, testable and usually independent packages that are verified before being marked as complete, i.e., the development stage and testing stage are tightly coupled and also happen in parallel considering the independent work packages [16].

At the end of each iteration, the resulting piece of software extended by the iteration's increment should be deployable, i.e., the software is working without flaws detected while testing [16, 17]. The advantage of this approach is a fast and continuous delivery which in turn provides the possibility of quick feedback from the targetted user base of the software [16, 18].

The agile model allows for stakeholders to control the project's direction to get the best business value out of the delivered software while minimizing cost [16].

V-Model

The V-Model is similar to the waterfall model due to its sequential nature, however, verification and validation are heavily focused in this SDLC. Its name is derived from the arrangement of the stages (see *SDLC - V-Model* [19]). Each stage is contrasted by a corresponding verification and validation stage. Only if the validation of the current stage was successful, transitioning to the next stage is granted. The order is

from abstract (requirements analysis) to concrete (development) tasks regarding the implementation of the software [8, 18, 19].

The benefit of this model compared to the waterfall model lies in the early testing stages instead of relying on testing at the very end of the waterfall. However, this is also its pitfall if time is a constraint. Reviewing each outcome of every single stage is very time consuming. Therefore, the V-Model is not very flexible considering the nature of the strict verification and validation stages. If changes occur, not only the requirements documentation but also the testing documentation have to be updated [18].

Defect Management

All of the SDLC models mentioned before have one of the key tasks in common: defect management, also known as bug management which is part of the maintenance stage. Unfortunately, software cannot be developed without faults because software is developed by humans and as stated very long ago, "errare humanum est" [20]. As long as this is the case, faults will be part of any software created by human kind. Since it is impossible to prevent defects in software, there has to be a way to keep track of these faults. In the past, several bug or more general ITSs have emerged to facilitate recording the details of the faulty behaviour in a software product. Such a record is called bug report (BR) or defect report (DR). The terms defect and bug will be used interchangeably in the following chapters.

Nowadays, one of the popular ITSs is JIRA. It provides several useful features such as quick access to issue reporting and filtering along with customizability simplifying the tracking of a BR's life cycle.

However, a tool only is not enough for effective software maintenance. It requires an efficient workflow managing the reported defects to be able to cope with the amount of new defects raised daily, especially in larger software projects. Therefore, it is crucial to optimize the bug workflow as far as possible.

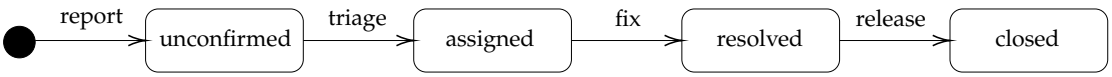


Figure 2.1: A simplified life cycle of a BR in case there is no reason for rejecting the BR.

In general, bug management consists of four distinct steps (see Figure 2.1):

1. Report

2. Triage
3. Fix
4. Release

According to Ohira et al. [21], the bug management steps can mapped to four bug management patterns in open source projects with respect to the roles *reporter*, *triager* and *fixer*:

1. $reporter = triager = fixer$
2. $reporter = triager \neq fixer$
3. $reporter \neq triager = fixer$
4. $reporter \neq triager \neq fixer$

Depending on the project structure and the available resources one of the patterns above is used to ensure a fault in software is handled adequately so that the quality of the product is maintained or even increased.

2.2 Defect Management at Sky

Considering the bug management patterns described above, Sky best fits into pattern #4 even though BRs are tracked in a closed bug repository and the software is closed source as well. The roles are split across human resources at Sky where *reporters*, *triagers* and *fixers* are generally represented by different persons.

The overall life cycle of a BR is very close to the one outlined in Figure 2.1. However, at the beginning, it is split into two different branches where it is checked if the BR targets a released or unreleased software version, i.e., whether end users are faced with the problem or not. In this work, we base our research on the branch handling BRs targeting unreleased software.

After this initial check is completed, a BR traverses all stages as seen in Figure 2.1. Additionally, there are situations where a BR can be rejected, i.e., because it is a duplicate, or redirected to a previous stage, for instance, if the BR is not yet ready for triage because of missing or inconsistent information.

Defect triage is performed in a so called *scrub* meeting which is scheduled daily as a remote call to cope with the high amount of BRs raised daily and to ensure BRs are progressed accordingly. In this short lived meeting, the participants discuss BRs which are selected by the DM beforehand. He ensures that BRs meet a certain level of quality to be able to perform analysis within the meeting and he also acts as a moderator of the meeting who announces each BR at the beginning of its discussion.

When an issue has been discussed, it is not directly assigned to a developer. Sky employs teams of developers who consume issues assigned by a respective component matching the team instead of using the field *assignee*. Similarly, Anvik et al. [2] mention the generic use of the field *assignee* which does not always point to a single developer.

In the case of uncertainty within the *scrub* team, the issue is put on hold to retrieve more details about the given problem from the reporter. When the requested information is provided, the ticket will be discussed in a future *scrub* meeting.

Reporters of bugs are advised to search for existing BRs before raising a new one but this measure cannot entirely prevent the creation of duplicate BRs. If a duplicate is identified, the BR is resolved as such and linked to the original BR. Contrary to Hiew [22], at Sky a bug report can be marked as duplicate even though chronologically it should be the original one. During the review of potential BRs for the *scrub* meeting and if the DM is certain that a given BR is a duplicate, it will be resolved as such as well and will not be included in the pool of BRs waiting for discussion in the *scrub* meeting.

BRs that get assigned to a development team follow the flow from Figure 2.1 if the team accepted the BR, i.e., confirmed the assignment of the triagers, or reappear in the *scrub* because the wrong team was assigned.

3 Related Work

This chapter will provide a non-exclusive overview of existing solutions for tackling the problem with duplicate defect reports in issue management systems.
First, we will take a closer look at deduplication tools presented by the scientific sector.

Second, two plugins [23, p. 9–10] for the software development tool JIRA will be presented.

Last, we will briefly present an insight into related tools which do not particularly aim for resolving the problem of duplicate defect reports.

Since the inception of JIRA in 2002 it merely took a year until users of the tool requested native support for detecting similar issues. For this reason, Dawson [24] created a feature request for JIRA in 2003. In 2015, more than ten years later, the authors of JIRA updated the issue with the information to not include the feature on their roadmap due to available plugins in the JIRA Marketplace [23, p. 11] two of which we will briefly present later but first, we will examine some of the solutions provided by previous research.

3.1 Similar Bug Report Detection Systems by the Scientific Sector

Coping with an open bug repository (2005)

Anvik et al. [2] expanded on the initial investigations into automatic bug triaging by Murphy and Cubranic [25]. Their research focused on two main tasks of bug triaging in two open bug repository projects, namely Eclipse and Firefox: ticket assignment and duplicate detection. They applied machine learning techniques to train a Support Vector Machine [26] classifier on historical data of defect reports by creating a model

that is incrementally expanded over time as new defects arrive to automatically identify the appropriate developer capable of fixing the issue.

For automatic duplicate detection, Anvik et al. [2] built a statistical model using machine learning methods based on existing bug reports again with incremental updates on arrival of new reports. To classify these, cosine similarity [4] was applied. When a duplicate BR is identified, the top three most similar existing BRs are associated with this BR. They recommend first detecting duplicates before automatically providing names of suitable developers to fix the bug.

Assisted detection of duplicate bug reports (2006)

Hiew [22] examined detection of duplicate bug reports on Firefox's open bug repository. By combining similar defect reports into respective groups, they were able to provide top-n recommendations of potentially similar bug reports. These groups represent a model of reports that associates similar ones into so called centroids [27]. Similarly to Anvik et al. [2], the model is improved incrementally.

To create an increment, they first create a document vector [4] representation of each bug report. The document vector is built by textual features from the report. These features are preprocessed by stemming each word [28] and removing stop words [29] including domain specific phrases like "steps to reproduce" [22] followed by conversion into a document vector using *TF-IDF* [4].

Second, they compare the new document vector to each existing centroid represented as *TF-IDF* vector using normalized cosine similarity [4] and if a certain threshold is exceeded, the bug report is identified as duplicate including top-n recommendations with one report from each matching centroid.

Last, the new report is either optionally added to an existing centroid (original bug id exists in any of the centroids) or represented in a new one (original bug id was not found in any of the centroids).

Automated duplicate detection for bug tracking systems (2008)

Jalbert and Weimer [3] claimed manually identifying duplicate bug reports is time-consuming. Therefore, they have proposed an approach for automated duplicate detection backed by a classifier trained on reports from the Mozilla project that relies

on surface features [30], textual semantics [31] and graph clustering [32]. Their main focus is on detecting unknown duplicates as opposed to Hiew [22] where grouping similar reports is of primary importance.

From a high level perspective, their predictions are the result of a linear regression classifier where a threshold marks the boundary between duplicate and non-duplicate bug report. Jalbert and Weimer [3] do not opt for an incrementally updated model. Instead, they suggest periodical updates claiming that with more bug reports historical data becomes progressively irrelevant.

However, preprocessing is similar to the approach of Hiew [22] using the Porter stemming algorithm [28] and removing stopwords. Interestingly, one of their claims [3, p. 57] criticises the usage of inverse document frequency [4] for detecting duplicates, instead, they obtain their document vector by applying a custom weighting function [3, p. 56]. For calculating the distance between two document vectors, again cosine similarity is used.

As far as graph clustering is concerned, a social network clustering algorithm introduced by Mishra et al. [32] is applied to circumvent the problem of unknown amount of clusters [3, p. 56] with the benefit of highlighting the cluster with most neighbours within itself. Textual features and clustering results act as input for their linear regression model.

Finally, they seem to be very confident about their approach because of filtering out all reports identified as duplicate but one from the triage process [3, p. 54].

A discriminative model approach for accurate duplicate bug report retrieval (2010)

Once again the choice of open bug repositories fell to Firefox and Eclipse. Additionally, Sun et al. [1] opted for including OpenOffice into the evaluation. Instead of filtering duplicate reports as Jalbert and Weimer [3] did, Sun et al. [1] provide a list of top-*k* most similar reports (sorted from highest to lowest relevance) due to their claim "one report usually does not carry enough information for developers to dig into the reported defect" [1] backed by Bettenburg et al. [33]. Their approach relies on a discriminative model which is able to express the relevance as a probability between two bug reports.

In general, bug reports are preprocessed the same way others [3, 22] performed preprocessing in the past. Sun et al. [1] do not explain this stage more thoroughly, i.e., they did not mention tools or algorithms used.

In contrast to the previous approaches [2, 3, 22], textual similarity is expressed by the sum of inverse document frequencies of two bags of words, one from the query report and one from the other report, and two bags of bigrams. In total, they extracted 54 similarity features as a base for their model.

For building their discriminative model, they trained a Support Vector Machine [26] classifier. The dataset is split into buckets consisting of a master bug report and all its duplicate reports. A master bug report is defined as the oldest report in a master-duplicates bucket. Positive samples, i.e., a pair of a master and one of its duplicates, together with negative samples, i.e. a pair of unique reports, are fed into the training algorithm to fit the model.

Towards more accurate retrieval of duplicate bug reports (2011)

Sun et al. [34] focused on improving the accuracy of existing tools including their own approach based on a discriminative model [1].

Their main focus lied in extending the BM25F ranking function, a modified version of Okapi BM25 [35], by supporting lengthy queries like bug reports as opposed to short queries for which BM25F originally is suited best [34]. Instead of solely relying on textual information like bug summary and description, they have considered categorical features like component and version in their contribution.

Preprocessing the textual features was performed the same way as before [1] and their similarity measure was retrieved by their retrieval function REP using their BM25F extension [34].

In order to optimize their similarity function, Sun et al. [34] relied on the work of Taylor et al. [36] who introduced RNC, a simplified RankNet [37] cost function.

A contextual approach towards more accurate duplicate bug report detection (2013)

Alipour et al. [38] have adopted the approach of Sun et al. [34] where the original bug report (master) and all its duplicates are grouped into buckets.

In general, their approach matches the previous work: Preprocess the new bug report, extract similarity measures and finally retrieve a top-*k* list of most similar defect reports to the query report. However, they combine a set of metrics retrieved from textual, categorical and contextual features, for instance the non functional requirements context, by obtaining the cosine similarity [4], the Euclidean distance and a more complex logistic regression based metric to generate the top list.

Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports (2016)

Yang et al. [39] use word embeddings to further improve the *recall rate@k* (see section 6.1) of similar bug report detection tools (SBRDTs).

They resort to Gensim Python library[1] (GENSIM), a python framework combining several functions for leveraging information retrieval tasks. However, because of claiming that traditional information retrieval techniques complement word embeddings, they used not only word embedding vectors but also TF-IDF document vectors to compute and combine the resulting similarity scores [39, p. 129].

In addition to the bug reports, Yang et al. [39] incorporate commit messages linked to individual reports by bug id from source code control stored at Github to reason about similarity in terms of mutual source code files and to build the ground truth for the similar report detection task.

The final similarity score is a combination of three similarity metrics retrieved from comparing the query report and all other reports. First, they calculate the cosine similarity of the TF-IDF vectors, second they retrieve the word embedding vector similarity by calculating the cosine similarity and the last score consists of the similarity of categorical features such as *product* and *component*.

[1] `https://radimrehurek.com/gensim/` (verified 2020-03-10)

3.2 Commercial Solutions

Find Duplicates

Find Duplicates is a plugin for JIRA. It claims to find duplicate BRs while a BR is being created and existing BRs are associated with a list of potential duplicates.

From our personal communication with the developers of Find Duplicates, we learned that it relies on Apache Lucene[2] (LUCENE) which provides indexing and search features.

Unfortunately, no more details about the techniques used were shared with us.

Similar Issues Finder

Similar Issues Finder is another plugin for JIRA and provides a similar set of features as Find Duplicates.

According to our personal communication with the developers, it is based on JIRA Query Language[3] (JQL) similarity search leveraged by LUCENE which is also used in stemming the user queries. JQL provides the "~" operator for searching occurrences of free text. Therefore, they create queries similar to "BR summary ~ *keyword*" and return the top result as similar BRs. In addition, they apply some unspecified techniques to improve the relevance of the result. Depending on the configuration of the plugin, other BR features are incorporated to retrieve the list of similar BRs.

[2] `https://lucene.apache.org/` (verified 2020-04-05)

[3] `https://www.atlassian.com/blog/jira-software/jql-the-most-flexible-way-to-search-jira-14` (verified 2020-04-02)

4 Methodology

In this chapter, we describe our methodology to retrieve a list of top-k recommendations of most similar BRs for a given query BR q. Figure 4.1 shows an overview of our workflow for finding such a list. This workflow is split into two phases, *ramp-up* and *running*.

In the *ramp-up* phase (indicated by the dotted arrows in Figure 4.1), there are two stages where first, existing BRs are extracted from the source data storage system in an *extract, transform, load* (ETL) process (section 4.1) and then preprocessed in a *preprocessing* (PREP) process (section 4.2) to be finally stored in a database as vectorized bug reports (VBRs). We will provide more details about VBRs including the usage of machine learning methods in section 4.2. The ETL process is required because we were not allowed to directly access the database where all JIRA tickets are stored including the BRs we were interested in.

After the *ramp-up* is completed, we move to the *running* phase. The first two stages are the same as in the *ramp-up* phase. Additionally, in the third stage we obtain predictions, i.e., potential top-k similar BRs, for each new VBR q' by querying it against existing VBRs in a *two-pass prediction* (PRED) process (section 4.3). Subsequently, in the last stage these predictions are published in a *publish* (PUB) process (section 4.4).

4.1 Extracting Defect Reports from the Issue Tracking System

As mentioned before, we had to extract the BRs from the JIRA project. Apart from the permission to work on life data, we would need to go through the process of deploying our approach to a test system first and then target the production system (JIRA). Since we are mostly interested in the defect management process at Sky, the *scrub* workflow in particular, choosing this direction would go beyond the scope of this research. Instead, we opted for extracting the BRs into our own database and work with the data from there.

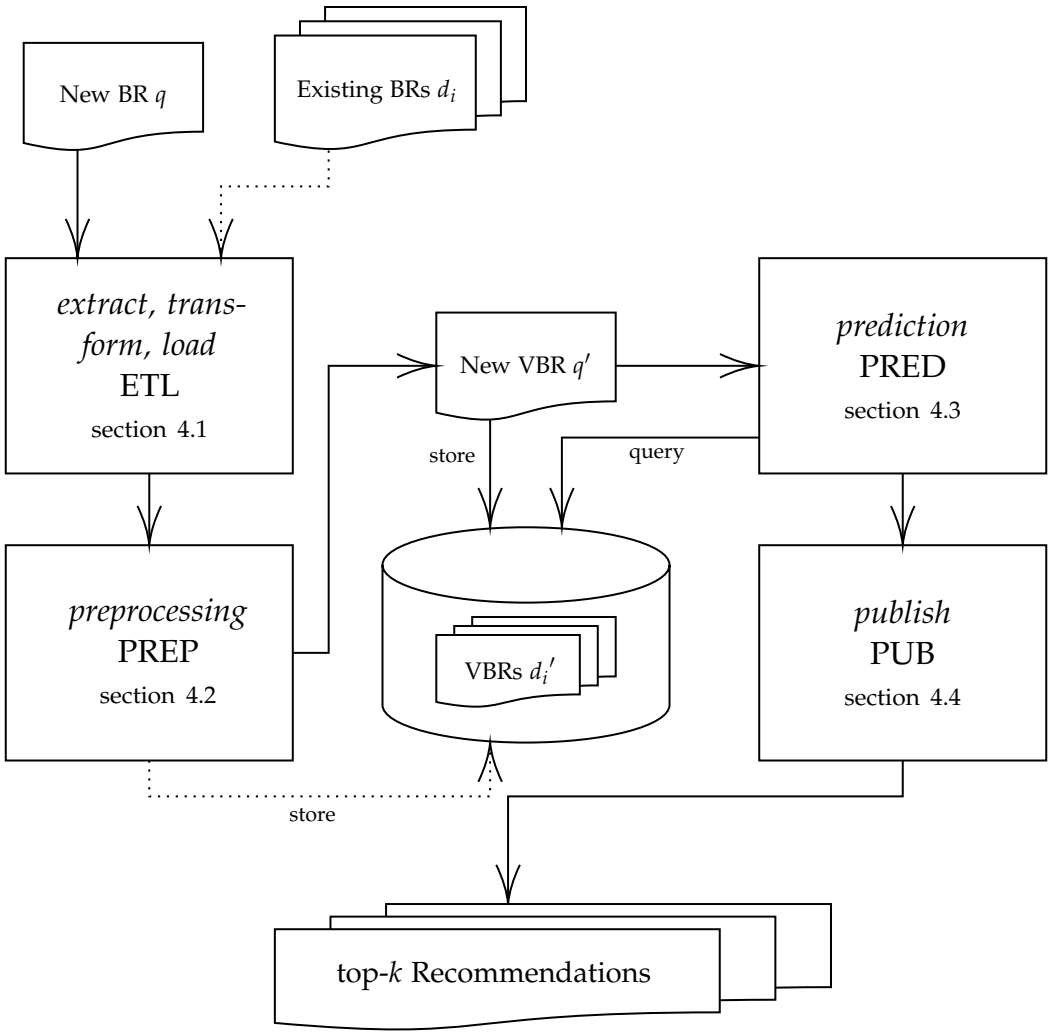


Figure 4.1: High level workflow overview of our approach.

