# INAUGURAL-DISSERTATION

submitted

to the

**Combined Faculty of Mathematics, Engineering, and Natural Sciences**

of

**Ruprecht-Karls-University**

**Heidelberg**

**for the Degree of Doctor of Natural Sciences**


Put forward by

**Michael Anders M.Sc.**

Born in Charlotte USA


Oral examination: .....

# Automating Feedback Analysis to Support Requirements Relation and Usage Understanding


Supervisors: Prof. Dr. Barbara Paech Heidelberg University

Prof. Dr. Kurt Schneider Hannover University

# Abstract

**Context:** Software development is an iterative process requiring continuous adaptation to user needs. However, a gap often exists between developers' assumptions and users' actual expectations for the software. While direct user participation is valuable for bridging this gap, practical constraints often make it difficult. Online user feedback provides an alternative source for insights but is typically unstructured and lacks context. To bridge the gap between developers and users through online feedback, two key challenges need to be tackled: (1) identifying which functionalities users discuss in their feedback and (2) understanding how users interact with these functionalities. Moreover, manually analyzing large volumes of feedback is time-consuming, highlighting the need for automation and tool support.

**Objective:** The goal of this thesis is to introduce two approaches to tackle the above mentioned challenges. One approach, handling challenge (1), facilitates the relation of feedback and existing requirements of a software. The functionalities of a software are documented in its requirements. By relating feedback directly to the requirements, developers are able to identify the discussed functionalities of the software. The other approach, handling challenge (2), allows the analysis of users' needs and expectations by extracting the usage information in their feedback. Usage comprises both the real-life actions of the users as well as the interactions of the user with the software. This usage information is captured through the application of the TORE framework on the user feedback and allows developers to gain a better understanding of how the users interact with the software's functionalities. For both of these approaches, the relation and the usage information analysis, this thesis offers machine learning classifiers and tool support to reduce the manual labour required for the analysis.

**Methods:** This thesis follows the Design Science approach consisting of solution investigation, treatment design and treatment validation. The solution investigation is conducted via two systematic mapping studies in order to identify existing machine learning classifiers and evaluate their applicability to feedback requirements relation and usage information classification. The treatment design contains goals that match the two approaches presented in this thesis: The design and implementation of an approach and accompanying automatic classifiers to relate feedback to existing requirements and to identify the usage information contained in feedback. The implemented classifiers are also evaluated on multiple manually created datasets to identify the best-performing ones. Additionally, a software prototype is presented as part of the treatment design, which offers tool support for the developed approaches. The treatment validation evaluates the developed classifiers in the context of a hypothetical deployment scenario in a company.

**Contributions:** The main contributions of this thesis are the presented approaches for feedback requirements relation and usage information analysis as well as the classifiers for the automation of these approaches and their evaluation. Multiple manually created datasets are also provided to train and evaluate the presented classifiers. Furthermore, two mapping studies are included, which capture the current state of research towards the relation of software artifacts and detailed user feedback classification. Additionally, a prototype (Feed.UVL) is created to provide tool support for the developed approaches. A Jira plugin is also provided to integrate the tool support for the approaches into existing development workflows.

# Zusammenfassung

**Kontext:** Softwareentwicklung ist ein iterativer Prozess, der eine kontinuierliche Anpassung an die Bedürfnisse der NutzerInnen erfordert. Häufig besteht jedoch eine Disparität zwischen den Annahmen der EntwicklerInnen und den tatsächlichen Erwartungen der NutzerInnen an die Software. Die direkte Beteiligung der NutzerInnen ist zwar wertvoll, um diese Disparität zu überbrücken, wird aber durch praktische Beschränkungen oft erschwert. Online-Feedback bietet eine alternative Quelle für Erkenntnisse, ist aber in der Regel unstrukturiert und hat keinen Kontext. Um die Unterschiede zwischen EntwicklerInnen und NutzerInnen durch Online-Feedback zu überbrücken, müssen zwei zentrale Herausforderungen bewältigt werden: (1) Identifizierung der Funktionen, die NutzerInnen in ihrem Feedback diskutieren, und (2) Verständnis dafür, wie NutzerInnen mit diesen Funktionen interagieren. Darüber hinaus ist die manuelle Analyse großer Mengen von Feedback zeitaufwändig, was den Bedarf an Automatisierung und Toolunterstützung unterstreicht.

**Zielsetzung:** Das Ziel dieser Dissertation ist es, zwei Ansätze vorzustellen, um die oben genannten Herausforderungen zu bewältigen. Ein Ansatz, der Herausforderung (1) behandelt, erleichtert den Bezug zwischen Feedback und bestehenden Anforderungen einer Software. Die Funktionalitäten einer Software sind in ihren Anforderungen dokumentiert. Indem das Feedback direkt mit den Anforderungen in Verbindung gebracht wird, können EntwicklerInnen, die diskutierten Funktionalitäten der Software identifizieren. Der andere Ansatz, der Herausforderung (2) behandelt, ermöglicht die Analyse der Erwartungen der NutzerInnen durch die Extraktion der Nutzungsinformationen in ihrem Feedback. Die Nutzungsinformationen umfassen sowohl die realen Handlungen der NutzerInnen als auch die Interaktionen der NutzerInnen mit der Software. Diese Nutzungsinformationen werden durch die Anwendung da TORE-Frameworks erfasst und ermöglichen es den EntwicklerInnen, ein besseres Verständnis dafür zu erlangen, wie die NutzerInnen mit den Funktionen der Software interagieren. Für beide Ansätze, die Bezugsherstellung und die Nutzungsinformationsanalyse, bietet diese Arbeit Klassifikatoren und Werkzeugunterstützung, um den manuellen Arbeitsaufwand für die Analyse zu reduzieren.