We also targeted a system which could be deployed and used after the research for this work is complete. Therefore, keeping the extracted data in sync with the source was key in the decision making process. Our pick for the ETL process was the software platform SyncPipes[1] (SYNCPIPES) by Koch [40]. Unfortunately, at the time of creation of our ETL process SYNCPIPES did not support proper syncing from a JIRA project, i.e., it was possible to extract the data as is, but to our best knowledge incremental updates, i.e., actually syncing the data, were not implemented. Consequently, we extended

[1] `https://wwwmatthes.in.tum.de/pages/2gh0u9d1afap/SyncPipes` (verified 2020-03-08)

SYNCPIPES with this functionality for JIRA projects to enable incremental syncing and to keep the data on our end up to date. Apart from that, we stuck to the examples by Koch [40] to configure our ETL process with the help of a SYNCPIPES pipeline.

SYNCPIPES does not support scheduled pipeline runs. For that reason, we made sure that syncing is performed periodically to keep the data up to date and additionally to avoid to much stress on the infrastructure hosting the JIRA project with the data of our interest, i.e., BRs. Because of the pipeline run scheduling, updates to BRs or new BRs are fetched in batches. Accordingly, recommendations of similar BRs cannot be presented in real time.

Initially, we wanted to provide these recommendations while a new BR is being created to reduce duplication of BRs. This requires to be as close as possible to the source data, i.e., be able to directly access it. The proper way to achieve this with JIRA is to create a JIRA plugin.

However, due to the mentioned limitations, we had to resort to a ETL process with the drawback of periodic pipeline runs as well as the advantage of being more generalizable (section 6.4), because a JIRA plugin cannot run standalone.

With a very heavily frequented JIRA project like the one at Sky, updates to tickets are very common. Therefore, we selected a cool down period of five minutes between pipeline runs reducing the load on the server while still having fairly current data.

4.2 Preparing Defect Reports for Ranking

To be able to find similarities between two BRs, they have to be transformed into a numerical representation. We call this process *vectorizing*. In previous work, vectorizing involves converting the BR's description and sometimes its summary into a vector by applying the vector space model [41]. Typically, this is done by calculating the *TF-IDF*, a modification or partial appliance of it, i.e., only TF or IDF [1, 22, 34, 38, 39].

TF-IDF is a very well spread statistical measure [42] consisting of two parts targeted at indicating the importance of a word in a document. For instance, one could refer to a document as the BR's description. TF is the frequency of a word in a document d. IDF is the inverse frequency of the documents where the word appears in. *TF-IDF* is defined as follows:

$$\boldsymbol{tfidf}(t, d, D) = \boldsymbol{tf}(t, d) * \boldsymbol{idf}(t, D) \quad d \in D \tag{4.1}$$

where: t : the word of which the score should be calculated
d : the document, i.e., a BR's description
D : the set of available documents.
$\boldsymbol{tf}$: the raw count of t in d (other weights are also possible [43, p.128]).
$\boldsymbol{idf}$:

$$\boldsymbol{idf}(t, D) = \frac{|D|}{|\{d \in D : t \in d\}|} \tag{4.2}$$

Word embeddings. Since *TF-IDF* is very common among recommender systems [42], we decided to try a less frequent way of representing BR descriptions in a numerical form. We opted for *word embeddings* obtained from Word2vec [44] models because it allows for catching synonyms in text as opposed to *TF-IDF* which is a discrete representation of words [45]. Word embeddings can be seen as transforming each word in a document into an n-dimensional vector where n is defined by the vector space which is the output of a neural network model trained on a large corpus of text [39, 46]. In order to have the best context for creating word embeddings, we would have to create our own neural network model based on the BRs from the JIRA project at Sky. Since the corpus of text has to be large to retrieve reasonable vector space representations of documents, this task has proven itself to be hard, because the JIRA project does not provide the volume of words needed to produce acceptable results [45]. Therefore, we resorted to two general purpose models, one trained on the large corpus of Stack Overflow articles (Stack Overflow 200 (SO200), $n = 200$) [47] and the other one trained on Google News articles (Google News 300 (GN300), $n = 300$) [45].

However, contextual Word2vec models targeted at specific domains, for instance, broadcast media at Sky, can outperform general purpose models [48], but it requires proper optimization of hyperparameters [49] due to the low volume of the input corpus. This optimization is beyond the scope of this research and therefore, we put only the general purpose models into consideration for our approach after heaving tried to train a contextual Word2vec model.

Stemming and removal of stop words. Similarly to classic IR approaches relying on *TF-IDF*, we preprocess the BR descriptions by removing stop words and applying stemming before retrieving the word embedding for a particular word, i.e., words such as "the", "a", "an", "in" are removed and forms of words such as "crashed" or "flickering" are reduced to their infinitive form "crash" and "flicker" in order to prevent distortion in the resulting word embeddings.

Static rank features (SRFs). Instead of only focusing on the descriptions of BRs, we took several other textual, categorical and time-based features into account (see Tables 4.1, 4.2 and 4.3) which turned out to improve *recall rate@k* (see section 6.1). Additionally, in section 4.3, we will introduce dynamic rank features which are calculated on the fly for a set of two BRs before retrieving the predictions.

Table 4.1: Textual Static Rank Features.

feature	encoding	description
components.SO200 components.GN300	word embedding	A list of predefined components associated with the BR q. The list was converted into a space seperated string.
customer_impact.SO200 customer_impact.GN300		A custom textual field similar to a short summary regarding the customer impact of the BR.
description.SO200 description.GN300		A BR's description.
issuelabels.SO200 issuelabels.GN300		A list of custom labels associated with the BR q. The list was converted into a space seperated string.
summary.SO200 summary.GN300		A BR's summary.

Features with a SO200 suffix were encoded with $WEM = SO200$
Features with a GN300 suffix were encoded with $WEM = GN300$ (see Equations 4.3 and 4.4)

Tables 4.1, 4.2 and 4.3 consolidate all features providing its names, descriptions and encoding. Converting raw values of features, i.e., the priority of a BR, is referred to as encoding with respect to machine learning. A fairly common type of encoding involves *one hot* encoding [50] where raw values are converted into a list of bits with a single high bit and the rest of low bits. A *true* value indicates the existence of the category, a *false* value indicates otherwise. *Multi category* encoding is very close to one hot encoding with the difference in the amount of high bits. For each matching category a high bit is registered. For instance, if a bug was reproduced in two out of three countries (Germany and Austria), the reporter would associate the BR with these two countries. Then, during encoding of this feature a high bit for Germany and Austria respectively would be registered.

Table 4.2: Categorical Static Rank Features.

feature	encoding	description
priority	one hot	The priority of a BR. Possible values: *low*, *medium*, *high* and *very high*.
intermittence		The intermittence of a bug. Possible values: *not intermittent*, *intermittent* and *highly intermittent*.
severity		The severity of a bug. Possible values: *minor*, *major* and *showstopper*.
country	multi category	A list of territories or countries where the bug was reproduced.
platform		A list of platforms (i.e. hardware devices) where the bug was reproduced.
reporter	raw	The reporter of the BR.

The features *priority*, *intermittence* and *severity* were one hot encoded but are actually treated as ordinal features with the order being decoupled from the encoding in our approach.
The feature *reporter* was used as is while converting the BR to a VBR because this form was sufficient for creating a dynamic rank feature (see Table 4.4).

Table 4.3: DateTime Static Rank Features.

feature	encoding	description
created	raw	A BR's date of creation

Features with no encoding (*raw*) are passed through without change and end up in a VBR as is. We decided to not encode these features in this stage because we do not need them pre-encoded for later when creating dynamic rank features (section 4.3).

Before we can encode the features of a BR, all categories of all categorical features have to be queried from all existing BRs so that it is possible to retrieve the proper one hot or multi category vector (list of bits consisting of high and low bits). Apart from that we need to take care of the textual features for which we retrieve word embeddings because we are not interested in embeddings for single words only but in the vector representation of a feature, i.e., description.SO200. Inspired by Mikolov et al. [51], we retrieve a word vector for each word, sum the word vectors of the feature and divide it by the word count of the feature to get a feature vector (feature embedding):

$$v_{w,WEM} = Word2vec(WEM, w) \tag{4.3}$$

$$v_{rf_{WEM}} = \frac{1}{W} \sum_{w}^{W} v_{w,WEM} \tag{4.4}$$

where: WEM : a word embedding model, i.e., SO200
rf_{WEM} : a feature from Table 4.1
w : a word from the feature rf_{WEM} ($w \in rf_{WEM}$)
$Word2vec$: Word2vec implementation from the Gensim Python library
$v_{w,WEM}$: the word embedding for the word w and word embedding model WEM
W : the word count in feature rf
$v_{rf_{WEM}}$: the feature embedding for the feature rf_{WEM}.

Non linear distance (NLD). Most of the previous solutions transform documents into document vectors using *TF-IDF* and calculate the cosine distance (CD)[2]. In the recent years, Mikolov et al. [44] proposed a novel way of converting documents into vectors called *word embeddings* which we described earlier. Therefore, we experimented with Word2vec and a non linear distance based on a neural network to find out if it can outperform CD.

While CD is widely spread [2, 3, 22, 38, 39] and according to our results (see section 6.1) performs better than NLD for small k (top-k-Recommendations), we believe that the

[2]Cosine similarity is the more familiar term

mathematical space of distances between document vectors is not a linear space. It seems natural to think of an Euclidean linear space because it is simpler to imagine and therefore, it is reasonable to apply the CD measure to express the distances between words or documents.

However, best to our knowledge, it remains to be proven to be a linear space. A logical conclusion is to think of non linearity, specifically of a manifold which is a topological space locally akin to the Euclidean space near each point (locality property of manifolds), yet globally this rule no longer applies. A good way of imagining a practical example of this theory is by thinking of the Earth. Locally, it seems to be planar (we neglect the topography at this point), but globally it is a sphere. With this in mind, we experimented with a neural network based non linear distance measure. Our preliminary algorithm for retrieving NLD between the descriptions of two BRs involves the steps outlined in Figure 4.2.

1. From all BRs select all pairs consisting of an original BR and its duplicate BR (*ori-dup*).
2. Apply Equation 4.4 to retrieve a vectorized form of all descriptions in the set of *ori-dup* pairs (positive samples).
3. Create an equal amount of negative samples (*ori-nondup* pairs) by pairing the vectorized descriptions of the original BRs with descriptions of non-matching duplicate BRs, i.e., a pair of two descriptions can only exist in either the set of positive samples or the set of negative samples.
4. Label the positive samples with “1” and the negative samples with “0”.
5. Calculate the difference vectors of all pairs respectively by subtracting the *ori* vector from the *dup* vector in case of a positive sample and by subtracting the *ori* vector from the *nondup* vector in case of a negative sample.
6. Feed the difference vectors and the corresponding labels into a neural network with non linear activation functions (see Figure 4.3).
7. Obtain the NLD by subtracting the output of the neural network from 1 (Equation 4.5).

Figure 4.2: Algorithm for retrieving NLD between a feature of two BRs.

$$NLD(q', d', rf_{WEM}) = 1 - NLDNN(v_{rf_{WEM},q'} - v_{rf_{WEM},d'}) \tag{4.5}$$

where:
- q' : the *dup* or *nondup* VBR in the *ramp-up* phase or the new VBR in the *running* phase.
- d' : the *ori* VBR in the *ramp-up* phase or the exiting VBR in the *running* phase.
- rf_{WEM} : a feature from Table 4.1, i.e. "description.WEM" in the *ramp-up* phase or a textual feature name (matching the given WEM) in the *running* phase.
- $v_{rf_{WEM},q'}$: the feature embedding for feature rf_{WEM} of VBR q' (see Equation 4.4)
- $v_{rf_{WEM},d'}$: the feature embedding for feature rf_{WEM} of VBR d' (see Equation 4.4)
- $NLDNN$: see Figure 4.3

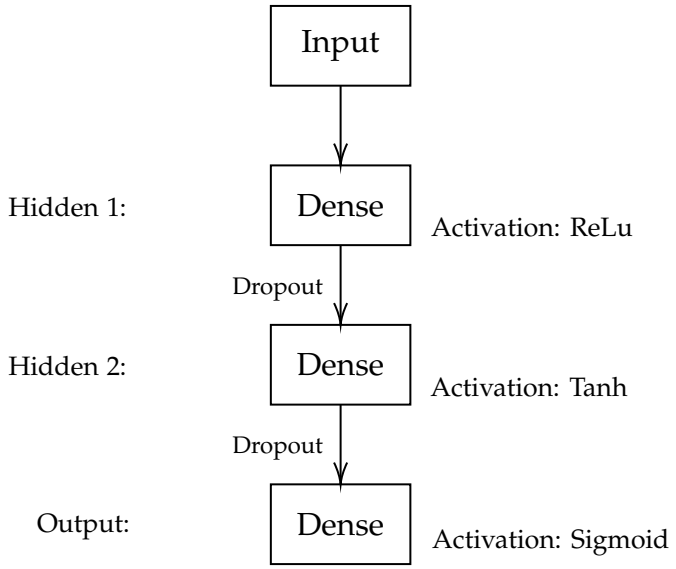


Figure 4.3: The architecture of the NLD neural network (NLDNN).

This algorithm is preliminary due to experimental results of this approach without proper evaluation. For instance, we tried to feed the absolute value of the difference vector (Equation 4.5), i.e., $|v_{rf_{WEM},q'} - v_{rf_{WEM},d'}|$, into the NLD neural network (NLDNN), but the results turned out to be worse. Another experiment could involve flipping the difference vector, but in the end it just means a flip of the result's sign. We expect a neural network to be able to figure out such a characteristic of the input on its own with a sufficient amount of positive and negative samples. To be able to move on with our research, we decided to rely just on the difference vector as input, because from our limited set of experiments it yielded the best results.

Figure 4.3 shows the neural network's architecture. It consists of an input layer, two

hidden layers and one output layer all of which are fully connected, i.e., dense. Because we believe that the distances between two text documents are located in a non linear space, we chose only non linear activation functions for the neurons within each dense layer. We expect our neural network to approximate the cost function fairly well due to the universal approximation theorem [52] which is based on the Kolmogorov–Arnold representation theorem [53, 54]. Additionally, Leshno et al. [55] have shown that simple neural networks can be understood as an universal approximator in the case of only non polynomial activation functions, especially when these networks consist of a growing number of layers limited in width according to Hornik [56].

Despite of using the descriptions of the BRs for training NLDNN, we used the same neural network for all the other textual features of a BR to calculate the NLD as long as the feature name's suffix matched the general purpose model (see Table 4.1). Since we relied on two general purpose models, we had to fit two distinct instances of NLDNN, i.e., $NLDNN_{SO200}$ and $NLDNN_{GN300}$. Training had to be performed only once, thus it is part of the *ramp-up* phase. We did not opt for updating the neural network models as time passed and leave this task for future work.

Dynamic rank features (DRFs). NLD alone returns reasonable results (see section 6.1 and Figures 6.2 and 6.4), but a BR provides far more features than the description only. Even though the description is the feature containing the most information in a BR, it was shown that combining other features with textual features can improve *recall rate@k* [3, 34, 39]. In Tables 4.1, 4.2 and 4.3, we already introduced several SRFs which can also be considered as base features for our ranking approach. When working with a single, textual feature, i.e., description.SO200, the simplest form of ranking is sorting the list of NLDs between the query BR q and all BRs d_i in ascending order. It works simply because in NLDNN (Figure 4.3) the output layer's activation function is the sigmoid function or more specifically the logistic function as shown in Equation 4.6 with a range of output values between 0 and 1.

$$\begin{aligned} f &: \mathbb{R} \to (0,1) \\ x &\mapsto \frac{1}{1+e^{-x}} \quad x \in \mathbb{R}. \end{aligned} \tag{4.6}$$

Since we labeled positive samples with 1 and negative samples with 0 (see Figure 4.2), the closer the output of NLDNN is to 0 the higher NLD will be, but we are interested in small distances, thus we have to sort the resulting list in ascending order to get top-k recommendations for similar BRs.

However, we did not rank only by the NLD of BR descriptions, but also by DRFs. We call them dynamic because contrarily to SRFs which are grouped to a VBR and stored in a database (see Figure 4.1) they are calculated dynamically at runtime when a new VBR q' arrives at the PRED stage (section 4.3). With NLD, we already introduced a DRF partially, but we need to convert it back to a similiarty measure which we do in Equation 4.7. Table 4.4 shows an overview of all DRFs used for ranking the similarity relevance between a VBR q' and the list of VBRs ${d_i}'$ retrieved during the PRED stage. The Equations 4.7 to 4.15 show how the individual DRFs are calculated to obtain an overall relevance score.

$$textsim(q', d', rf_i) = 1 - \frac{NLD(q', d', rf_i) + CD(q', d', rf_i)}{2} \tag{4.7}$$

$$catsim(q', d', P_{rf_i}) = 1 - \frac{|p_{q'} - p_{d'}|}{|P_{rf_i}|} \quad p_{q'}, p_{d'} \in P_{rf_i} \tag{4.8}$$

$$multicatsim(q', d', rf_i) = \frac{\left|M_{q',rf_i} \cap M_{d',rf_i}\right|}{\left|M_{q',rf_i} \cup M_{d',rf_i}\right|} \tag{4.9}$$

$$\Delta t = |t_{d'} - t_{q'}| \quad t \in T \tag{4.10}$$

$$cossim_{\Delta t}(\Delta t, a, b) = \begin{cases} \frac{1}{2}\cos(\frac{\pi}{b}\Delta t) + \frac{1}{2} & \text{if } a < \Delta t < b \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

$$squaresim_{\Delta t}(\Delta t, a, b) = \begin{cases} 1 - \frac{1}{b^2}\Delta t^2 & \text{if } a < \Delta t < b \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

$$sqrtsim_{\Delta t}(\Delta t, a, b) = \begin{cases} 1 - \sqrt{\frac{1}{b}\Delta t} & \text{if } a < \Delta t < b \\ 0 & \text{otherwise} \end{cases} \tag{4.13}$$

$$invcatsim(q', d', rf_i) = \begin{cases} 1 & \text{if } j_{q'} \neq j_{d'} \\ 0 & \text{otherwise} \end{cases} \quad j_{q'}, j_{d'} \in J_{rf_i} \tag{4.14}$$

$$meansim(S) = \frac{1}{|S|}\sum_{s}^{|S|} s \quad s \in S \tag{4.15}$$

where: rf_i : a dynamic rank feature (see Table 4.4).
CD : the cosine distance of the two VBRs q' and d' with respect to feature rf_i.
P_{rf_i} : a set of priority weights for the given DRF rf_i.
The set is best imagined as an ordinal encoding of categories. However,

we used one hot encoding for these categories instead of ordinal encoding (see Table 4.2). The advantage of this approach lies in the decoupling of the encoding itself from the actual order of categories where the order can be changed on demand without the requirement of re-encoding all BRs. This order is actually important for calculating the distance between a set of two categories, i.e., *high* and *very high* are closer to each other than *low* and *very high* with respect to the priority (rf_{11}) of a BR.

$p_{q'}$: the encoded category feature of the VBR q' weighted with the according priority weight from P_{rf_i}

$p_{d'}$: same as $p_{q'}$ but for the VBR d'

M_{q',rf_i} : a set of categories associated with the VBR q' for feature rf_i

M_{d',rf_i} : same as M_{q',rf_i} but for the VBR d'

T : a set of time-based features (see Table 4.3)

t : a time-based feature, i.e., *created*

$t_{q'}$: the time-based feature t associated with VBR q'

$t_{d'}$: the time-based feature t associated with VBR d'

Δt : the time delta between two dates (usually timestamps)

a : the lower bound of the interval within which Δt typically falls.
This value usually is 0 but can be any other positive number as long as it is smaller than b. The closer Δt is to a, the more similar the VBRs q' and d' are in terms of feature t.

b : the upper bound of the aforementioned interval,
b can be any positive number as long as it is greater than a. The closer Δt is to b, the more the VBRs q' and d' differ in terms of feature t,
b should be selected so that irrelevant outliers are smoothed out, i.e., in terms of duplicate detection, we believe that a set of BRs should receive a lower similarity score if their dates of creation are far apart.

J_{rf_i} : the set of categories for feature rf_i, i.e. all bug reporters (see Table 4.2).

S : the set of similarity measures (see previous equations) for which a combined mean similarity has to be calculated.

With $textsim(q', d', rf_i)$ (Equation 4.7) we define the similarity of a textual feature of two VBRs. This similarity measure consists of NLD (Equation 4.5) combined with CD. Combining these two distance measures turned out to perform better than NLD or CD on their own (see section 6.1).

Categorical similarity between two VBRs with respect to a categorical feature is expressed by $catsim(q', d', P_{rf_i})$ (Equation 4.8). Instead of matching the category exactly,

Table 4.4: Dynamic rank features

feature	short name	similarity	equation
components.SO200	rf_1	$textsim(q', d', rf_1)$	4.7
components.GN300	rf_2	$textsim(q', d', rf_2)$	"
customer_impact.SO200	rf_3	$textsim(q', d', rf_3)$	"
customer_impact.GN300	rf_4	$textsim(q', d', rf_4)$	"
description.SO200	rf_5	$textsim(q', d', rf_5)$	"
description.GN300	rf_6	$textsim(q', d', rf_6)$	"
issuelabels.SO200	rf_7	$textsim(q', d', rf_7)$	"
issuelabels.GN300	rf_8	$textsim(q', d', rf_8)$	"
summary.SO200	rf_9	$textsim(q', d', rf_9)$	"
summary.GN300	rf_{10}	$textsim(q', d', rf_{10})$	"
priority	rf_{11}	$catsim(q', d', P_{rf_{11}})$	4.8
intermittence	rf_{12}	$catsim(q', d', P_{rf_{12}})$	"
severity	rf_{16}	$catsim(q', d', P_{rf_{16}})$	"
country	rf_{13}	$multicatsim(q', d', rf_{13})$	4.9
platform	rf_{14}	$multicatsim(q', d', rf_{14})$	"
reporter	rf_{15}	$invcatsim(q', d', rf_{15})$	4.14
diff.created.cosine	rf_{17}	$cossim_{\Delta t}(\Delta t, a, b)$	4.11
diff.created.squared	rf_{18}	$squaresim_{\Delta t}(\Delta t, a, b)$	4.12
diff.created.sqrt	rf_{19}	$sqrtsim_{\Delta t}(\Delta t, a, b)$	4.13
description.all	rf_{20}	$meansim(\{rf_5, rf_6\})$	4.15
summary.all	rf_{21}	$meansim(\{rf_9, rf_{10}\})$	"
customer_impact.all	rf_{22}	$meansim(\{rf_3, rf_4\})$	"
issuelabels.all	rf_{23}	$meansim(\{rf_7, rf_8\})$	"
components.all	rf_{24}	$meansim(\{rf_1, rf_2\})$	"

The equations mentioned in the "equation" column define how each of the DRFs is calculated.

we focus on the relation between categories of a particular feature. This approach allows for a more granular representation of categorical similarity as opposed to binary values where exact matches evaluate to similar (= 1) or non-matches to not similar (= 0).

Features that allow for associations to multiple categories are encoded as DRF with $multicatsim(q', d', rf_i)$ (Equation 4.9). We make use of the Jaccard Index [57] for representing the similarity of multi categorical features.

Equations 4.10, 4.11, 4.12 and 4.13 define the similarity calculation of time-based features. We took only a single time-based feature from BRs into account, i.e., the date when a BR was filed. Based on our assumption, two BRs are more similar the closer the date of their creation is. It is also an important DRF because it expresses the relation between time and similarity. Although it is possible that BRs which are years apart are very similar or even duplicates, BRs which are closer to each other tend to have more in common. This is especially true when considering the version of a software because current BRs usually target the same or close versions. Instead of constraining the similarity detection by version, we opted for date of creation which is in indirect relation to the software version. Since we believe that this feature is important, we introduced three time-based similarity measures: $cossim_{\Delta t}$, $squaresim_{\Delta t}$ and $sqrtsim_{\Delta t}$. Figure 4.4 shows the plots of these time-based similarity measures with

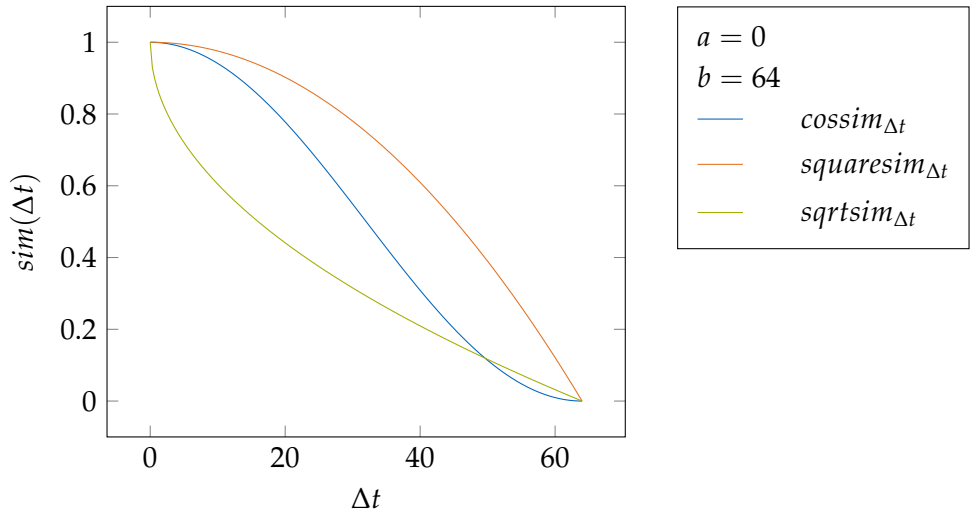


Figure 4.4: Example of time delta similarity measures with $a = 0$ and $b = 64$.

a and b adjusted to the dateset retrieved from the JIRA project at Sky. With $b = 64$, for

a pair of BRs, we considered the features rf_{17}, rf_{18} and rf_{19} (see Table 4.4) not to be similar ($sim(\Delta t) = 0$, see Equations 4.11, 4.12 and 4.13) if the creation dates of the BRs in that pair were more than 64 days apart. Looking at the plots in Figure 4.4, we have two similarity measures with $cossim_{\Delta t}$ and $squaresim_{\Delta t}$ which are not very sensitive to small Δt. This allows for BRs which are a few days apart still being classified as rather similar with a steeper decline, i.e., greater distance, the closer the difference in days of the creation dates gets to the upper bound b. The third time-based similarity measure $sqrtsim_{\Delta t}$ penalizes each day added to the difference in days for small Δt. Due to the nature of square root functions, we get values from this similarity measure that favour very small Δt which should help detecting duplicate BRs that were reported on the same day.

We spent quite some time on how the similarity of a pair of BRs in terms of their respective reporters could be expressed. After looking at the dataset regarding duplicate BR rates of reporters, we identified some that are more prone to duplicating a BR than others. While using the duplicate rate could be viable with a fixed set of reporters, it would not represent the real world. New reporters can appear every single day due to the nature of modern ways of acquiring resources, i.e., contractors who only work for a company for a limited period of time. This requires keeping the set of reporters and their duplicate rates up to date, especially for the new ones where the duplicate rate will fluctuate heavily in the beginning before settling around a certain value (principle of mathematical expectation [58]). However, with $invcatsim(q', d', rf_i)$ we resorted to a far more simple approach than incorporating the individual duplicate rate as weight leaving the use of more complex encodings for future work. This similarity measure basically represents a pair of BRs to be close to each other if the reporters of these BRs are distinct and vice versa.

Finally, we introduce $meansim(S)$. This similarity measure is another simple one where previously calculated similarity values are grouped into a set of similarities to retrieve the mean of the the provided set. By considering Table 4.4, it is apparent that we applied $meansim(S)$ only for textual features and only for DRFs that are based on the same raw feature of a BR, i.e., in the case of the description of a pair of BRs, we first caclulate $textsim(q', d', description.SO200)$ and $textsim(q', d', description.GN300)$ and then provide the results as set of similarities (S) to obtain the mean similarity for *description.all* by calculating $meansim(S)$. Hence, each raw textual feature is encoded in three ways resulting in three DRFs for each raw textual feature.

Learning to rank. Now we have 24 DRFs (see Table 4.4) grouped into what we call a similarity vector (SV) sv expressing the similarity between two BRs as vector but we do not know how each of the DRFs contributes to the overall similarity, i.e., we want to retrieve a more sophisticated measure instead of the average of the elements (DRFs) in sv.

To help figure out this problem, we applied RankNet [37]. It is a pairwise approach trying to predict the relevance of a pair of documents based on a query. The output of RankNet is a number in the range from $-\infty$ to ∞. The higher the number, the more relevant a pair of BRs is to each other. Projected onto the problem of detecting duplicates we are faced with in defect management at Sky, it means that we want to find out if the given query BR is similar to or duplicate of another existing BR. It is pairwise because during training a relevant similarity vector sv_{rel} and an irrelevant similarity vector sv_{irr} are paired and fed into RankNet with the corresponding label, i.e., the relative rank between the two SVs which is a number between 0 and 1 [37].

Since we designed the training routine based on the RankNet proposal by Alcorn [59] and Egg [60] in the way that the "left" SV, i.e., the first one, is always the relevant one, i.e., a SV resembling high similarity between a pair of BRs, the labels (relative ranks) are all "1". Hence, samples for training are retrieved like follows:

$$sample(ori\text{-}dup, U) = (sv_{rel}, sv_{irr}, 1) \tag{4.16}$$

where: *ori-dup* : a pair of BRs where one BR is the original report (*ori*) and another one (*dup*) is a known duplicate of this report (linked with type "Duplicate" in JIRA)

U : the set of BRs without any issue links in JIRA

sv_{rel} : the relevant SV calculated from the *ori-dup* pair

sv_{irr} : the irrelevant SV calculated from a *nondup-dup* pair where *dup* is the same BR as in the *ori-dup* pair

1 : the label, i.e., relative rank.

We decided to use RankNet because it is simple to create many irrelevant SVs given a relevant SV without a predefined order due to the learning-to-rank problem being approximated by a binary classification problem.

In our case, during the training phase of RankNet which is part of the PREP stage, a relevant SV is calculated from an *ori-dup* pair. Consequently, an irrelevant SV is calculated from a *nondup-dup* pair. We know which pairs of BRs are eligible for creating relevant SVs due to the issue link type "Duplicate" in JIRA. As far as irrelevant SVs

are concerned, we picked the *dup* BR from each *ori-dup* pair and randomly selected a BR (*nondup*) from all BRs without any links. With this approach, it is possible to create many irrelevant SVs for one relevant SV and therefore creating many samples by simply replacing sv_{irr} only and reusing sv_{rel} in Equation 4.16 for each *ori-dup* pair.

4.3 Ranking Defect Reports by Similarity

In section 4.2, we have shown what we need to prepare to be able to retrieve a list of top-k recommendations of similar BRs with respect to a query BR. Ideally, we would create a SV for each existing BR paired with the query BR and sort the resulting list in descending order. However, retrieving a lot of SVs is a time consuming task which does not scale well because of the ever growing amount of BRs. Jalbert and Weimer [3] even claimed that BRs become irrelevant over time and therefore, they limited the pool of candidate BRs.

In order to satisfy the requirement of current recommendations, it is crucial to provide these close to or at the time of change or creation of the BR (see Table 6.13). Therefore, a mechanism of picking the best candidates for ranking with RankNet needs to be set in place.

Figure 4.5 shows the flow of providing top-k recommendations for a query BR which incorporates such a mechanism. During *vectorizing* textual features are encoded with the *word embedder* and for other SRFs we use the *vec encoder*. Ranking is performed in a two-pass approach where first a list of top-n potential candidates is selected for comparision with the query BR and then in the second pass, a list of top-k recommendations is created which is associated (see section 4.4) with the query BR ($k < n$).

Whenever a BR is created or updated, it has to be transformed into a VBR before it can be paired with the existing VBRs to obtain a SV for each candidate pair (see Figure 4.1). This helps to reduce the amount of "$BR \rightarrow VBR$" transformations because each update triggers only one such transformation. Afterwards the VBR can be used for retrieving a list of recommendations of similar BRs, i.e., continuing the flow shown in Figure 4.5, or it can be reused as input for creating a candidate pair when another query BR needs to obtain recommendations of similar BRs. The VBR stays untouched until its corresponding BR is updated which triggers a "$BR \rightarrow VBR$" transformation to update the VBR.