**Methode:** Diese Arbeit folgt der Design-Science-Methode, bestehend aus Lösungsuntersuchung, Lösungsentwurf und Lösungsvalidierung. Die Lösungsuntersuchung wird mittels zweier systematischer Mapping-Studien durchgeführt, um bestehende Klassifikatoren zu identifizieren und ihre Anwendbarkeit auf die Bezugsherstellung und die Nutzungsinformationsklassifikation zu bewerten. Der Lösungsentwurf enthält Ziele, die mit den beiden in dieser Arbeit vorgestellten Ansätzen übereinstimmen: Der Entwurf und die Implementierung eines Ansatzes und begleitender automatischer Klassifikatoren, um Feedback mit bestehenden Anforderungen in Beziehung zu setzen und die im Feedback enthaltenen Nutzungsinformationen zu identifizieren. Die implementierten Klassifikatoren werden außerdem an mehreren manuell erstellten Datensätzen evaluiert, um die am besten funktionierenden Klassifikatioren zu identifizieren. Zusätzlich wird ein Software-Prototyp als Teil des Lösungsentwurfs vorgestellt, der Werkzeugunterstützung für die entwickelten Ansätze bietet. Die Lösungsvalidierung evaluiert die entwickelten Klassifikatoren im Rahmen eines hypothetischen Einsatzszenarios in einem Unternehmen.

**Beiträge:** Die Hauptbeiträge dieser Arbeit sind die vorgestellten Ansätze zur Bezugsherstellung und Nutzungsinformationsanalyse sowie die Klassifikatoren zur Automatisierung dieser Ansätze und deren Bewertung. Mehrere manuell erstellte Datensätze werden ebenfalls bereitgestellt, um die vorgestellten Klassifikatoren zu trainieren und zu evaluieren. Darüber hinaus sind zwei Mapping-Studien enthalten, die den aktuellen Stand der Forschung zur Bezugsherstellung von Software-Artefakten und zur detaillierten Klassifikation von Feedback darstellen. Zusätzlich wird ein Prototyp (Feed.UVL) erstellt, um die entwickelten Ansätze zu unterstützen. Ein Jira-Plugin wird ebenfalls bereitgestellt, um die Werkzeugunterstützung für die Ansätze in bestehende Entwicklungs-Arbeitsabläufe zu integrieren.

# Acknowledgements

To begin with, I would like to thank my supervisor, Professor Barbara Paech, not only for supervising my bachelor's and master's thesis but for offering me the opportunity to pursue my PhD in the first place and for providing me with guidance throughout the last 5 years. Her hard work for our research group and in giving me continuous feedback on my work and providing me with ideas, direction and advice was invaluable. I would also like to thank her for giving me the opportunity to gain experience in teaching others through numerous lectures and seminars. I hope I will be able to apply what I have learned from her in my future endeavours.

I would also like to sincerely thank Professor Kurt Schneider for being the second supervisor of my dissertation and for his hard work on the UVL project, which set the cornerstone for my dissertation. Together, Professor Paech and Professor Schneider allowed me and the others working on the project the chance to not only explore a highly relevant and interesting topic but also learn more about scientific research than I would have otherwise ever been able to. On that note, I would also like to thank the other PhD candidates working on the project, namely Martin Obaidi and Alexander Specht. Thank you for all your work on UVL, especially the sometimes cumbersome manual coding. The work you did contributed significantly to my own research and I wouldn't have been able to do it without your help.

I would like to thank Stephanie for all her support and everything she did to keep our group running smoothly. Without her, I would never have been able to navigate the jungle of paperwork and bureaucracy that working at a university can entail. My thanks also go out to Willi for all his technical support regarding Feed.UVL, Jira and anything else that decided not to work on any random day of the week. I would also like to thank my former colleagues for all the advice and support they have given me. Thank you, Anja, for allowing me to share your office, showing the ropes and being there whenever I had any questions. Thank you, Leon, not only for your help on various lectures but also for our conversations in the office over the last five years. Thank you, Astrid, for all the work you did on ISW, which helped me significantly when it was my time to supervise the lecture.

I thank all the talented students whom I had the pleasure of supervising during their Bachelor's and Master's Thesis (Table 1) as well as any practicals I was able to oversee: Johannes Daub, Jakob Weichselbaumer, Eileen Dickson, Johannes Wenzel, Robin Alter, Lukas Bockstaller, Benjamin Tuna, Anh Tu Duong Nguyen, Daniel Knorr, Tim Waibel, Shreeharsh Gudibandi and Vasu Thakur. Your hard work provided invaluable contributions to my dissertation, and I hope I was able to adequately guide you through your thesis and instil in you a lasting interest in software engineering.

I sincerely thank my family and friends not only for their support over the last five years but throughout my entire life. First and foremost, I would like to thank my parents for everything they have done for me and for their unconditional support. I would not be where I am today without them. Thank you for all the opportunities you have given me in life, for all the weekend retreats away from work you've offered me and for always being there when I needed you. I'd also like to thank my brother Sebastian, his wife Sarah and both of my nephews for being part of my support network and for giving me a chance to get away from it all whenever I needed it. Special thanks go out to my best friend Patrick, whom I have known since kindergarten and have probably spent more time with in my life than anyone else. Thanks for all the laughs, late-night gaming, and discussion sessions, for always having an open ear, and for providing much-needed distractions. In that regard, I'd also like to thank Gerrit, Constantin and Marsi for the countless discussions, laughs, distractions, and all-around weirdness most days of the week. Last but not least, even though she will never read this, I must also thank my cat, Lumi, for always greeting me at the door whenever I came home and for having to listen to my complaints whenever I had a rough day.