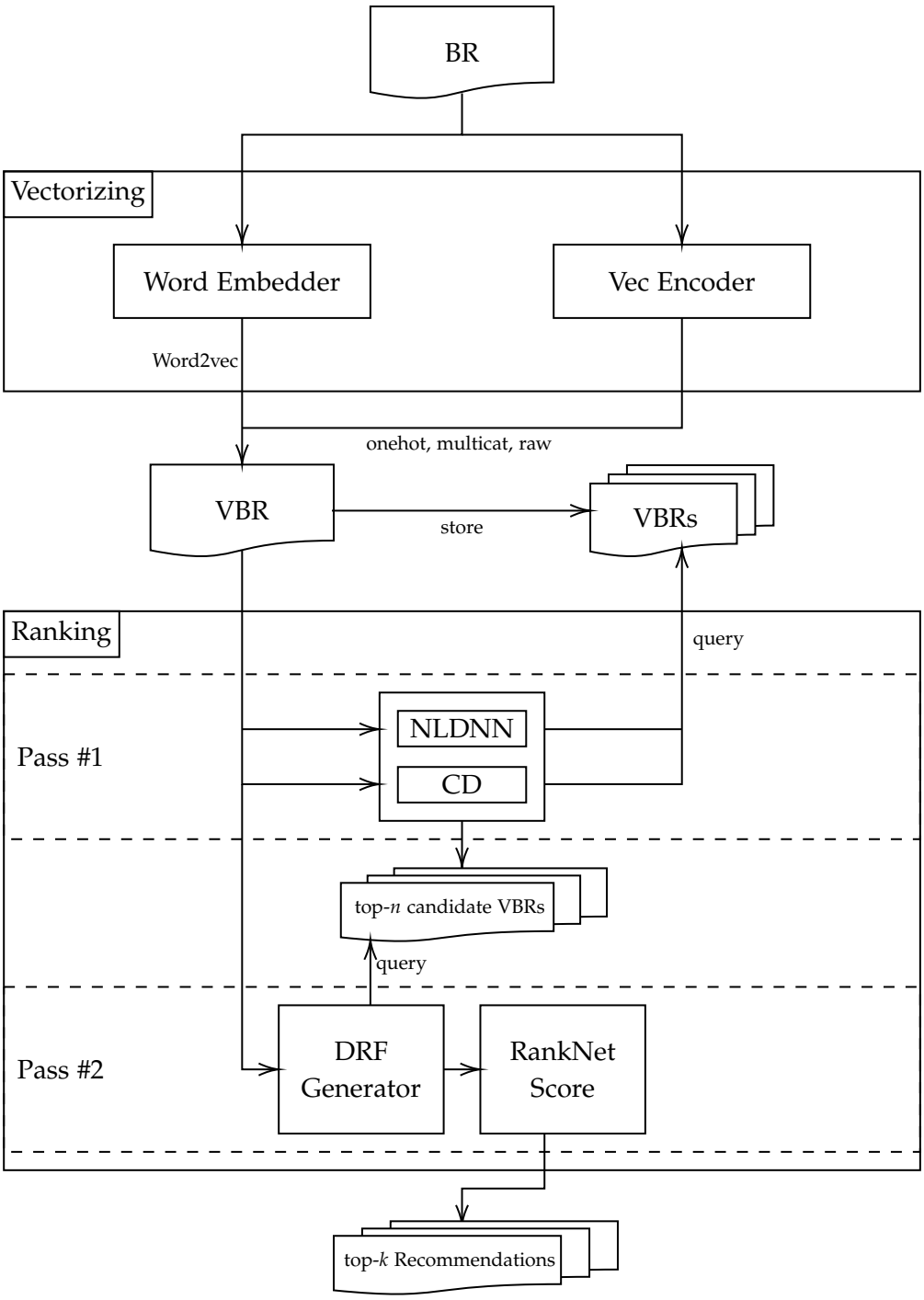
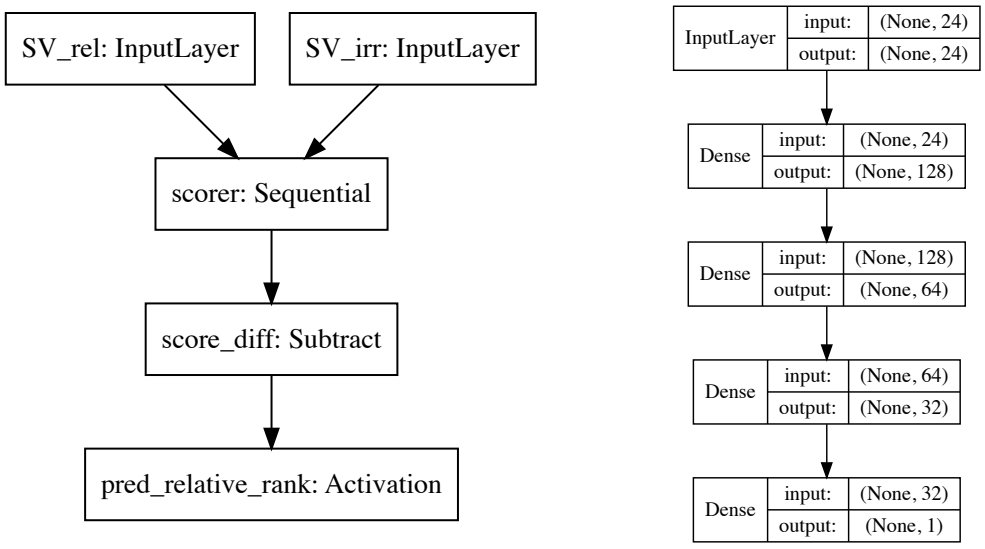


Figure 4.5: Similar bug report retrieval for a given query BR. No formal notation was used to show the flow from query to result but we use elements from flow chart notation.

A candidate pair consists of the freshly transformed VBR (obtained from the query BR) and one of the best candidates from all VBRs. The best candidates are selected by calculating the DRF *description.SO200* for each existing VBR paired with the query VBR. From the resulting list the top-n highest similarity scores are selected. At this point, we calculate the remaining DRFs for each pair consisting of the query VBR and one of the top-n candidates giving us a list of n SVs. These are then fed into RankNet which is the second pass of ranking.



(a) RankNet: Siamese-based network [37]

(b) RankNet: Scoring Function

Figure 4.6: RankNet: Neural network architectures of the siamese-based network and its neural network based scoring function. Figure 4.6b shows the expanded graph of the *scorer: Sequential* layer in Figure 4.6a.

In our approach, RankNet is designed as a siamese-based neural network [37] where the scoring function is implemented as another neural network (see Figure 4.6).

At *training time*, two SVs sv_{rel} and sv_{irr} are fed into the neural network (shown in Figure 4.6a). The goal is to minimize the amount of inversions in ranking, i.e., the output of the siamese-based neural network should not drop below 0.5 considering all labels are 1 in our case. If the output drops below 0.5 it means that sv_{irr} was ranked higher than sv_{rel} which is not what we want.

At *prediction time*, we only resort to the scoring function which is a neural network outlined in Figure 4.6b where the input layer's dimension matches the amount of DRFs (i.e. $dim = 24$). Here is where we retrieve a similarity score (relevance score) as scalar value for each pair of query VBR and top-n candidate VBRs. The resulting list of scalar similarity scores is then sorted in descending order where we select the top-k recommendations of potentially similar BRs. They are potentially similar because we get similarity scores in any case, even though in reality, they are not similar to the query BR. However, if they are not similar, the scores will be lower.

Due to how the activation functions in the scoring net (see Figure 4.6b) work, the scores are not probabilistic and can take any value in $\mathbb{R}$. Nonetheless, we observed potential upper and lower bounds during training, so that it would be possible to convert these scores into the range between 0 and 1 to express them as reasonable percentage scores which should be easier to grasp for users of such a recommender system, but we leave this task for the future.

4.4 Publishing the Ranking Result

Table 4.5: Example list of top-*k* Recommendations for query BR *DEV-805*.

#	Related Bug	Score	Summary
1	DEV-693	61.22	Mismatch between the series and episode numbers ...
2	DEV-673	59.95	Record Series action button is displayed instead ...
3	DEV-245	57.42	The hero zone of an episode level show page is ...
4	DEV-810	57.02	Intermittently, 'Remove series link' and 'Other showings' ...
5	DEV-717	55.86	Blank space seen in the action menu when entering ...
6	DEV-748	55.33	If a show page is open at the time that an unrelated ...
7	DEV-483	54.80	Duplicate buttons seen to add a series link for a ...

Summary of *DEV-805*: "Both, 'Record Series' and 'Remove series link' ... intermittently".
BR summary texts were truncated because this list just serves as an example.

As demonstrated in Figure 4.5, the result of the retrieval process is a list of top-k recommendations. The step before retrieving these recommendations is where the actuall ranking score is calculated. This is the raw ranking information shown in Table 4.5 (third column, "Score"). Along with the ranking or relevance score, we

present the rank (first column), the bug id of the similar BR (second column) and the full summary of the BR (fourth column) (summary was truncated). This list is associated as comment to the corresponding JIRA ticket for each eligible query BR which is user-configurable and is based on the JIRA-internal filter functionality.

Instead of returning just the rank along with bug id and summary, we decided to provide the score as well. We believe that users should determine a threshold for the relevance score depending on the results they see over time because each domain is different and since we do not convert the ranking scores into probabilities it is even more important to let the users decide which treshold to use. If the threshold is not exceeded, the query BR will not receive a list of recommendations even though it was eligible. However, at Sky it was not possible for the users to learn what threshold to use because of our tool creating to much noise in terms of email notifications (see section 6.2).

4.5 Comparision to Existing Approaches

In chapter 3 we presented a non-exclusive set of approaches for targeting the problem of detecting duplicates where we briefly outlined each of 7 major research contributions published from 2005 to 2016. In this section, we provide an overview of these approaches in Table 4.6 and how it compares to our approach. Apart from research contributions, we briefly examined existing commercial solutions for JIRA. Due to unavailability of the commercial solutions for standalone use (outside of the JIRA ecosystem) and application of the research approaches on the bug repository at Sky being beyond the scope of this work, we only studied the similarities and differences of the design of the techniques as far as possible.

We accessed Sky's closed bug repository and the Jenkins CI[3] (JENKINS) open bug repository to evaluate the *recall rate@k* of Defejavu, i.e., our proposed solution in this work (see section 6.1 for more details on the results).

Considering Table 4.6, our solution is closest to Sun et al. [34] regarding the ranking of SVs. Both approaches resort to RankNet introduced by Burges et al. [37] in order to provide ordered lists of top-*k* recommendations for similar BRs. However, they differ in the way how textual features are *vectorized*. We use word embeddings obtained with the help of general purpose models whereas Sun et al. [34] rely on IDF-based

[3] `https://issues.jenkins-ci.org/projects/JENKINS` (verified 2020-02-24)

Table 4.6: Overview of duplicate detection approaches.

Author	Textual Vectorizing	Similarity Method	Features	Model
Anvik et al. [2]	*unknown*	cosine similarity	textual	statistical [2, p. 39]
Hiew [22]	*TF-IDF*	cosine similarity & clustering	textual	threshold
Jalbert and Weimer [3]	custom [3, p. 56]	cosine similarity & clustering	textual & categorical	Linear Regression & threshold
Sun et al. [1]	IDF	sum of IDFs of two Bags of Words	textual (combinations of BoWs)	SVM
Sun et al. [34]	IDF [1]	REP (including the extension of BM25F) [34]	textual [1] & categorical	RNC [36, 37]
Alipour et al. [38]	IDF [1]	*textual & categorical*: REP [34], *contextual*: software word lists, *combined*: cosine similarity	textual, categorical & contextual	several (evaluation)
Yang et al. [39]	*TF-IDF* & word embeddings	cosine similarity	textual, categorical & commit messages	non-negative score (combined)
this work	word embeddings (Word2vec)	NLD, CD & RankNet score (Equation 4.7 & Figure 4.6b)	textual, categorical & datetime (Table 4.4)	NLDNN & RankNet (Figures 4.3 and 4.6)

features. Here is where the approach of Yang et al. [39] is the closest to ours because they *vectorize* textual features with the help of word embeddings as we do.

The main difference to all approaches outlined in this work and also to all other approaches best to our knowledge lies in the usage of a non-linear similarity measure (Figure 4.3). With NLD (Equation 4.5) we introduced a novel way of expressing the similarity of vectors because as mentioned before, we do not believe that the space is an Euclidean space but rather a manifold.

5 Implementation of the Recommender System Defejavu

In previous chapters we already mentioned the ITS in use is JIRA. For the purpose of similar BR detection, it is required to access the underlying data in a certain way. There are mainly two possibilities how this is possible with respect to JIRA. Implementing a plugin represents the first way and is generally preferred because of the proximity to the data. This is how the solutions presented in section 3.2 were implemented.

Unfortunately, for the approach in this work it was not possible to create a JIRA plugin because the data we accessed is handled by third parties and therefore, it would require to get permissions of these third parties. Even if we got the permissions, it would still be difficult because deploying an untested plugin on a production system is not what organisations look after. For that reason, we opted for the second way, i.e., the JIRA representational state transfer (REST) application programming interface (API) which is sufficient to extract and feedback data for our purpose.

Language. Python 3.7[1] (PY) was the language of choice for implementing the components described in sections 4.2, 4.3 and 4.4 primarily due to useful libraries like pandas[2] (PD), numpy[3] (NP), GENSIM and keras[4] (KERAS).

Environment. In order to isolate the runtime from other processes, we use Docker[5] (DOCKER) to spin up all components in respective containers. Reproducible orchestration of the environment is achieved by using Red Hat Ansible[6] (ANSIBLE).

[1] `https://www.python.org/` (verified 2020-04-05)
[2] `https://pandas.pydata.org/` (verified 2020-04-05)
[3] `https://numpy.org/` (verified 2020-04-05)
[4] `https://keras.io/` (verified 2020-04-05)
[5] `https://www.docker.com/` (verified 2020-04-05)
[6] `https://www.ansible.com/` (verified 2020-04-05)

ETL. We deployed the latest version of SYNCPIPES that was available during our research. Unfortunately, SYNCPIPES was lacking the feature of incrementally extracting issues from JIRA. Each time we launched a pipeline, all tickets were re-extracted. Therefore, we implemented a service to incrementally extracted tickets based on the *updated* time stamp of JIRA issues. Since SYNCPIPES is implemented in TypeScript[7] (TS), we also used TS instead of PY to implement this service.

Database. SYNCPIPES leverages mongoDB[8] (mongoDB) for storing its configuration. We wanted to prevent introducing another technology. Therefore, we use the same database instance as SYNCPIPES does.

Inter process communication. Unfortunately, we missed the oportunity to soleley rely on RabbitMQ[9] (RMQ) which is what SYNCPIPES uses for communication between task workers and its REST API server.

In our components, we have a mix of PY's *multiprocessing* module and RMQ. The queues managed by RMQ are used for inter component communication and *multiprocessing* queues are used for process communication within a component, i.e., within a DOCKER container.

As soon as a component is done with the current batch of tickets, it posts a message to its corresponding queue in RMQ. The next component awaits messages from the previous queue and starts working on the new batch (see Figures 4.1 and 4.5). The only exception is the transition from the ETL stage to the PREP stage where polling is applied in order to check if new tickets need to be vectorized.

Configuration. Apart from SYNCPIPES which stores its configuration in mongoDB, Defejavu's configuration is file based and stored within a single JavaScript Object Notation[10] (JSON) file. However, our configuration mapper is also able to load the configuration from mongoDB. To prevent multiple downloads of the same file, we cache the downloaded file in the temporary directory of each DOCKER container, i.e., each of the containers still needs to download the configuration into local storage but

[7] `https://www.typescriptlang.org/` (verified 2020-04-05)
[8] `https://www.mongodb.com/` (verified 2020-04-05)
[9] `https://www.rabbitmq.com/` (verified 2020-04-05)
[10] `https://www.json.org/json-en.html` (verified 2020-04-05)

all processes started within the container will have access to this configuration file. This approach helps to keep the configuration separate from the code.

6 Results

In this chapter, we present the results of the interviews and the evaluation of our approach measured by the metric *recall rate@k*.

We conducted five interviews with expert triagers at Sky. Table 1.1 shows each expert's role and experience in years. In Table 6.1, we extend this information by the weights used for calculating the score of a notion in each topic. The topics covered in the interviews are explained in section 1.3. For each topic, we compiled a set of notions each of which contributing to the *topic sentiment score*. Before we can explain this score,

Table 6.1: Group of expert triagers and their scoring weights.

Expert	Defect Management (w_D)	JIRA (w_J)	Mean ($\widetilde{w}$)	Mean (norm.) ($\overline{w}$)
SIE1	0.95	0.60	0.78	0.24
SIE2	0.50	1.00	0.75	0.23
SIE3	0.40	0.50	0.45	0.14
SM	0.23	0.45	0.34	0.10
DM	1.00	0.85	0.93	0.29
Total			3.25	

we need to describe the following mathematical items. First, we define the subsequent sets to reason about the values from Table 6.1:

$$E = \{SIE1, SIE2, SIE3, SM, DM\} \tag{6.1}$$

$$D = \{19, 10, 8, 4.5, 20\} \tag{6.2}$$

$$J = \{6, 10, 5, 4.5, 8.5\} \tag{6.3}$$

$$W = \{w_D, w_J\} \tag{6.4}$$

In Equation 6.5 the expert's experience in defect management is expressed as scoring weight. We define this item as follows:

$$w_{D,e} = \frac{d_e}{max(D)} \quad e \in E \quad d \in D \tag{6.5}$$

where: E : the set of experts defined in Equation 6.1
d : the expert's experience in defect management (years) (see Table 1.1)
D : the set defined in Equation 6.2
$w_{D,e}$: the expert's experience in defect management (weight) (see Table 6.1).

The expert's experience with JIRA is another scoring weight and is defined similarly to $w_{D,e}$:

$$w_{J,e} = \frac{j_e}{max(J)} \quad e \in E \quad j \in J \tag{6.6}$$

where: j : the expert's experience with JIRA (years) (see Table 1.1)
J : the set defined in Equation 6.3
$w_{J,e}$: the expert's experience with JIRA (weight) (see Table 6.1).

With the weights from Equations 6.5 and 6.6, we get a combined weight:

$$\widetilde{w}_e = \frac{1}{m} \sum_i w_{i,e} \quad i \in \{D, J\} \quad e \in E \tag{6.7}$$

where: m : $|W|$ (see Equation 6.4)
$w_{i,e}$: an expert's scoring weight (see Equations 6.5 and 6.6)
$\widetilde{w}_e$: the mean of the expert's scoring weights $w_{i,e}$.

Now we defined all items required for the actual scoring weight used in calculating the *topic sentiment score* to evaluate a topic:

$$\overline{w}_e = \frac{\widetilde{w}_e}{\sum_i \widetilde{w}_i} \quad i \in E \quad e \in E \tag{6.8}$$

where: $\widetilde{w}_i$: an average scoring weight (see Table 6.1 and Equation 6.7)
$\widetilde{w}_e$: same as $\widetilde{w}_i$ but fixed to Expert e
$\overline{w}_e$: the normalized average scoring weight used for calculating the *topic sentiment score* of a topic.

Based on the answers given by the experts and the notions we extracted from these answers for each topic we linked a pair of a notion c and an expert e with one of the following values (opinions):

$$o_{c,e} = \begin{cases} 1 \\ 0.5 \\ 0 \end{cases} \qquad e \in E \tag{6.9}$$

where: c : a notion (see the answers to the research questions from section 1.3 in this chapter for all the notions, i.e., in Table 6.6)
$o_{c,e}$: 1, if the expert agreed to the notion within a topic
0.5, if the notion was not mentioned by the expert
0, if the expert disagreed with the notion.

Combining Equations 6.8 and 6.9 the notion score $ns_{t,c}$ is defined like this:

$$ns_{t,c} = \sum_{e} \overline{w}_e o_{c,e} \quad e \in E \quad c \in C_t \quad t \in T \tag{6.10}$$

where: T : the set of topics discussed in the interviews
C_t : the set of notions extracted for a topic t
$ns_{t,c}$: the notion score for a notion c in a topic t.