Table 1.: Bachelor and Master Theses Contributing to the Dissertation

| **Master Theses** |
| --- |
| J. Daub (2021), "Tool Support for the Automatic Analysis of Natural Language User Statements". Master Thesis, Heidelberg University |
| E. Dickson (2022), "Utilization of Deep Learning for the Automatic Classification of Software Aspects". Master Thesis, Heidelberg University |
| J. Wenzel (2022), "Applying Machine Learning to the Automatic Classification of Software Aspects". Master Thesis, Heidelberg University |
| L. Bockstaller (2023), "Multi-Stage Software Aspect Classification". Master Thesis, Heidelberg University |
| B. Tuna (2024), "Automatically Relating Feedback and Existing Requirements". Master Thesis, Heidelberg University |
| D. Knorr (2024), "Using Pre-Processing to Improve the Extraction of Usage Information from Online Feedback". Master Thesis, Heidelberg University |
| T. Waibel (2024), "Support for Practitioners in Relating Feedback to Requirements". Master Thesis, Heidelberg University |
| V. Thakur (2024), "Improving the Automatic Relation of Feedback and Requirements". Master Thesis, Heidelberg University |
| **Bachelor Theses** |
| J. Weichselbaumer (2021), "Implementing toolchain support for annotation of natural language datasets". Bachelor Thesis, Heidelberg University |
| R. Alter (2022), "Visualization of Natural Language Annotation Models". Bachelor Thesis, Heidelberg University |
| A. Ngyuen (2024), "Relating Feedback and Existing Requirements using ChatGPT". Bachelor Thesis, Heidelberg University |

# Contents

**Abstract** i

**Zusammenfassung** iii

**Acknowledgements** v

**List of Acronyms** xi

**I. Preliminaries** 1

**1. Introduction** 2
1.1. Motivation & Problem . . . 2
1.2. Solution Idea . . . 4
1.3. Research Methodology . . . 4
1.3.1. Design Science Methodology . . . 4
1.3.2. Research Goals & Design Cycle . . . 5
1.4. Contributions . . . 8
1.5. Dissertation Structure . . . 9
1.6. Previous Publications . . . 9

**2. Fundamentals** 11
2.1. Feedback Analysis . . . 11
2.1.1. Analysis Goals . . . 12
2.1.2. Statement, Sentence and Word Analysis . . . 12
2.1.3. Feedback Sources . . . 13
2.2. Usage Information Classification Framework TORE . . . 15
2.2.1. Original TORE Framework . . . 15
2.2.2. Adapting TORE for Feedback Analysis . . . 18
2.2.3. Alternative Models . . . 19
2.3. Machine Learning Concepts and Classifiers . . . 20
2.3.1. Machine Learning . . . 20
2.3.2. Deep Learning . . . 21
2.3.3. Large Language Models . . . 22
2.3.4. Stanford Named Entitiy Recognizer (SNER) . . . 22
2.3.5. Bidirectional Long Short-Term Memory (Bi-LSTM) . . . 23
2.3.6. Bidirectional Encoder Representations from Transformers (BERT) . . . 24
2.3.7. GPT & Llama . . . 26

2.3.8. Classifier Performance Metrics . . . 26
2.4. Development Tools . . . 28
2.5. Training & Evaluation Datasets . . . 29
2.5.1. SmartAge Project . . . 29
2.5.2. Feedback Requirements Datasets . . . 29
2.5.3. Usage Information Datasets . . . 32

**II. Solution Investigation** **37**

**3. State of the Art - Software Artifact Relation** **38**
3.1. Study Design . . . 38
3.1.1. Research Questions . . . 38
3.1.2. Selection Procedure . . . 39
3.2. Overview of Publications . . . 39
3.3. Threats to Validity . . . 39
3.4. Conclusion . . . 41

**4. State of the Art - Fine-Grained Feedback Classification** **43**
4.1. Study Design . . . 43
4.1.1. Research Questions . . . 43
4.1.2. Selection Procedure . . . 44
4.2. Results and Comparison . . . 47
4.2.1. Overview of Publications . . . 47
4.2.2. Synthesis . . . 49
4.3. Threats to Validity . . . 57
4.4. Conclusion . . . 58

**III. Treatment Design** **61**

**5. Feedback Requirements Relations** **62**
5.1. FeReRe Approach . . . 62
5.1.1. Overview . . . 62
5.1.2. Use Cases . . . 63
5.2. FeReRe Classifier . . . 64
5.2.1. Research Questions . . . 64
5.2.2. Experiment Configuration . . . 65
5.2.3. Prompt Engineering . . . 66
5.2.4. Results . . . 66
5.2.5. Discussion . . . 69
5.2.6. Threats to Validity . . . 70
5.3. Conclusion . . . 71

**6. Usage Information Classification** **72**
6.1. UIC Approach . . . 72
6.1.1. Overview . . . 72
6.1.2. Granularities . . . 73
6.1.3. Use Cases . . . 74
6.1.4. Approach Application . . . 74

6.2. UIC Classifier . . . 76
6.2.1. Research Questions . . . 77
6.2.2. Experiment Configuration . . . 78
6.2.3. Prompt Engineering . . . 81
6.2.4. Results . . . 83
6.2.5. Discussion . . . 90
6.2.6. Threats To Validity . . . 92
6.3. UIC Conclusions . . . 93

**7. Feed.UVL** **95**
7.1. Feed.UVL Architecture . . . 95
7.1.1. Foundation . . . 95
7.1.2. Microservice Architecture . . . 96
7.1.3. Deployment Architecture . . . 98
7.1.4. Functionality Overview . . . 99
7.2. Feed.UVL Functionalities . . . 101
7.2.1. Feedback Collection . . . 102
7.2.2. Feedback Management . . . 107
7.2.3. Manual Feedback Analysis . . . 110
7.2.4. Automatic Feedback Analysis . . . 118
7.2.5. Feed.UVL Dashboard . . . 123
7.2.6. Jira Plugin . . . 128
7.3. Conclusion . . . 131

**IV. Treatment Validation** **133**

**8. Feedback Requirements Relation Evaluation** **134**
8.1. Evaluation Methodology . . . 134
8.2. Evaluation Scenario . . . 135
8.2.1. Scenario . . . 135
8.2.2. Time Efficiency . . . 136
8.2.3. Risk of Imperfect Classification . . . 136
8.2.4. FeReRe Deployment . . . 138
8.2.5. Maintainability . . . 139
8.3. Conclusion . . . 139

**9. Usage Information Classification Evaluation** **140**
9.1. Evaluation Methodology . . . 140
9.2. Evaluation Scenario . . . 140
9.2.1. Scenario . . . 141
9.2.2. Time Efficiency . . . 141
9.2.3. Risk of Imperfect Classification . . . 142
9.2.4. UIC Deployment . . . 143
9.2.5. Maintainability . . . 144
9.3. Conclusion . . . 144

**V. Conclusion** **146**

**10. Summary** **147**

**11. Future Work** **150**

**VI. Appendix** **153**

**A. Digital Appendix for Tools and Data** **154**

**B. Supplementary Material for Solution Investigation** **155**
B.1. Fine-Grained Feedback Classification . . . . . 155

**C. Supplementary Material for Treatment Design** **157**
C.1. FeReRe . . . . . 157
C.1.1. FeRere Prompts . . . . . 157
C.2. Feed.UVL . . . . . 159
C.2.1. Domain Data . . . . . 159
C.2.2. UI-Structure . . . . . 159

**Bibliography** **169**

**List of Figures** **170**

**List of Tables** **172**

# List of Acronyms

**TORE** Task-oriented Requirements Engineering . . . 11
**GUI** graphical user interface . . . 16
**UI** user interface . . . 16
**ML** machine learning . . . 20
**DL** deep learning . . . 20
**LLM** large language model . . . 20
**NLP** natural language processing . . . 20
**SNER** Stanford Named Entity Recognizer . . . 22
**NER** named entity recognition . . . 22
**CRF** conditional random fields . . . 23
**Bi-LSTM** Bidirectional long short-term memory . . . 23
**LSTM** long short-term memory . . . 23
**RNN** recurrent neural network . . . 23
**BERT** Bidirectional Encoder Representations from Transformers . . . 24
**TP** True Positives . . . 27
**FP** False Positives . . . 27
**FN** False Negatives . . . 27
**TN** True Negatives . . . 27
**ITS** issue tracking system . . . 28
**NFR** non-functional requirements . . . 50
**FeReRe** feedback requirements relation . . . 4
**UIC** usage information classification . . . 4
**DSM** design science methodology . . . 4

# Part I.

# Preliminaries

Chapter 1

# Introduction

This chapter introduces the motivation and associated problems in Section 1.1 as well as the solution idea for the work presented in this dissertation in Section 1.2. Section 1.3 describes the applied research methodology. Section 1.4 lists the contributions of this work. Section 1.5 gives an overview of the structure of this dissertation. Lastly, Section 1.6 lists previous publications.

## 1.1. Motivation & Problem

Software development is an iterative process that requires continuous improvement to meet user expectations and remain competitive (Bosch, 2014). As software evolves, developers must adapt to changing user needs and preferences. However, understanding these needs is not always straightforward. One significant challenge is the difference in background knowledge and experience between developers and users (Paech and Schneider, 2020). Technically inclined developers often have a different perspective on how a system should work compared to the users who interact with the software in their day-to-day activities. This gap in understanding can result in discrepancies between what developers assume users need and what users actually expect or require from the software.

To bridge this gap, user participation is crucial (Sharp et al., 2023). Direct communication between users and developers allows the latter to gain insights into how the software is used in real-world scenarios, which can inform decisions about software functionalities and improvements (Abelein and Paech, 2014). However, achieving effective user participation is not always easy. Direct communication, for example, in the form of interviews, may not be available or feasible. Various factors, such as time constraints, geographical distance, or the size of the user base, can make it challenging to establish direct communication with users. The fact that direct communication can be so difficult to achieve (Abelein, 2013) can lead to issues where software behaviour that developers find intuitive may not be so clear to actual users (Mann, 2002).

In situations where direct interaction is not possible, online feedback from users can serve as an alternative. Users often leave valuable feedback in online forums, social media, app stores, or customer support channels. These sources provide abundant user-generated content, offering insights into user experiences and potential requirements for further development (Pagano and Maalej, 2013). However, these sources present their own challenges. Feedback collected from online sources is often unstructured, with users discussing various software functionalities in different contexts (Anders et al., 2023).