Finally, the *topic sentiment score* (ts) is defined as follows:

$$ts_t = \frac{1}{|C_t|} \sum_{c} ns_{t,c} \quad c \in C_t \quad t \in T \tag{6.11}$$

where: ts_t : a score indicating the amount of agreement towards all notions within a topic t, the closer the score is to 1, the more the experts agree to the notions, i.e., a higher score means a more coherent sentiment across all experts regarding the topic t.

Table 6.2 shows a generic example of how the evaluation matrices look like generated from the notions which we extracted from the experts' answers. It should serve as a template indicating where the items defined in the equations from this chapter can be located within an evaluation matrix of a topic.

The next sections aim to answer the research questions presented in section 1.3. Additionally, observations and other insights are provided which are not covered by any of the research questions in particular.

Table 6.2: Sentiments of the topic *SampleTopic*.

Notion (c)	e_1	e_2	...	e_m	Score (ns)
c_1	o_{c_1,e_1}	o_{c_1,e_2}	...	o_{c_1,e_m}	$ns_{SampleTopic,c_1}$
c_2	o_{c_2,e_1}	o_{c_2,e_2}	...	o_{c_2,e_m}	$ns_{SampleTopic,c_2}$
⋮	⋮	⋮	⋱	⋮	⋮
c_n	o_{c_n,e_1}	o_{c_n,e_2}	...	o_{c_n,e_m}	$ns_{SampleTopic,c_n}$
topic sentiment score					$ts_{SampleTopic}$

6.1 General Insights

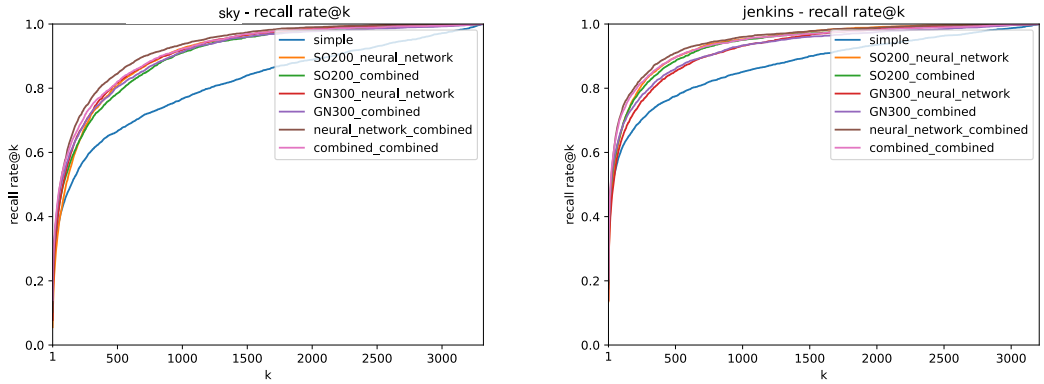
The results presented in this section were calculated using the dev machine outlined in Table 6.3.

Table 6.3: Development Machine Stats.

Model	MacBook Pro (15-inch, 2017)
Processor	2,8 GHz Quad-Core Intel Core i7
Memory	16 GB 2133 MHz LPDDR3

First, we want to focus on only using BR descriptions for retrieving a list of most similar BRs. As explained in sections 4.2 and 4.5, before the similarity can be calculated applying the vector space model is a common approach, i.e., using *TF-IDF* (see Table 4.6). In our approach, we resorted to word embeddings obtained from using Word2vec.

Figure 6.1 shows the results, i.e., *recall rate@k*, for the JIRA projects Sky and JENKINS. For both projects, we extracted more than 3000 *ori-dup* pairs. In total, we trained four NLDNNs: two for each project based on word embeddings obtained from using the general purpose models SO200 and GN300. Figures 6.1a and 6.1b, both show seven curves each. Each of the curves represents the *recall rate@k* (y axis) for a given k (x axis). The data for each curve is obtained by aggregating the amount of hits at a given k. A hit is recorded if the list of top-k recommendations contains the *ori* BR with respect to an *ori-dup* pair where each *dup* BR represents a query BR. In other words, we consider all the *dup* BRs from the aforementioned *ori-dup* pairs to be query BRs to evaluate our approach and if the obtained list of similar BRs contains the *ori* BR, we record a hit. Hits are accumulated, i.e., if the list of similar BRs contained the *ori* BR with a list size



(a) Sky **(b)** JENKINS

Figure 6.1: *recall rate@k* (all).

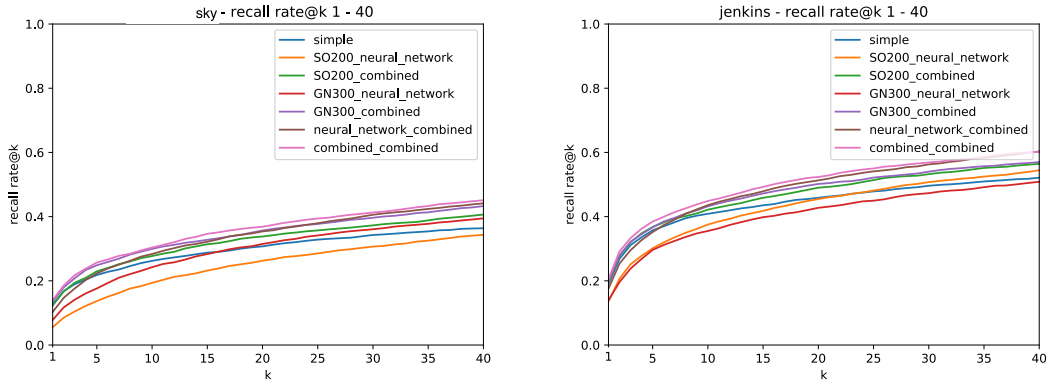
of $k = 1$, for instance, the *ori* BR will be contained in any other list of recommendations with $k > 1$ for the same query BR (*dup*).

Table 6.4: Similarity Methods for Hit Aggregation.

Curve	Similarity Method	Top List
simple	CD	top-k
SO200_neural_network	$\text{NLDNN}_{\text{SO200}}$	top-k
SO200_combined	$\text{NLDNN}_{\text{SO200}}$+ CD	top-k
GN300_neural_network	$\text{NLDNN}_{\text{GN300}}$	top-k
GN300_combined	$\text{NLDNN}_{\text{GN300}}$+ CD	top-k
neural_network_combined	$\text{NLDNN}_{\text{SO200}}$+ $\text{NLDNN}_{\text{GN300}}$	top-k
combined_combined	$\text{NLDNN}_{\text{SO200}}$+ $\text{NLDNN}_{\text{GN300}}$+ CD	top-k
ranked@f14n1000	$\text{NLDNN}_{\text{SO200}}$+ CD + RankNet	top-n & top-k

"ranked" curve initially appears in Figure 6.3

The curves mainly differ in how the underlying similarity is calculated. Table 6.4 shows the corresponding methods we relied on when collecting the data points. The *simple* plot is obtained by calculating CD of the descriptions of two BRs and is considered to be the reference plot for all other curves in Figures 6.1, 6.2, 6.3, 6.4 and 6.5. For both projects (see Figure 6.1) the improved performance of NLDNN in contrast to CD is clearly visible. This is important for later when we refer to the results of the *ranked@f14n1000* curve in Figures 6.3, 6.4 and 6.5.



(a) Sky **(b)** JENKINS

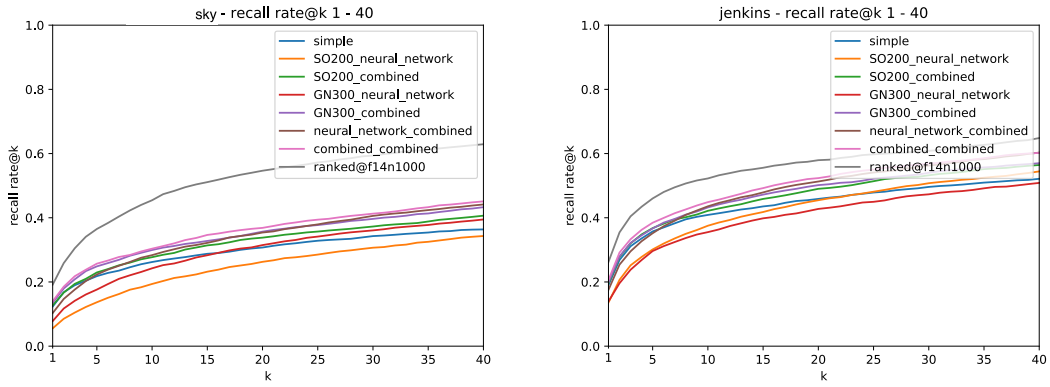
Figure 6.2: *recall rate@k* (top 40).

Figure 6.2 shows the results up to $k = 40$ of the plots from Figure 6.1. We highlight this portion of the curves because the higher k gets, the less useful the list of recommendations becomes. This is because of the amount of results the user has to consider. It is best compared to a search engine where the task is to provide the most relevant results within the first few results. For the short period of Defejavu's uptime, we picked $k = 7$ but the experts in the interviews preferred a shorter list of recommendations, i.e., $k = 4$, especially when considering the effort to check each of the individual recommendations within such a list. Even if the best similarity method (*combined_combined*) is used, lists with only four results offer a *recall rate@4* of 25% (Sky) and 38% (JENKINS).

Second, when we introduced a second pass of ranking, i.e., not only using the description of a BR but also other features (Table 4.4), by incorporating a learning-to-rank algorithm like RankNet (Figure 4.5), we experienced a high increase in the *recall rate@k*. However, for improved comparability between Sky and JENKINS, we stripped down the set of DRFs because not all fields were available on both JIRA projects. Therefore, we resorted to the features a BR offers by default in both projects:

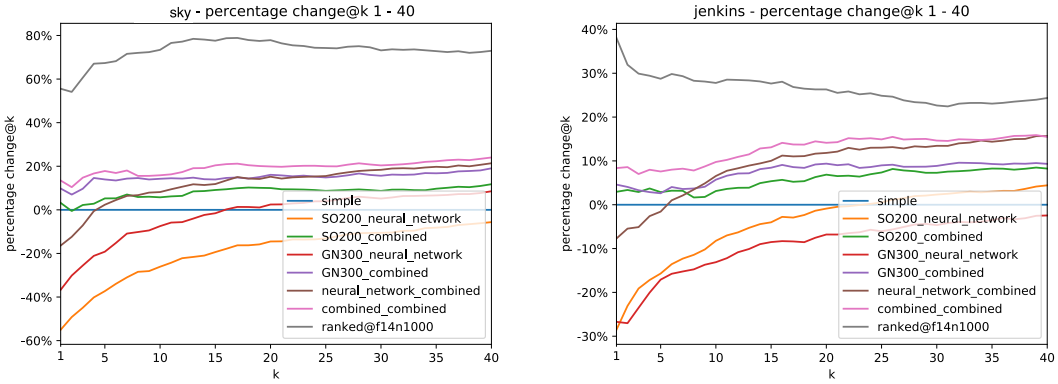
$$rf_i \quad i \in \{1, 2, 5, 6, 9, 10, 11, 15, 17, 18, 19, 20, 21, 24\} \quad \text{(see Table 4.4)}$$

Figure 6.3 is basically the same like Figure 6.2 with the exception of including the *ranked@f14n1000* curve where the name contains the amount of features and the value



(a) Sky **(b)** JENKINS

Figure 6.3: *recall rate@k* with ranking (top 40).



(a) Sky **(b)** JENKINS

Figure 6.4: Comparision of percentage change (top 40).

of n. With $k = 4$, we observed a percentage change of +67% (Sky) and +29% (JENKINS) compared to the *simple* curve increasing the *recall rate@4* to 34% (Sky) and 43% (JENKINS) compared to the previous best method *combined_combined*.

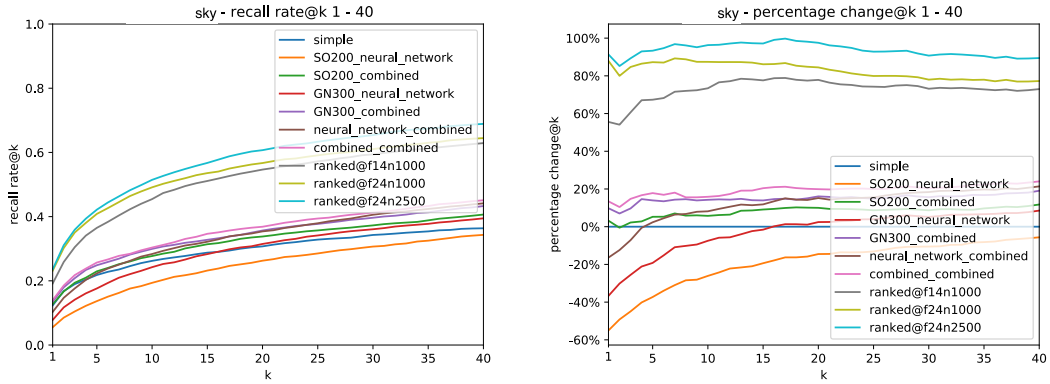
For obtaining the top-n list, a look at Figure 6.1 is required. We set $n = 1000$ and resorted to the similarity method *SO200_combined*, simply because we started our research with the general purpose model SO200. We leave the comparision of other similarity methods for aggregating the top-n list for the future. The *recall rate@1000* is picked so that both the results in the top-n list and the time consumed for computing are reasonable. It still takes several seconds on the dev machine (Table 6.3) to calculate DRFs for 1000 pairs which is a shortcoming of Defejavu. With $n = 1000$ the *recall rate@1000* is at 91% (Sky) and 95% (JENKINS). Obviously, there is room for improvement but in most cases the most relevant BRs are contained in the top-n list. If we were to resort to CD, we would see these BRs only 77% and 85%, respectively, of the time. By increasing n we would see improved recall rates but at the cost of extended computation time.

Interestingly, GN300 performs better than SO200 for the Sky dataset. The opposite is the case for the JENKINS dataset where SO200 returns better results. This is best observed when looking at the percentage change curves in Figure 6.4. It seems that this dataset is more specific to software development than the data we extracted from Sky where GN300 is better suited because of its broad range of topics covered.

We conclude and we also believe that training a custom Word2vec model instead of relying on general purpose models can further improve the curves from Figure 6.1 leading to a better performance during the second ranking pass (see Figure 4.5) but this is left for the future.

At the end of this section, we want to include the results for ranking presented in this section when all DRFs from Table 4.4 were used. Unfortunately, we only have results based on the Sky dataset.

Considering *recall rate@4*, in Figure 6.5 one can see that adding more features is far more effective (+4%, *ranked@f24n1000*) than just increasing n which results in a change of only +1.5% (*ranked@f24n2500*) but more than doubles the cost of time consumed when calculating the top-n list. For us, the most important takeaway from looking at the curves in this section is the importance of multiple features and not necessarily focusing only on textual features.



(a) *recall rate@k* **(b)** *percentage change@k*

Figure 6.5: Performance on the Sky dataset: 24 DRFs.

Observations of the *scrub* meeting

We tracked the time taken for each defect discussed in the 15 sessions where we were invited to participate as guests of the *scrub* meeting. In these sessions, 167 BRs were discussed in an average time of 14 minutes 29 seconds and roughly 11 BRs per session. The total time taken for all 15 sessions was 3 hours 34 minutes and 49 seconds. A ticket was fully discussed and updated after a median time of 59 seconds. Interestingly, roughly 20 seconds of the discussion time per ticket were attributed to updating the BR, i.e., changing the component, etc.

The following sections will outline our observations in more detail also by referring to the research questions presented in section 1.3 but Table 6.5 already provides a quick overview of what the experts would like to see improved and one of the notions is *Duplication Detection* which is what motivated us to conduct this research.

We were surprised that the next ticket was always announced by voice by the DM. Only in the last session, screen sharing was used to aid identifying the currently discussed BR. Therefore, it is natural that at least one of the experts suggested *Screensharing* as an improvement for the *scrub* meeting.

For the other insights, we begin with the answer to the first research question.

Table 6.5: Sentiments of the topic *Triaging Improvements*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Increased Strictness		+	+	+	+	0.88
Screensharing			+			0.57
Extended Ticket Discussion			+			0.57
Increased Team Representation	+					0.62
Reduced Ticket Inspection	+					0.62
Reduced Custom Fields		+				0.62
Duplication Detection		+			+	0.76
Better Prioritization		+	+	+		0.74
$ts_{\textit{Triaging Improvements}}$						0.67

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

RQ1 Do duplicate defect reports pose problems in a software development life cycle?

Results. When considering the initial feature request for JIRA to support flagging similar or duplicate issues natively [24], the answer to this research question is clearly *yes*.

As already mentioned, JIRA has been around for nearly two decades. The lack of this feature motivated several researchers in the past to investigate the problem with duplicate defect reports with each contribution improving the existing state of the art approach in a certain way, either by automatically assigning reports to proper developers or development teams [2, 3], flagging similar issues by providing a list of top-k most similar recommendations [1, 2, 22, 34, 38, 39] or even automatically hiding duplicates from triagers and developers [3]. The creators of JIRA even claim [24] due to existing JIRA plugins [23, p. 9–10] to not prioritize the native implementation of this feature.

Clearly, one can see that duplicate bug reports remain an issue as of writing this thesis. This is backed by the sentiments regarding *Duplicate Reports* we obtained through the interviews with the triage experts at Sky. We identified eight notions

($\left|C_{Duplicate\ Reports}\right| = 8$) in the answers given by the experts which are depicted in Table 6.6.

Table 6.6: Sentiments of the topic *Duplicate Reports*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Encounter	+	+	+	+	+	1.00
Problematic	+	+		+	+	0.93
Variable Duplicate Rate	+					0.62
Time Wasted on Triaged Duplicates	+	+	+	+	+	1.00
Increased Stress	+	+	+	+	+	1.00
Hard to Identify				+	+	0.69
Time Wasted while Triaging	–	+	–	+	+	0.62
Time Wasted while Reporting		+		+		0.67
$ts_{Duplicate\ Reports}$						0.82

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

All of them already have been faced with duplicate defect reports and most consider these to be problematic.