Research on the analysis of feedback from online sources has been extensive (Dabrowski et al., 2022) (Khan et al., 2019) (Lim et al., 2021) (Wang et al., 2019). Among the most common analysis goals are the classification of feedback into predefined categories, information extraction,

and content analysis. These approaches directly analyze the content of the feedback without putting it into the context of the software's functionalities. Thus, they do not answer the question of which of the existing functionalities of the software the feedback is actually addressing. Creating this context would allow feedback to be grouped by common themes more easily, which is a common task of developers in industry (Z. S. Li et al., 2024). This motivates the first problem that our approach aims to treat:

**P1: Understand which functionalities users are discussing in their feedback**

Additionally, feedback also contains concepts relevant to usage information (Anders et al., 2022). Usage comprises the interactions with the software in terms of UI elements touched, data input and output and the functionalities used. These interactions are part of a usage context in terms of the user tasks and activities. Usage information comprises context and usage. Unlike opinions, such as *"this feature is confusing"*, usage information focuses on the observable or inferred activities that lead to these opinions. For example, if the feedback of users is influenced by the fact that they consistently use a particular feature in unexpected ways, this is reflected in their usage information. By providing this information about how users talk about the software and their use of it, developers could gain a better understanding of the users. They may also discover differences in how they envision software and its functionalities being used as opposed to how their actual user base does. An example taxonomy shows that product quality, user intention, user experience, and sentiment are classified but not the concepts relating to usage information (R. Santos et al., 2019).

Users provide both explicit feedback in the form of written messages as well as implicit feedback in the form of monitoring data, such as logs (Maalej et al., 2009). Monitoring data can provide insights into which functionalities users are using and how they are using them. However, monitoring is limited to surface-level interactions and doesn't capture the full range of usage information, such as the rationale behind user actions or feedback on specific UI elements. It also presents ethical and data privacy challenges. The ACM Code of Ethics advises against collecting more personal information than necessary (Gotterbarn et al., 2018), making explicit user feedback a more viable source of information. Thus, in order to address the software issues raised by feedback, developers need to understand how the users are using the functionalities of the software by analyzing explicit user feedback. (Guzman and Maalej, 2014). This motivates the second problem our approach aims to treat:

**P2: Understand how the users use the functionalities of the software by analysing their feedback**

Despite the importance of feedback, manually analyzing it is a time-consuming and labour-intensive process, especially given the large volume of feedback developers must handle (Pagano and Maalej, 2013). Automation and tool support are required to integrate feedback analyses into development processes adequately. However, as interview studies reveal, developers in the industry often still manually analyse user feedback instead of using automated approaches (Z. S. Li et al., 2024) (Johanssen et al., 2019). These studies also highlight the lack of tool support. Thus, in order to properly address the previously stated problems (P1 and P2), developers need assistance through automation and tool support.

In summary, developers face the dual challenges of (1) understanding which functionalities of the software users are using (P1) and how they are using them (P2) and (2) processing vast amounts of user feedback efficiently, ethically and accurately to gain this understanding.

## 1.2. Solution Idea

As highlighted in the previous section, to find discrepancies between developers' expectations and users' actual needs, developers must first understand which functionalities of the software users are discussing in their feedback (P1) and how they are using those functionalities (P1). To facilitate this, the solution idea encapsulates two approaches: the relation of feedback to existing software requirements and the extraction of usage information from feedback. Both of these approaches can be used independently and in conjunction.

The first approach, introduced as feedback requirements relation (FeReRe), focuses on relating user feedback to existing software requirements in order to treat P1. Software requirements describe how a functionality is envisioned and how it is designed to work. By relating the feedback directly to these requirements, it allows developers to understand which functionalities of the software users are discussing.

The second approach expands on feedback analysis by focusing on usage information classification (UIC) from feedback in order to treat P2. While other frameworks could be used to classify usage information, our approach uses the TORE framework (Paech and Kohler, 2004) to categorize feedback. By classifying feedback according to the TORE levels, we gain a more granular understanding of how users interact with the system. This allows developers to compare their expectations to the users' actual use of the software by analysing how the users actually use the software.

Additionally, tool support for both approaches is needed to make the information more accessible for developers.

In summary, our dual approach—relating feedback to requirements through FeReRe and classifying usage information in feedback through UIC helps with the discovery of discrepancies between developer expectations and user needs. By automating key aspects of the process, developers are offered the tools they need to understand their users more effectively and ensure their software evolves in line with users' needs.

## 1.3. Research Methodology

In this section, the methodology used for research and design of the approaches proposed in this dissertation is explained. Section 1.3.1 explains the fundamental design science methodology used. Section 1.3.2 introduces the research goals of this dissertation and explains the design cycle.

### 1.3.1. Design Science Methodology

The research in this dissertation follows Wieringa's *design science methodology (DSM)* (Wieringa, 2014). DSM is a structured framework for problem-solving in the context of information systems and technology design. The methodology emphasizes the iterative cycle of designing and evaluating artifacts, such as models, software, or processes, to address real-world problems. DSM integrates both solution investigation and treatment development through two main cycles: the design cycle and the engineering cycle.

The design cycle focuses on creating a solution based on current knowledge and requirements. It consists of the *problem investigation*, *treatment design*, and *treatment validation*. As the problem tackled by this dissertation is already established (P1 and P2 in Section 1.1), we perform a *solution investigation* instead of the *problem investigation*. Here, we investigate the current state of the art of related solutions to transfer the already existing knowledge to the tackled problem.

*Solution Investigation* is the process of systematically exploring potential solutions to a problem. This involves researching existing solutions for related problems and identifying challenges. *Treatment Design* refers to the creation of an artifact (such as a system, process, or software) intended to solve a specific problem. The "treatment" is the proposed solution that aims to improve a situation by addressing a set of requirements or constraints. *Treatment Validation* involves testing the designed treatment to determine if it effectively solves the problem or meets the targeted objectives. This can be done through simulations, experiments, or field tests and focuses on evaluating whether the solution satisfies its intended purpose.

The design cycle is part of the larger engineering cycle, where the results of the design cycle are transferred to the real world (i.e., industry settings) for *treatment implementation* and *implementation evaluation*. In this dissertation, we are focusing solely on the design cycle without later industry implementation.

The individual steps of the design cycle are made up of two kinds of goals. The first is the *design goal*, which describes the design of an artifact to improve the problem context of stakeholders. Stakeholders, in this case, are people within the social context that are affected by the project. The Stakeholders have their own goals. In the case of this dissertation, these stakeholders are software users and developers. The second type of goal that is part of the design cycle is the *knowledge goal*. *Knowledge goals* refer to the objective of gaining new insights or contributing to scientific knowledge through investigation or evaluation. The knowledge goal seeks to understand the principles behind why certain treatments work or fail, contributing to the broader understanding of design science.

### 1.3.2. Research Goals & Design Cycle

This section describes the social context goals of the two stakeholders relevant to this dissertation: software users and developers. Afterwards, the design science research goals that support the social context goals are explained. Figure 1.1 shows the users' and developers' primary goals and the problems preventing developer goal success.

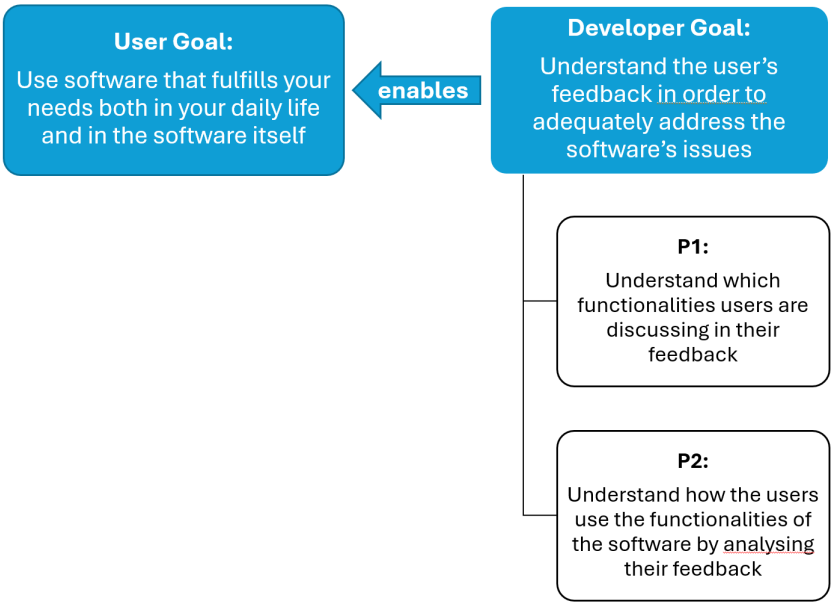



Figure 1.1.: Social Context Goals and Problems that Prevent Goal Success

In the context of this dissertation, the software users' primary goal within the social context is to use software that fulfils their needs. These needs can be software-specific, for example, a

specific UI design, or domain-specific, such as the need for specific requirements that aid the users in their daily lives.

The user goal is enabled by the developers through their primary goal: Understanding the user's feedback in order to adequately address the software's issues. By understanding the feedback, developers are able to address any issues the users might have with the software, thus allowing them to develop software that fulfils the user's needs. However, as explained in Section 1.1, two problems exist that prevent this goal from being achieved. **P1**, the difficulty of understanding which functionalities users are discussing in their feedback and **P2**, the difficulty of understanding how the users are using the functionalities of the software by analysing their feedback.

The approaches presented in this dissertation tackle P1 by relating feedback to the existing requirements of the software and tackle P2 by extracting usage information from the feedback, which explains how the users are using the software.