They all agree on the notion *Time Wasted* because of duplicate reports especially when considering the stage of a bug as soon as it has been triaged and is awaiting a fix, i.e., the development team or one of the team's members started to analyse the defect. This increases the stress on the developers because of the higher workload due to redundant work where effort could be targeted at proper bug reports instead of already well-known and maybe even fixed ones. Time is also wasted for the reporter who raised the defect in the ITS according to some of the experts. There is a split opinion on the notion *Time Wasted while Triaging* and by that we mean the time spent to prepare and perform the triage meeting. A few experts do not believe that time is wasted discussing duplicate reports because it prevents wasted effort on the development side of things while the other experts confirm that time is also wasted on the triage of duplicate reports. Interestingly, the experts farther away from writing source code (SM and DM) mentioned duplicates to be hard to identify as such. One of the expert triagers emphasized the problem of duplicates is rather easy to cope with when a project is not

too busy. Contrarily, in times where the project is under a lot of pressure, the duplicate rate rises and these reports slip through the triage process more easily.

Summary. With $ts_{Duplicate\ Reports} = 0.82$, duplicate reports *still* pose problems in a software development life cycle nowadays.

6.2 Introduction of Defejavu to the Defect Management Process at Sky

Before we address the results of RQ2, we have to point out that unfortunately it was not possible to properly launch the evaluation of Defejavu. It is designed to rank all new or updated BRs, however, it also provides the feature of selecting which rankings should be published (see section 4.4). This is achieved by applying a given set of filters. Since the ITS in use was JIRA, we resorted to JIRA's builtin filtering functionality instead of creating our own. We had to implement an abstraction and include it into Defejavu to be able to interact with JIRA filters through the REST API. Then, the JQL filter was configured within JIRA. Here is where we made a mistake. Instead of filtering for relatively new BRs where "relatively new" means "created within the last 30 minutes" based on the start of the pipeline run for a set of query BRs (see section 4.1 and Figure 4.1), we also mistakenly included BRs that have been updated within the same time span.

Due to being able to update almost any BR, Defejavu provided top-k lists for BRs which can be of any age. The older a BR is, the more users had the opportunity to relate themselves with this BR by posting a comment or updating the description, for instance. In any case which leads to an "update" of a BR in JIRA, the user who provides the update is added to the list of watchers of the BR.

Consequently, when an update to a BR is issued, all watchers are notified by email. Since the JIRA project at Sky is frequented very often during a day causing many updates to BRs, we caused a lot of noise with Defejavu in the watchers' email inboxes. Because of this we had to take Defejavu offline.

While the fix was blatantly easy where only the JQL filter had to be updated, by producing a great portion of noise in the inboxes, we established an incisive amount of distrust towards Defejavu which is hard to recover from.

RQ2 What are the reasons for not incorporating tools aiming at detecting duplicate reports?

Results. The age of the feature request to introduce similar issue detection [24] in JIRA is nearly close to the age of the tool itself. Yet nothing close to an automated approach for detecting similar issues is implemented at Sky. Considering the age of the feature request [24] this is rather unexpected especially when there are plugins available able to provide this feature. We did not anticipate to receive useful feedback to answer this research question when asked directly. In fact, some of the experts mentioned without being asked that they do not know the reasons for Sky not introducing a SBRDT on their own. They pointed vaguely at busy projects and shortage of resources engaged with improving the detection of such reports. Instead, we grouped a few topics to collect a set of sentiments from the experts to be able to reason about the outcome.

Table 6.7: Sentiments of the topic *Lack of Native Similar Issue Detection*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Versatile Descriptions	+	−			+	0.65
Hard Task for NLP	+	+	+	+		0.86
Lack of Information	−	+				0.50
Irrelevant Feature				+		0.55
$ts_{\textit{Lack of Native Similar Issue Detection}}$						0.64

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

First, we discussed the *Lack of Native Similar Issue Detection* in JIRA. From the answers given, we identified four major notions ($\left|C_{\textit{Lack of Native Similar Issue Detection}}\right| = 4$) which can be seen in Table 6.7. It can be assumed that all of the experts suppose detecting duplicate or similar bug reports to be a *Hard Task for NLP* even though the DM did not mention this notion directly. However, according to the DM, bug reporters are all unique and thus create reports in a unique way which leads to *Versatile Descriptions* but can be also considered challenging for natural language processing (NLP) (*Hard Task for NLP*).

The notion *Versatile Descriptions* seems to be controversial, similarly to the whole topic with $ts_{\textit{Lack of Native Similar Issue Detection}} = 0.64$ which is the most neutral *topic sentiment score*

among all topics discussed in the interviews. The closer the score is to $o_{c,e} = 0.5$ the more unique notions were extracted from the answers. Even more, one expert claimed the lack of versatile bug report descriptions (*Lack of Information*) to be the cause for the lack of native automated similar issue detection in JIRA. In their view, the descriptions of bug reports often do not provide enough detail to be useful for fixing the fault or even for triage and thus it is also hard to implement an automated solution to highlight similarities between reports. Interestingly, another expert indicated a potentially low value of native similar issue detection incorporated in JIRA (*Irrelevant Feature*). According to the SM, the people behind JIRA probably do not consider implementing the feature to be beneficial for the product (JIRA) and this idea matches the update by Atlassian to the feature request [24].

Second, the topic *Preventive Measures against Duplicate Reports* was discussed which yielded five notions ($\left|C_{\textit{Preventive Measures against Duplicate Reports}}\right| = 5$) as listed in Table 6.8. Since the experts believe that automating similar issue detection is hard (*Hard Task for NLP*) it is not surprising that some of them mentioned the *Accuracy of Automation*. A tool that provides recommendations for similar bug reports has to be accurate and reliable. As long as that is not the case, it cannot be incorporated into the defect management process. This is exactly what happened to Defejavu as described at the beginning of section 6.2. With that said, it is also not surprising that the experts believe automating the flagging of similar issues to be hard (*Hard Task for Automation*) which is very similar to the notion *Hard Task for NLP* from the topic *Lack of Native Similar Issue Detection*. Another view and a probable cause for not introducing automated tools into

Table 6.8: Sentiments of the topic *Preventive Measures against Duplicate Reports*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Accuracy of Automation	+		+			0.69
Hard Task for Automation	+	+				0.74
Implementation of Measures		+	+	−	+	0.78
Manual Search Before Submission			+	−	+	0.66
Hard Task for People				+	+	0.69
$ts_{\textit{Preventive Measures against Duplicate Reports}}$						0.71

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

the triaging process to flag duplicates is the practice of preventive measures already set in place (*Implementation of Measures*):

- Search for existing bug reports before raising a new report (*Manual Search Before Submission*, see Table 6.8)
- Triage output is shared with external bug reporters
- Bug report description scraper (proof of concept tool using NLP) (discontinued).

However, not all experts share the same opinion and especially the SM seems to be convinced of the opposite, but he states that preventing duplicate defect reports is a *Hard Task for People* (bug reporters). We suppose that this is probably the reason for not agreeing on the notion *Manual Search Before Submission*.

Last, we obtained a notion from the same expert who pointed out the *Lack of Information* in bug report descriptions that tool assisted approaches not necessarily struggle because of the notions used to answer RQ2 but because of a process problem (*External Problem*, see Table 6.13) causing low quality bug reports. In the expert's mind, any tool targeted at finding similar issues would struggle if bug reports did not meet a certain level of quality.

Summary. We believe that especially the notions *Versatile Descriptions* and *Hard Task for NLP* from the topic *Lack of Native Similar Issue Detection* are most influential for not introducing automated tools to detect similar bug reports. The topic *Preventive Measures against Duplicate Reports* mostly contributes to the the reasons (RQ2) because of the notions *Hard Task for Automation* and *Implementation of Measures*. A process problem (*External Problem*) paired with the notions *Lack of Information* and *Irrelevant Feature* could also hinder automation in this regard but requires more investigation. However, we can only suppose what the reasons are because the scores (*topic sentiment score*) for both topics are rather neutral with $ts_{\textit{Lack of Native Similar Issue Detection}} = 0.64$ and $ts_{\textit{Preventive Measures against Duplicate Reports}} = 0.71$.

6.3 Manual vs. Tool Supported Defect Triage

RQ3 Does tool assisted detection help in highlighting duplicates before the *scrub* meeting?

Results. From our research we learned that triaging is split into two stages at Sky (see section 2.2). In the answer to this research question we mainly focus on the first stage where the scrub meeting is prepared by the DM. However, many of the insights can be mapped to the answer to RQ4. Since we covered several topics (see Tables 6.9, 6.10, 6.11, 6.12 and 6.13) to collect the notions most useful to answer other research questions as well, we want to emphasize the notions that are essential to the results relevant for RQ3.

Table 6.9: Sentiments of the topic *Triage Preparation*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Low Frequency	+	+	+	+		0.86
Post Work	+			+	+	0.81
Brief Ticket Inspection	+	+	+	+	−	0.71
$ts_{\textit{Triage Preparation}}$						0.80

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

First, we take a look at *Triage Preparation* (see Table 6.9) and all the corresponding notions ($\left|C_{\textit{Triage Preparation}}\right| = 3$). The experts admitted to rarely preparing (*Low Frequency*) for the triage meeting except for the DM who actively manages the list of bug reports open for triage. This task involves pre-analysing the pending reports which can include new but also existing ones that need triaging. From personal communication with the DM outside the interviews we learned that this task is also heavily relying on memory of existing bug reports that are duplicated by the pending ones, i.e., it is common for duplicates to slip through this pre-triage stage into the triage meeting itself. Actually, when we reached out to the DM initially, it was very quickly apparent that his attitude is very positive towards an automated approach primarily to reduce the reliance on memory and to improve the linkage of bug reports. Since the other experts do not really prepare for the triage meeting and resort to post meeting work where

reports might be inspected to strengthen the understanding of the ticket, i.e., to assign the report to the proper development team in case the report reappears on the triage meeting, we do not have anymore valuable results from the topic *Triage Preparation* for this research question other than the assumption of recommendations of similar tickets improving the understanding of the overall issue if information about a bug is spread over multiple tickets [61] (*Spread of Information*, see Table 6.12).

Table 6.10: Sentiments of the topic *Triaging Challenges*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Correct Assignment	+		+	+		0.74
Duplication		+	+			0.69
Lack of Information		+			+	0.76
Understanding the Ticket		+	+	+	+	0.88
Big Picture (Software Stack)				+		0.55
Understand the Priority				+	+	0.69
Prevention of Waste of Time					+	0.64
$ts_{Triaging\ Challenges}$						0.71

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

The second topic was *Triaging Challenges* ($|C_{Triaging\ Challenges}| = 7$). To answer RQ3, we picked all but the notion *Correct Assignment*. Our approach is not meant to assign a bug report to a capable developer. However, there are approaches that support automatically assigning a suitable developer or development team [2, 3].

The main challenge in triaging according to the interviewed experts is *Understanding the Ticket*. It ties in with the notion *Lack of Information* already mentioned by some of the experts previously in *Lack of Native Similar Issue Detection* and brought up again while discussing the current topic. Because of many individual reporters creating reports in their own way (*Versatile Descriptions*, see Table 6.7) it is the primary cause for the expert triagers to struggle when handling pending bug reports. According to the DM, defects can be raised in a non-native (non-english) language because of distributed teams in different territories which in turn makes it very hard for the triagers to understand what the report is about. Unfortunately, we missed the opportunity to question the lack of inclusion of non-english speaking triagers at this point, so we do not have any results

on this notion, but we believe that similar issue detection could help *Understanding the Ticket*, i.e., if one report is information rich and the other is not, yet the tool was able to highlight the relevance between the two tickets.

Another challenge lies in *Duplication* itself. Although not all mentioned it directly as challenge when discussing this topic, the triagers left the feeling behind of struggling with too much duplication which is backed by our results of RQ1. Clearly, this is the main task of automated deduplication or similarity flagging tools to improve upon. *Understand the Priority* of a bug report is key for release management and is considered to be tough by the experts. They even ask for support from colleagues in release management to improve triaging and *Prevention of Waste of Time*.

The rare notion *Big Picture (Software Stack)* is an interesting take on triaging which we did not anticipate in the first place. It should be fairly obvious that understanding the software stack simplifies the process of correct assignment of bug reports to the proper development team.

In our opinion and based on the notions with respect to *Triaging Challenges* we obtained in the interviews, SBRDTs affect the pre-triage stage and can save time to spend effort on more meaningful tasks which is the metric companies seek to improve in the end. One expert even mentioned the potential for total automatic triaging without the need for a scrub meeting at all but in our view and also in the view of the other experts detecting similar bug reports is a hard task and without 100% reliability there is too much room for error resulting in damage that could cost the company valuable money. Especially in terms of *Understanding the Ticket*, we believe that an improvement could be achieved because as mentioned before, multiple similar tickets can contain unique information. But also in terms of *Understand the Priority*, we are convinced of the benefits of tool assisted similar issue detection, for instance, because of bugs being flagged as similar to already fixed high priority ones that reappeared in a new release of the software.

The next topic, namely *In-depth Issue Discussion*, originated from observations of the scrub meeting. In general, bug reports are discussed rather quickly (see *Observations of the scrub meeting* in section 6.1). However, under certain circumstances, these reports are covered more in-depth. We wanted to learn about the reasons for these extended discussions and relate these reasons to the pre-triage stage because we found a relation between the extra time spent on such bug reports and the report being resolved as duplicate in the *scrub* meeting.

Table 6.11: Sentiments of the topic *In-depth Issue Discussion*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Slow Ticket Progress	+					0.62
Unclear Tickets	+		+	+	+	0.88
Reappearing of Same Tickets	+					0.62
High Frequency of Same Problem	+					0.62
Uncompleted Feature Work	+					0.62
High Assignment Rate Per Team		+				0.62
Proper Assignment		+		+	+	0.81
$ts_{\textit{In-depth Issue Discussion}}$						0.68

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

We identified seven notions ($\left|C_{\textit{In-depth Issue Discussion}}\right| = 7$) (see Table 6.11). Again, one notion is very prominent among the answers of the experts. *Unclear Tickets* not only cause difficulties in *Understanding the Ticket* (Table 6.10), but also ensure that a bug report is taken a closer look at. In our first interview, the expert mentioned more detailed reasons which do not directly contribute to the answer of this research question, i.e., *Slow Ticket Progress* and *Uncompleted Feature Work*[1]. However, *Reappearing of Same Tickets* and *High Frequency of Same Problem* are difficulties which could be simplified by SBRDTs. Regarding *High Frequency of Same Problem*, in particular, is where we see the best fit for Defejavu. Especially when the similar report is a TYPE I duplicate. The notions *High Assignment Rate Per Team* and *Proper Assignment* can be indirectly affected by tools like Defejavu. Nevertheless, these notions would be better suited to be included in the answer to RQ4 but we do not believe that such tools can help to reduce the time spent discussing a particular bug report, so we leave them be just as information for the reader.

We already mentioned *Spread of Information* while presenting *Triage Preparation* (see Table 6.9). Now we want to take a closer look at this notion and the other three ones ($\left|C_{\textit{Linking Policy}}\right| = 4$) from the topic *Linking Policy* (see Table 6.12). Interestingly, at Sky or more specifically in the teams we were in touch with, a linking policy as such (*No Linking Policy*) is not known or applied according to the experts but there is a Atlassian

[1] Uncompleted product features that are already being tested

Table 6.12: Sentiments of the topic *Linking Policy*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
No Linking Policy	+	+	+	+	–	0.71
Unclear Main Ticket Rule	+				+	0.76
Common Sense Rule for Duplicate Links		+	+	+	–	0.59
Spread of Information		+				0.62
$ts_{\textit{Linking Policy}}$						0.67

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

Confluence[2] (CONFLUENCE) page with some rules regarding links authored by the DM who pointed it out during our interview.

Linking duplicates to the original ticket is mandatory, yet as seen in Table 6.12 most experts consider this to be common sense (*Common Sense Rule for Duplicate Links*) instead of a clear rule. Sometimes, the chronologically more recent BR is considered to be the main report and older reports are then resolved as duplicates. Again, it is assumed by most of the experts that there is no clear rule for defining the main ticket (*Spread of Information*) in case of a duplicate report.

However, the ruleset of the DM demands newer tickets to be resolved as duplicate and the oldest one should be kept as the main ticket. Nonetheless, even the DM described an exception to this rule in our interview: the ticket with the most useful and detailed information will be used as main ticket and sometimes this can cause the older ticket to be resolved as duplicate.

Because of this and in contrast to Sun et al. [1] and Alipour et al. [38], we did not opt for *oldest ticket equals main ticket* in our approach. Instead, we neglected the direction because our solution does not rely on clustering. Furthermore, incorporating the direction of the link as constraint would not reflect the real world at Sky.

Regardless of the rule for the main ticket, we believe that linking adds to improved documentation within an ITS. By highlighting possible duplicates or similar issues at least, we are convinced of SBRDTs helping to build an interwoven ITS easy to follow

[2] `https://www.atlassian.com/software/confluence` (verified 2020-03-03)

by users and newcomers who are unfamiliar with the new domain especially when considering the *Spread of Information* as mentioned by one expert and Bettenburg et al. [61].

Table 6.13: Sentiments of the topic *Tool Assisted Similar Issue Detection*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Positive Disposition	+	+	+	+	+	1.00
Support Triage	+	+	+	+		0.86
Ease of Use	+		+	+	+	0.88
Early Feedback	+	+	+	+	+	1.00
Pre-Submission Feedback			+	+	+	0.76
Relevance Score	+					0.62
Limited List of Most Relevant Tickets	+	+	+	+	+	1.00
Simple Bug Tracking System Integration		+		+	+	0.81
External Problem		+				0.62
Reduced Time Wasted			+	+		0.62
$ts_{\textit{Tool Assisted Similar Issue Detection}}$						0.82

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

Even more, when similar tickets are suggested close to or before creation of a bug report, it is considered to add the most value as seen in Table 6.13 which introduces the next topic, namely *Tool Assisted Similar Issue Detection*. According to the experts, *Early Feedback* and even *Pre-Submission Feedback* is considered to be the most helpful. They are all positive regarding tool assisted similar issue detection (*Positive Disposition*) and consider it to be supportive with respect to triaging (*Support Triage*) except for the DM who did not mention this notion directly, but due to his attitude towards such tools, we still believe that in his view it adds value in the pre-triage meeting stage especially when considering his approval in terms of *Early Feedback*. Automated similar Issue detection can also help to reduce wasted effort spent on bug reports that are duplicates of TYPE I (*Reduced Time Wasted*) when considering the notion *Time Wasted on Triaged Duplicates* from the topic *Duplicate Reports* (Table 6.6).

The remaining notions (*Ease of Use*, *Relevance Score*, *Limited List of Most Relevant Tickets* and *Simple Bug Tracking System Integration*) from Table 6.13 will be addressed when

presenting the results of RQ5. Additionally, we picked the notion *Relevance Score* to be included in the results of RQ4.

Summary. Based on the discussions we had with the experts in the interviews, the answer to this research questions is *yes*.

Even more, we identified the following advantages of automated similar issue detection before triaging:

- reduce the reliance on memory (see Table 6.9)
- help better understanding the report (*Understanding the Ticket* and *Spread of Information*, see Tables 6.10 and 6.12)
- reduce time spent by the DM on similar or duplicate reports (*Prevention of Waste of Time* and *Reduced Time Wasted*, see Tables 6.10 and 6.13)
- clarify unclear tickets for the DM (*Unclear Tickets*, see Table 6.11)
- help to better intertwine an ITS and therefore improve the quality of its documentation feature (see Table 6.12)
- early presented recommendations of similar issues add most value (*Early Feedback* and *Pre-Submission Feedback*, see Table 6.13)

Regarding the *topic sentiment score*s for the topics covered in this section, there are many individual and unique opinions for each topic and so these scores indicate a rather neutral outcome. However, we did not include all notions from each topic. Instead, we took only specific ones into account and thus it makes more sense to consider the more detailed notion score of each notion. Nevertheless, the experts seem to be very uniform in their opinions regarding *Tool Assisted Similar Issue Detection* with $ts_{\textit{Tool Assisted Similar Issue Detection}} = 0.82$.

RQ4 Does tool assisted detection affect finding duplicate defect reports during the *scrub* meeting?

Results. To provide an answer for RQ4 we want to refer to the answer to RQ3 because the results presented there are also mostly valid for the results here. Therefore, we will mention the topics from RQ3 shortly if the results can be mapped directly. However,

we also want to highlight the differences to RQ3 which we found in the interviews with the experts.

In terms of *Triage Preparation* (see Table 6.9), the main difference lies in the DM's task in the scrub meeting as described in section 2.2. Since he is the moderator of the meeting, in this case, SBRDTs are more valuable for the other participants who actually triage the bug reports.

Apart from that, due to *Low Frequency* of preparation before the *scrub* meeting, automation with respect to SBRDTs should render itself useful, because it supports experienced triagers with lots of domain knowledge (see Table 6.13), but helps in onboarding triagers new to the defect management process as well when considering the improved linking in the ITS due to such tools.

Actually, one expert mentioned that the day after our interview was his last day in the *scrub* which corroborates the usefulness of SBRDTs and considering this research question, we believe it would diminish the need for *Triage Preparation* to some extent or more specifically reduce the time taken to prepare for the *scrub* meeting for new triagers, in particular.

As far as *Triaging Challenges* are concerned (see Table 6.10), we do not have much more to add other than SBRDTs being helpful in handling of some of these challenges and therefore, we refer to the results of RQ3 at this point.

The same applies to the topic *In-depth Issue Discussion* (see Table 6.11) with one exception. As already mentioned, we attended 15 *scrub* meetings where we could correlate in-depth discussions to resolutions as duplicate for the bug report under triage. This was not a notion obtained from the answers of the experts in the interviews, but we are convinced of SBRDTs contributing to the reduction of time spent of such reports so that triagers can focus on other tickets in the meeting and other tasks after the meeting.

For the topic *Linking Policy*, we do not have anything to add, so we refer to the answer to RQ3 (Table 6.12 and its description, in particular).

In our approach, we presented the recommendations of similar reports not only with the ticket id and summary, but also provided a relevance score along with the other information (see section 4.4, Table 4.5). While discussing the topic *Tool Assisted Similar Issue Detection* (see Table 6.13), one of the experts expressed his thoughts in a positive way about the *Relevance Score*. According to him, the score was very useful, in general. He continued by pointing out helpful recommendations in the case of a high relevance

score. Obviously, this is only one opinion and the tool was not in use for a sufficient period of time to reason about significant results, but it provides an idea of the outcome of a future evaluation. We believe that this has to do with the *Positive Disposition* towards SBRDTs in general and therefore, automatically providing recommendations of similar bug reports definitely adds value to a triage workflow.

Continuing the useful aspects of SBRDTs, we want to shortly pick up *Triaging Improvements* which we presented in "Observations of the *scrub* meeting" in section 6.1 (see Table 6.5). There are two notions which matter for the results of RQ4, one being *Duplication Detection* and the other one being *Better Prioritization*.

Duplication Detection is literally the motivation of our research regarding improvements of a triage workflow. Two of the experts mentioned this improvement directly. For the others, we are convinced that it would constitute an improvement as well, because all experts were well disposed towards SBRDTs (*Positive Disposition*, see Table 6.13). However, we did not push them to express their favour of *Duplication Detection* as benefit while discussing *Triaging Improvements*, because we wanted to acquire unbiased thoughts of the experts.

Better Prioritization is an improvement that can be inferred from the triaging challenges presented in Table 6.10. We already mentioned the possibility of SBRDTs to support to *Understand the Priority*, so we refer to the description of Table 6.10 at this point.

Summary. We would like to answer this research question with a *yes* without a doubt due to the conveniences described earlier and the favourable disposition of the experts, but the *scrub* meeting at Sky is highly optimized and moves very quickly even though there is room for improvement (see Table 6.5). According to the experts it is better to provide *Early Feedback* (see Table 6.13), so the answer is *yes* but the impact of SBRDTs during the *scrub* meeting is not that high compared to pre *scrub* recommendations. We still compiled a list of advantages of SBRDTs where we think triagers would draw benefits from:

- support triaging (see Table 6.13)
- reduce the time spent in general and on in-depth discussions (*Unclear Tickets* and *Reduced Time Wasted*, see Tables 6.11 and 6.13)
- indicate the relevance of a recommendation for a similar bug report by providing a relevance score (*Relevance Score*, see Table 6.13)

6.4 Generalizability of Defejavu

RQ5 How generalizable is the proposed duplicate detection tool to support bug triage processes in other domains?

Results. In order to obtain the notions on how defect triage is understood by the experts at Sky, we first asked them to describe their view of the *scrub* meeting (*Triage Process Description*).

Table 6.14: Sentiments of the topic *Triage Process Description*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Correct Assignment	+	+	+	+	+	1.00
Improve Ticket Progress	+	+			+	0.88
Meeting Scope		+	+	+	+	0.88
Quality of Report Check		+	+	+	+	0.88
$ts_{\textit{Triage Process Description}}$						0.91

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

When Table 6.14 is considered, it is clear that the experts try to correctly assign a BR to the proper development team which is one of the few notions where all experts agree (*Correct Assignment*). Additionally, they feel that the meeting's purpose is to check the quality of new BRs (*Quality of Report Check*), i.e. in case the reported bug is valid, it should be possible for a developer to provide a fix and this is only possible if the BR is clear and of high quality. Most of the experts even mentioned that defect triage is meant for ensuring and also improving ticket progress (*Improve Ticket Progress*). Without the *scrub*, tickets would tend to stall. To summarize this topic, they agree on the scope of the *scrub* meeting and this is why the *topic sentiment score* for this topic is very high ($ts_{\textit{Triage Process Description}} = 0.91$).

We used this topic (*Triage Process Description*) to form a baseline for the generalizability of Defejavu. However, we also wanted to learn how the experts see the *scrub* in comparision to defect triage outside of Sky, i.e., if they think it is a rather tailored or general process which is consolidated in Table 6.15 as *Triage Process Classification*. This topic turned out to be hard to grasp because most of the experts, although having

Table 6.15: Sentiments of the topic *Triage Process Classification*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Up to date	+	–			+	0.65
General	+	+	+		+	0.95
Unchanged		–		+	+	0.58
$ts_{\textit{Triage Process Classification}}$						0.72

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

plenty of experience, not only did not have the opportunity to participate in defect triage outside of Sky, but also they did not agree on the notions forming the topic which is reflected in the rather low *topic sentiment score* ($ts_{\textit{Triage Process Classification}} = 0.72$). Only two of the experts believe the process to be up to date with one expert disagreeing with this notion (*Up to date*). Interestingly, the same expert who does not believe the process to be up to date also thinks that the process did not remain unchanged which is surprising at first glance considering the DM clearly agreeing to the notion *Unchanged*. We believe that this is due to the shorter time of participation in the *scrub* meeting by the DM compared to the expert SIE2. In spite of the two notions mentioned previously resulting in a greater spread of opinions, the experts still agree on the *scrub* being rather *General* than tailored towards the needs of Sky.

Table 6.16: Sentiments of the topic *Perks of the Triage Process*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Short and Proper Duration	+		+	+		0.74
Effective	+	+	+	+	+	1.00
Broad Insights (Products)	+					0.62
Proficient Participants		+	+	+	+	0.88
$ts_{\textit{Perks of the Triage Process}}$						0.81

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

We already presented *Triaging Improvements* previously (Table 6.5), but we want to pick

it up again along with the new topic *Perks of the Triage Process* and its notions shown in Table 6.16 to reinforce the base for reasoning about the generalizability of Defejavu.

The experts consider the current form of defect triage and especially the *scrub* meeting to be very *Effective* mostly because of the presence of *Proficient Participants* who are able to quickly iterate over the outstanding BRs for each meeting (*Short and Proper Duration*). One notion (*Broad Insights (Products)*) which sticks out should actually be a prerequisite for defect triagers because they should have a high level overview over all software products to be able to assign a BR to a corresponding development team. We believe that this expert was referring to products from the perspective of a customer and outside the SDLC including defect management at Sky. In general, the experts are convinced of the current defect triage process and had to take a break to think about their answer when asked about *Triaging Improvements*. This is also visible by the sparse agreement (Table 6.5) on notions about this topic.

Where the experts mostly agree is *Increased Strictness*. In their view, the quality of BRs is essential to resolving a bug. Therefore, it is necessary to warrant that BRs contain as much correct information as possible. In this sense, they have to be more strict in demanding more or improved information from the reporters. At this point, we want to neglect the other notions from the topic *Triaging Improvements* because they are either too specific, too sparse or do not reflect the overall opinion of the group of experts with the only exceptions being *Duplication Detection* and *Better Prioritization* which we both covered in the answer to RQ4 in section 6.3.

Table 6.17: Sentiments of the topic *Rapid Topic Understanding*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Experience	+	+	+	+	+	1.00
$ts_{\textit{Rapid Topic Understanding}}$						1.00

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

Accordingly, defect management including the *scrub* meeting is a highly optimized process at Sky which seems to be hard to improve according to the experts who offer a lot of *Experience* leading to *Rapid Topic Understanding* (see Table 6.17), i.e., grasping the idea of a BR. However, next to *Increased Strictness*, *Duplication Detection* (topic *Triaging*

Improvements) is one of the few ways to support the experts in the task of defect triage. We want to revisit this notion for the answer to this research question.

Since we identified a way for improving defect management at Sky and the research of the past (see chapter 3) indicates that the problem of duplicate BRs remains partially unsolved, we are keen to make our contribution as general as possible.

Table 6.18: Sentiments of the topic *Software Tool Classification*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Productivity Boost	+	+	+	+	+	1.00
Positive User Experience	+		+	+		0.74
Minimal Set of Restrictions		+			+	0.76
High Accuracy			+		+	0.71
Ease of Use	+		+	+	+	0.88
Robustness				+		0.55
$ts_{\textit{Software Tool Classification}}$						0.77

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

Therefore, we asked the experts in the interviews to share their views on how a software tool should behave in general (topic *Software Tool Classification*, see Table 6.18) and brought their answers into relation with the results of the topic *Tool Assisted Similar Issue Detection* (see Table 6.13).

In general, the experts expect from a software tool to boost the productivity for a given task (*Productivity Boost*). They clearly denied restrictiveness of a tool for the most part (*Minimal Set of Restrictions*) where a specific process could restrict the feature set of a tool, i.e., to streamline the appearance of BRs which should contain information about "steps to reproduce", "expected behaviour", "actual behaviour", etc.

Positive User Experience can be seen as an overall requirement for any tool and is rather general because if a user is not satisfied or even frustrated with the experience of using a tool, productivity cannot be boosted and *Ease of Use* is not guaranteed. Interestingly, some of the experts mentioned *High Accuracy* and *Robustness* to be important, potentially because of their knowledge about the technical approach presented in this work where accurate and robust results are key to justified existence of such tools.

This ties into the answers given to questions related to the topic *Tool Assisted Similar Issue Detection* with the overlapping notion *Ease of Use*. When considering the notion *Relevance Score* and *Limited List of Most Relevant Tickets* and comparing it with how recommendations of similar BRs are presented (see Table 4.5), it is apparent that our approach matches the experts' idea of such tools.

In terms of *Simple Bug Tracking System Integration*, the experts suggested an improved way of the overall way of representation of top-*k* recommendations. They did not really like hijacking the comment feature of JIRA for the top-*k* lists. Additionally, an easy way of navigating back and forth between the query BR and the recommended similar BRs was missing in their opinion.

Summary. At its core, Defejavu is a tool for identifying similarities between a set of two arbitrary documents from a given dataset with the ability to present the results to a select group of users.

Based on the rather general way of defect triage at Sky and usage of REST APIs for extracting BRs and returning top-*k* recommendations of similar BRs, we believe that Defejavu is usable in other JIRA projects to support identifying duplicate BRs.

Additionally, we are convinced of more broad usages of Defejavu especially when users are interested in the similarities between a set of two documents as long as the underlying system used for tracking these documents is JIRA. However, since we did not built a JIRA bound plugin which cannot run standalone, our tool is easily extensible with an investment of a small effort. This is mainly due to the decoupled architecture of Defejavu (see Figure 4.1) relying on message passing using a message broker. Each of the four components is easy to configure, i.e., ETL, or even easy to replace with a custom implementation and each component relies on loosely coupled modules. This allows for quickly-to-establish support of systems other than JIRA.

Our conclusion regarding this research question is based on the following results:

- Visual presentation of recommendations matches the experts' idea in general (*Relevance Score*, *Limited List of Most Relevant Tickets*, see Tables 4.5 and 6.13).
- Defejavu was used to calculate *recall rate@k* for BRs from the JIRA projects JENKINS and Sky (see section 6.1).
- Message passing architecture allows for easy extensibility of Defejavu.

- In a well optimized defect management process, SBRDTs are one of the few ways for improving the process (*Duplication Detection* and *Short and Proper Duration*, see Tables 6.5 and 6.16).

6.5 Domain Knowledge as Underappreciated Asset

In 2005, Bowler [62] introduced the concept of the truck factor where the higher the number is the more people on a team have to be absent at the same time before consequences of the absence are impactful. In other words it means that if the truck factor is 1, only a single person has to be absent to cause issues within a project. Considering the motivation for this research question where we provided the example of MicroPython[3] the assumed truck factor would be close to 1. It is assumed because we are not involved in the project and we were not in touch with its maintainers.

RQ6 Are there any measures in place to cope with a total loss of domain knowledge of the defect *scrub* team?

Results. At the time of the interviews we were not aware of the term *truck factor* but we still wanted to learn about the measures at Sky to prevent loss of domain knowledge and what the experts think of a disastrous scenario where the whole *scrub* team was lost due to a catastrophic accident.

Table 6.19 shows the notions mentioned in the answers of the experts about the topic *Loss of Domain Knowledge*. It was initiated with an example where the experts should imagine an accident which would lead to the death of all defect *scrub* participants. We picked a dramatic example to ensure the interviewees were in the proper mindset to provide an answer as accurate as possible. Nearly all of them considered such a scenario to be tough (*Tough Scenario*) with the only exception being the SM. His answer was given based on the certainty of spread knowledge within the team he is in meaning that if he was gone, one of the other team members could replace his role in the defect *scrub*.

We have to point out that the opinion of the experts matched in terms of *No Disaster Strategy* which could lead to the conclusion of no measures at Sky to cope with the loss

[3] `https://github.com/micropython/micropython` (verified 2020-02-21)

Table 6.19: Sentiments of the topic *Loss of Domain Knowledge*.

Notion (c)	SIE1	SIE2	SIE3	SM	DM	Score (ns)
Tough Scenario	+	+	+	–	+	0.90
Onboarding Required	+	+		+		0.79
Introduction of Cheat Sheet	+					0.62
No Disaster Strategy	+	+	+	+	+	1.00
WIKIs	+	+				0.74
Spread/Duplication of Knowledge	+		+	+	+	0.88
$ts_{\textit{Loss of Domain Knowledge}}$						0.82

A plus (+) indicates approval or acceptance ($o_{c,e} = 1$) of the notion
A minus (–) indicates disapproval or rejection ($o_{c,e} = 0$) of the notion
A blank space ( ) indicates no mention ($o_{c,e} = 0.5$) of the notion

of the defect *scrub* team. However, the experts were certain about *Spread/Duplication of Knowledge* providing the possibility to replace any member on the *scrub*.

Nonetheless, they admitted to the need for onboarding of participants new to the defect *scrub* process at Sky (*Onboarding Required*) which is backed up by *WIKIs*, i.e. documentation about the defect management process stored in CONFLUENCE pages, and could be improved by providing a cheat sheet with all acronyms and terminology used in the defect *scrub* meeting (*Introduction of Cheat Sheet*).