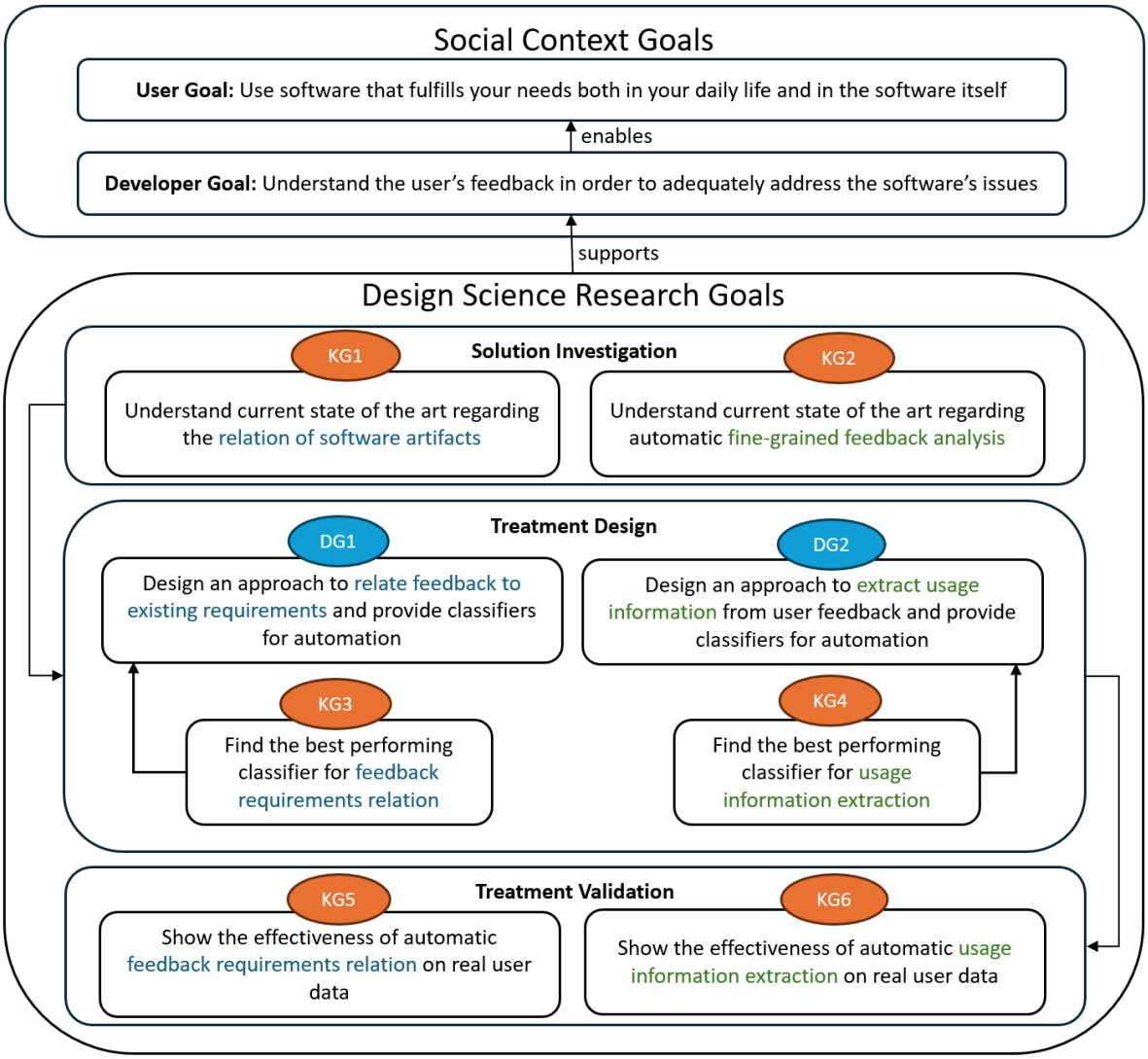



Figure 1.2.: Design Science Goal Structure of this dissertation including Design Goals (DG) and Knowledge Goals (KG). Arrows indicate that the fulfilment of one goal or investigation contributes to the fulfilment of the other

Thus, in order to tackle the problems that prevent the developers' goal from being achieved, which in turn enables the users' goal, design science research goals are formulated. These can be seen in Figure 1.2. Design Science Goals are symmetrical for both problems P1 and P2. Goals related to feedback requirements relation are marked in blue. Goals related to usage information classification are marked in green.

Starting with the *solution investigation*, *knowledge goals 1* and *2* are understanding the current state of the art regarding the relation of software artifacts and fine-grained feedback analysis. To achieve this, two mapping studies are performed to find existing approaches. To find approaches related to feedback requirements, we research approaches that perform the relation of software artifacts. These artifacts can be feedback, requirements, commit messages, or other natural language artifacts created during software development. For the usage information classification task, we look for approaches that perform similarly fine-grained user feedback classifications.

After analyzing the existing approaches, *design goals 1* and *2* are tackled during the *treamtent design*. During the design of the approaches, lessons learned from the current state of the art are considered. To then automate the approaches *knowledge goals 3* and *4* are completed. These include evaluating several classifiers to find the best-performing classifier for each task. Classifiers are based on machine learning models found during the state-of-the-art investigation. As part of the design of the approaches, tool support is also created to make the created information more accessible for developers. Because the requirements for the tool are directly related to the approaches themselves, there is no separate design goal for the tool support. The tool, Feed.UVL serves as a proof-of-concept prototype to show how the designed approaches can be integrated into the development process to help with P1 and P2.

The result of the *treatment design* are approaches for feedback requirements relation and usage information extraction along with classifiers for automation of each approach.

During the *treatment validation knowledge goals 5* and *6* validate the effectiveness of the presented approaches and classifiers. The validation is performed on real user data, namely, feedback and requirements from real software projects. A use case for both feedback requirements relation and usage information classification is established. The effectiveness of the automation of both tasks is then evaluated through metrics and comparison to manual task execution. The designed tool is not validated separately.

This dissertation investigates two different approaches that form separate tasks: the relation of feedback and requirements and the classification of usage information. Since both tasks are separate from one another but follow the same design cycle steps, Figure 1.3 shows a general template for both tasks. The design cycle is repeated for each task. The design cycle derives from the phases introduced in Figure 1.2.

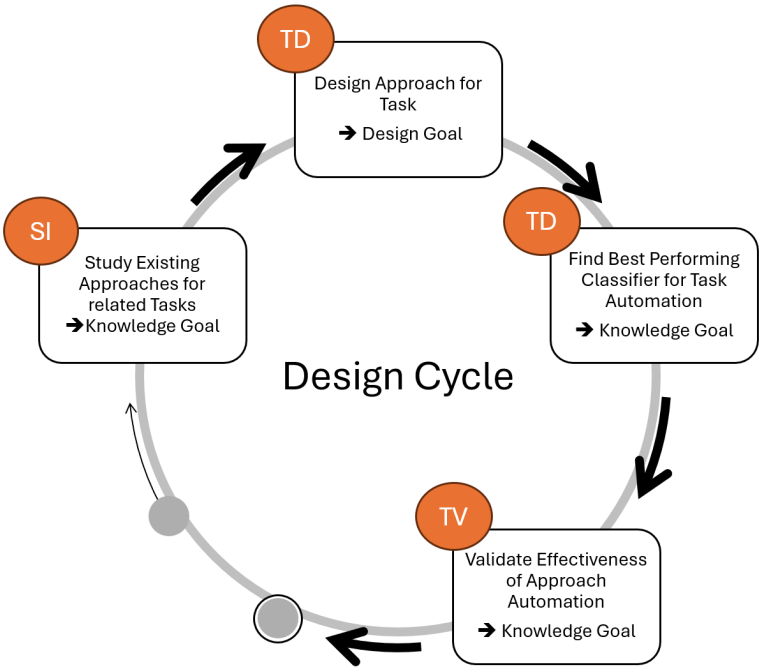



Figure 1.3.: Design Cycle Template for this Dissertation. Covering Solution Investigation (SI), Treatment Design (TD), and Treatment Validation (TV)