Summary. Considering the responses of the experts, we answer this research question with *yes* if only a few people are absent and *no* if many people are absent.

Domain knowledge seems to be well spread across team members and in case of the absence of team members, colleagues with equal knowledge can be set in place (*Spread/Duplication of Knowledge*).

If a new colleague has to be trained to partake in the defect management process, the onboarding process is backed by documentation in the form of CONFLUENCE pages (*WIKIs*). However, if something disastrous happened where the whole team was lost, it would pose a *Tough Scenario* for Sky.

We believe that coping with such a scenario can be improved by introducing tools such as SBRDTs that help automating the defect management process. As far as Sky is

concerned, there are no measures in place to properly cope with a disaster (*No Disaster Strategy*).

6.6 Threats to Validity

In order to improve or better validate our results, we have to further work on the following threats to validity.

Threats to Internal Validity

Training a NLDNN should be improved by providing negative samples that consist of *nondup-dup* pairs where the *nondup* truly is a BR that is not linked to any other BR. In this work, we picked a *dup* BR and paired it with another *dup* BR. This approach might leak the relation between BRs into the set of negative samples.

For creating the relevant and irrelevant samples a similar issue was observed. We rely on the validity of the underlying dataset, i.e., the BRs from the ITS. However, we observed BRs being resolved as duplicate without any links to another BR. Again, this might leak the relation between two BRs into the set of negative samples.

Additionally, we considered all tickets from ITS for training the RankNet based model, i.e., negative samples could consist of a BR paired with a user story. It is possible that users mistakenly pick the wrong issue type and flag a report as a user story instead of a BR. We believe that this type of error can be neglected and thus only considering BRs would narrow down the amount of candidates for calculating the top-n lists. However, we do not have data to prove our claim.

The idea of non-linearity of document vectors (NLD) is only experimental and was not fully evaluated. However, the initial experiments show promising results and need to be further investigated.

Threats to External Validity

Only two datasets were used and only the one extracted from Sky was analyzed more deeply to better optimize and select DRFs.

Furthermore, the answers given by the experts might be biased because we only considered experts at Sky.

Threats to Construct Validity

The metric *topic sentiment score* used to rate the experts' answers might be too weak because often it is closer to a neutral ranking due to the spread of notions mentioned by the different experts.

It is important to note that the results in Figures 6.1, 6.2, 6.3, 6.4 and 6.5 only represent the *recall rate@k* where only all *ori* and *dup* BRs where available as candidate similar BRs for calculation of the top-*n* lists. The rest was not considered. Therefore, it can be assumed that *recall rate@k* will be lower when all BRs are available as candidates.

We use the terms *similar* and *duplicate* interchangeably as if they meant the same but actually detecting duplicates is by far a more complex problem than detecting similarities between documents. Unfortunately, best to our knowledge, previous work did not point out if they were trying to solve the problem of detecting duplicates or simply highlighting similarities between documents but calling it duplicate detection.

Threats to Conclusion Validity

We conducted a total of five qualitative interviews with experts from a single industry partner. Apart from that, Defejavu was not compared to previous work (chapter 3) in terms of *recall rate@k*. Therefore, our results do not have a statistically significant effect on the measured outcome.

7 Conclusion and Future Work

In this work, we present a recommender system for finding similar BRs based on a query BR featuring a novel approach regarding the calculation of similarities between a set of two document vectors. With the non linear distance (NLD), we propose the idea of nonlinearity with respect to similarity of document vectors. Our early results show that the *recall rate@k* based on NLD surpasses the one based on cosine similarity (cosine distance) for $k < 22$. If we combine multiple Word2vec models, i.e., SO200 and GN300, we observe even better performance where CD is outperformed for $k > 4$. According to the experts from our interviews in the realm of issue tracking systems (ITSs), k should not exceed 4, i.e., no more than four items should be recommended based on a query bug report (BR) to prevent too much effort spent on inspecting the recommendations. At *recall rate@4*, we note percentage changes of $+67\%$ for the Sky dataset and $+29\%$ for the JENKINS dataset compared to CD using our two pass ranking approach.

From our interviews, we learned that similar issue detection is considered to be useful and supportive for defect management in a software development life cycle (SDLC). Recommendations should be provided as early as possible to prevent waste of effort primarily with respect to *fixers*, i.e., developers, but also regarding *triagers* and *reporters*. Therefore, we conclude that similar bug report detection tools (SBRDTs) add the most value when lists of similar BRs are associated with a new BR close to or at creation time. Such tools can be useful in the context of a triage meeting, i.e., the *scrub*, in a closed source environment but with the triage process we saw at Sky and also according to the experts the most benefits are gained through early recommendation.

In the future, we want to evaluate the NLD measure with more and improved experiments to draw a definite conclusion about the nonlinear similarity property of document vectors. Additionally, Defejavu should accept feedback to further improve its recommendations. On dynamic rank feature (DRF) level, it could be interesting to investigate an improved encoding of the feature *reporter* where duplicate BR rates are incorporated in the similarity value. For the Sky dataset, we observed better performance of NLD based on the general purpose model GN300. This model covers a

broader area of topics compared to SO200. Therefore, training a custom Word2vec model to further improve *recall rate@k* will potentially turn out to add even more value to Defejavu.

List of Abbreviations

ANSIBLE	Red Hat Ansible[1]
API	application programming interface
BR	bug report
CD	cosine distance
CONFLUENCE	Atlassian Confluence[2]
DM	Defect Manager
DOCKER	Docker[3]
DR	defect report
DRF	dynamic rank feature
ETL	*extract, transform, load*
GENSIM	Gensim Python library[4]
GN300	Google News 300
IDF	inverse document frequency
IR	information retrieval
ITS	issue tracking system
JENKINS	Jenkins CI[5]
JIRA	Atlassian Jira[6]
JQL	JIRA Query Language[7]
JSON	JavaScript Object Notation[8]
KERAS	keras[9]

[1] `https://www.ansible.com/` (verified 2020-04-05)
[2] `https://www.atlassian.com/software/confluence` (verified 2020-03-03)
[3] `https://www.docker.com/` (verified 2020-04-05)
[4] `https://radimrehurek.com/gensim/` (verified 2020-03-10)
[5] `https://issues.jenkins-ci.org/projects/JENKINS` (verified 2020-02-24)
[6] `https://www.atlassian.com/software/jira` (verified 2020-02-04)
[7] `https://www.atlassian.com/blog/jira-software/jql-the-most-flexible-way-to-search-jira-14` (verified 2020-04-02)
[8] `https://www.json.org/json-en.html` (verified 2020-04-05)
[9] `https://keras.io/` (verified 2020-04-05)

LUCENE	Apache Lucene[10]
mongoDB	mongoDB[11]
NLD	non linear distance
NLDNN	NLD neural network
NLP	natural language processing
NP	numpy[12]
PD	pandas[13]
PRED	*prediction*
PREP	*preprocessing*
PUB	*publish*
PY	Python 3.7[14]
REST	representational state transfer
RMQ	RabbitMQ[15]
SCRUB	triage
SDLC	software development life cycle
SBRDT	similar bug report detection tool
SIE	System Integration Engineer
SM	Scrum Master
SO200	Stack Overflow 200
SRF	static rank feature
SV	similarity vector
SYNCPIPES	SyncPipes[16]
TF	term frequency
TS	TypeScript[17]
TYPE I	TYPE I duplicate bug report
TYPE II	TYPE II duplicate bug report
VBR	vectorized bug report

[10] `https://lucene.apache.org/` (verified 2020-04-05)
[11] `https://www.mongodb.com/` (verified 2020-04-05)
[12] `https://numpy.org/` (verified 2020-04-05)
[13] `https://pandas.pydata.org/` (verified 2020-04-05)
[14] `https://www.python.org/` (verified 2020-04-05)
[15] `https://www.rabbitmq.com/` (verified 2020-04-05)
[16] `https://wwwmatthes.in.tum.de/pages/2gh0u9d1afap/SyncPipes` (verified 2020-03-08)
[17] `https://www.typescriptlang.org/` (verified 2020-04-05)

List of Figures

List of Tables

Bibliography

[1] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. "A discriminative model approach for accurate duplicate bug report retrieval." In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 45–54.

[2] J. Anvik, L. Hiew, and G. C. Murphy. "Coping with an open bug repository." In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. 2005, pp. 35–39.

[3] N. Jalbert and W. Weimer. "Automated duplicate detection for bug tracking systems." In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE. 2008, pp. 52–61.

[4] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[5] ISO/IEC/IEEE 12207:2017. *Systems and software engineering — Software life cycle processes*. Standard. Geneva, CH: International Organization for Standardization, Nov. 2017.

[6] J. MacKay. *Software Development Process: How to Pick The Process That's Right For You*. Oct. 2019. URL: `https://plan.io/blog/software-development-process/` (visited on 2020-02-13).

[7] *SDLC - Overview*. URL: `https://www.tutorialspoint.com/sdlc/sdlc_overview.htm` (visited on 2020-02-13).

[8] N. B. Ruparelia. "Software development lifecycle models." In: *ACM SIGSOFT Software Engineering Notes* 35.3 (2010), pp. 8–13.

[9] H. Benington. "Production of large computer programs." In: *Proceedings of the ONR Symposium on Advanced Programming Methods for Digital Computers*. June 1956, pp. 15–27.

[10] W. W. Royce. "Managing the development of large software systems." In: *Proceedings of IEEE Wescon 26*. Aug. 1970, pp. 1–9.

[11] I. Sommerville. "Software process models." In: *ACM computing surveys (CSUR)* 28.1 (1996), pp. 269–271.

[12] G. Gladden. "Stop the life-cycle, I want to get off." In: *ACM SIGSOFT Software Engineering Notes* 7.2 (1982), pp. 35–39.

[13] D. D. McCracken and M. A. Jackson. "Life cycle concept considered harmful." In: *ACM SIGSOFT Software Engineering Notes* 7.2 (1982), pp. 29–32.

[14] K. M. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. J. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. "Manifesto for Agile Software Development." In: 2013.

[15] *SDLC - Agile Model*. URL: `https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm` (visited on 2020-04-05).

[16] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002, pp. 20–22.

[17] V. Kannan, S. Jhajharia, and S. Verma. "Agile vs waterfall: A Comparative Analysis." In: *International Journal of Science, Engineering and Technology Research (IJSETR)* 3.10 (2014), pp. 2680–2686.

[18] S. Balaji and M. S. Murugaiyan. "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC." In: *International Journal of Information Technology and Business Management* 2.1 (2012), pp. 26–30.

[19] *SDLC - V-Model*. URL: `https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm` (visited on 2020-02-14).

[20] A. X. Fellmeth and M. Horwitz. *Guide to Latin in international law*. Oxford University Press, 2009.

[21] M. Ohira, A. E. Hassan, N. Osawa, and K.-i. Matsumoto. "The impact of bug management patterns on bug fixing: A case study of Eclipse projects." In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2012, pp. 264–273.

[22] L. Hiew. "Assisted detection of duplicate bug reports." PhD thesis. University of British Columbia, 2006.

[23] J. Kuruvilla. *JIRA Development Cookbook*. Packt Publishing Ltd, 2016.

[24] T. Dawson. *'find similar' feature to prevent creating duplicate issue*. May 2003. URL: `https://jira.atlassian.com/browse/JRASERVER-1633` (visited on 2020-02-03).

[25] G. Murphy and D. Cubranic. "Automatic bug triage using text categorization." In: *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer. 2004.

[26] S. R. Gunn et al. "Support vector machines for classification and regression." In: *ISIS technical report* 14.1 (1998), pp. 5–16.

[27] R. B. Segal and J. O. Kephart. "MailCat: An intelligent assistant for organizing e-mail." In: *Proceedings of the third annual conference on Autonomous Agents*. 1999, pp. 276–282.

[28] M. F. Porter et al. "An algorithm for suffix stripping." In: *Program* 14.3 (1980), pp. 130–137.

[29] C. Buckley, J. Walz, C. Cardie, S. Mardis, M. Mitra, D. Pierce, and K. Wagstaff. "The smart/empire tipster ir system." In: *TIPSTER TEXT PROGRAM PHASE III: Proceedings of a Workshop held at Baltimore, Maryland, October 13-15, 1998*. 1998, pp. 107–121.

[30] P. Hooimeijer and W. Weimer. "Modeling bug report quality." In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 34–43.

[31] P. Runeson, M. Alexandersson, and O. Nyholm. "Detection of duplicate defect reports using natural language processing." In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 499–510.

[32] N. Mishra, R. Schreiber, I. Stanton, and R. E. Tarjan. "Clustering social networks." In: *International Workshop on Algorithms and Models for the Web-Graph*. Springer. 2007, pp. 56–67.

[33] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. "What makes a good bug report?" In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 2008, pp. 308–318.

[34] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. "Towards more accurate retrieval of duplicate bug reports." In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE. 2011, pp. 253–262.

[35] S. Robertson, H. Zaragoza, et al. "The probabilistic relevance framework: BM25 and beyond." In: *Foundations and Trends® in Information Retrieval* 3.4 (2009), pp. 333–389.

[36] M. Taylor, H. Zaragoza, N. Craswell, S. Robertson, and C. Burges. "Optimisation methods for ranking functions with multiple parameters." In: *Proceedings of the 15th ACM international conference on Information and knowledge management*. 2006, pp. 585–593.

[37] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. "Learning to rank using gradient descent." In: *Proceedings of the 22nd international conference on Machine learning*. 2005, pp. 89–96.

[38] A. Alipour, A. Hindle, and E. Stroulia. "A contextual approach towards more accurate duplicate bug report detection." In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 183–192.

[39] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun. "Combining word embedding with information retrieval to recommend similar bug reports." In: *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*. IEEE. 2016, pp. 127–137.

[40] F. Koch. *REST-based data integration services for software engineering domain*. 2016.

[41] G. Salton, A. Wong, and C.-S. Yang. "A vector space model for automatic indexing." In: *Communications of the ACM* 18.11 (1975), pp. 613–620.

[42] J. Beel, B. Gipp, S. Langer, and C. Breitinger. "Research-paper recommender systems : a literature survey." In: *International Journal on Digital Libraries* 17.4 (2016), pp. 305–338. ISSN: 1432-5012. DOI: 10.1007/s00799-015-0156-0.

[43] C. D. Manning, P. Raghavan, and H. Schütze. "Scoring, term weighting, and the vector space model." In: *Introduction to Information Retrieval*. Cambridge University Press, 2008, pp. 100–123.

[44] T. Mikolov, K. Chen, G. S. Corrado, and J. A. Dean. *Computing numeric representations of words in a high-dimensional space*. US Patent 9,037,464. May 2015.

[45] T. Mikolov, K. Chen, G. Corrado, and J. Dean. "Efficient estimation of word representations in vector space." In: *arXiv preprint arXiv:1301.3781* (2013).

[46] *Word2vec*. July 2013. URL: https://code.google.com/archive/p/word2vec/ (visited on 2020-03-11).

[47] V. Efstathiou, C. Chatzilenas, and D. Spinellis. "Word embeddings for the software engineering domain." In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 38–41.

[48] A. Khatua, A. Khatua, and E. Cambria. "A tale of two epidemics: Contextual Word2Vec for classifying twitter streams during outbreaks." In: *Information Processing & Management* 56.1 (2019), pp. 247–257.

[49] M. Claesen and B. De Moor. "Hyperparameter search in machine learning." In: *arXiv preprint arXiv:1502.02127* (2015).

[50] S. L. Harris and D. M. Harris. "3 - Sequential Logic Design." In: *Digital Design and Computer Architecture*. Ed. by S. L. Harris and D. M. Harris. Boston: Morgan Kaufmann, 2016, pp. 108–171. ISBN: 978-0-12-800056-4. DOI: https://doi.org/10.1016/B978-0-12-800056-4.00003-0.

[51] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. "Distributed representations of words and phrases and their compositionality." In: *Advances in neural information processing systems*. 2013, pp. 3111–3119.

[52] B. C. Csáji et al. "Approximation with artificial neural networks." In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24.48 (2001), p. 7.

[53] A. Kolmogorov. "On the representation of continuous functions of several variables by superpositions of continuous functions of a smaller number of variables." In: *Proceedings of the USSR Academy of Sciences*. Vol. 108. 1956, pp. 179–182. English translation: *American Mathematical Society Translations: Series 2*. Vol. 17. 1961, pp. 369–373.

[54] V. Arnold. "On functions of three variables." In: *Proceedings of the USSR Academy of Sciences*. Vol. 114. 1957, pp. 679–681. English translation: *American Mathematical Society Translations: Series 2*. Vol. 28. 1963, pp. 51–54.

[55] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function." In: *Neural Networks* 6.6 (1993), pp. 861–867.

[56] K. Hornik. "Approximation capabilities of multilayer feedforward networks." In: *Neural Networks* 4.2 (1991), pp. 251–257.

[57] P. Jaccard. "Étude comparative de la distribution florale dans une portion des Alpes et des Jura." In: *Bulletin de la Société Vaudoise des Sciences Naturelles* 37 (1901), pp. 547–579.

[58] A. Hald. *History of Probability and Statistics and Their Applications before 1750*. John Wiley & Sons, Inc., Jan. 1990.

[59] M. A. Alcorn. *airalcorn2/RankNet*. Jan. 2019. URL: https://github.com/airalcorn2/RankNet (visited on 2020-03-19).

[60] A. Egg. *eggie5/RankNet*. Jan. 2019. URL: https://github.com/eggie5/RankNet (visited on 2020-03-19).

[61] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. "Duplicate bug reports considered harmful... really?" In: *2008 IEEE International Conference on Software Maintenance*. IEEE. 2008, pp. 337–345.

[62] M. Bowler. *Truck Factor*. May 2005. URL: http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/ (visited on 2020-04-02).

[63] *American Mathematical Society Translations: Series 2*. Vol. 17. 1961, pp. 369–373.

[64] *American Mathematical Society Translations: Series 2*. Vol. 28. 1963, pp. 51–54.