## 1.4. Contributions

This dissertation presents five contributions towards software engineering research.

The two main contributions are the approaches created for feedback requirements relation (FeReRe) (Section 5.1) and usage information classification (UIC) (Section 6.1) as part of *design goal 1* and *design goal 2*. These two approaches can be applied independently or in conjunction to highlight discrepancies between the developers' expectations of the users' needs and the users' actual, expressed needs. For both approaches, classifiers are designed to help automate the tasks (Sections 5.2 and 6.2) as part of *knowledge goal 3* and *knowledge goal 4*. The effectiveness of these approaches and related classifiers are evaluated on real user data in Chapters 8 and 9 as part of *knowledge goal 5* and *knowledge goal 6*.

The third contribution of this dissertation is Feed.UVL is a web-based microservice tool designed to support the two approaches and related classifiers, which is presented in Chapter 7. Feed.UVL is designed as part of *design goal 1* and *design goal 2* and thus provides functionalities to perform both feedback requirements relation and usage information classification manually and automatically. Additionally, it provides functionalities to gather feedback from online sources and manage feedback datasets. This makes the approaches easier to use. Also, a Jira plugin is presented, which connects to the Feed.UVL database and makes the feedback requirements relation and usage information classification more accessible for developers.

The fourth contribution is the two mapping studies which capture the current state of research regarding software artifact relation (Chapter 3) and fine-grained user feedback classification (Chapter 4). The approaches found during the mapping studies were fundamental towards identifying existing approaches and challenges. The lessons learned were considered during the development of the approaches presented in this dissertation. The studies were conducted as part of *knowledge goal 1* and *knowledge goal 2*.

The fifth contribution is a number of large, manually created datasets which were used for training and evaluation of the classifiers for both feedback requirements relation and usage

information classification. These are part of *design goals 1* and *2* and *knowledge goals 5* and *6*. The datasets presented in Section 2.5 are based on different software products and come from different feedback sources. All have been annotated through rigorous coding rules and interrater agreement sessions to provide high quality datasets for future development and reproducibility of the results presented here.

## 1.5. Dissertation Structure

As shown in Table 1.1, the dissertation is structured in five parts consisting of 11 chapters. Part I introduces the dissertation and necessary fundamentals. Part II discusses the two mapping studies conducted as part of the solution investigation. Part III describes the design of the FeReRe and UIC approaches designed as a result of design goal. These chapters also evaluate the best performing classifiers for each approach according to knowledge goals 3 and 4. Part IV evaluates the effectiveness of the designed approaches as part of a hypothetical deployment scenario in a company. Lastly, Part V provides a summary and outlook on future work.

Table 1.1.: Structure of the dissertation

| | | Preliminaries | Chapter |
|---|---|---|---|
| | Part I | Introduction | 1 |
| | | Fundamentals | 2 |
| **Design Cycle** | | Solution Investigation | |
| | Part II | **State of the Art - Software Artifact Relation**<br>*Knowledge Goal 1:* Understand the current state of the art regarding the relation of software artifacts | |
| | | **State of the Art - Fine-Grained Feedback Classification**<br>*Knowledge Goal 2:* Understand the current state of the art regarding automatic fine-grained feedback analysis | 3 |
| | | Treatment Design | |
| | Part III | **Feedback Requirements Relations**<br>*Design Goal 1:* Design an approach to relate feedback to existing requirements and provide classifiers for automation<br>*Knowledge Goal 3:* Find the best performing classifier for feedback requirements relation | 5 |
| | | **Usage Information Classification**<br>*Design Goal 2:* Design an approach to extract usage information from user feedback and provide classifiers for automation<br>*Knowledge Goal 4:* Find the best performing classifier for usage information extraction | 6 |
| | | Treatment Validation | |
| | Part IV | **Feedback Requirements Relation Evaluation**<br>*Knowledge Goal 5:* Show the effectiveness of automatic feedback requirements relation on real user data | 8 |
| | | **Usage Information Classification Evaluation**<br>*Knowledge Goal 6:* Show the effectiveness of automatic usage information extraction on real user data | 9 |
| | | Conclusion | |
| | Part V | **Summary** | 10 |
| | | **Future Work** | 11 |

## 1.6. Previous Publications

Several results of this dissertation have previously been published. Table 1.2 lists these publications and the chapters of this dissertation they correspond to.

Table 1.2.: Previous Publications

| **Publication** | **Chapter** |
|---|---|
| Schrieber H, Anders M, Paech B, Schneider K, "A Vision of Understanding the Users' View on Software". In: Joint Proceedings of REFSQ-2021 Workshops, Essen (Germany)/Virtual, 2021 | 2 |
| Anders M, Obaidi M, Paech B, Schneider K, "A Study on the Mental Models of Users Concerning Existing Software". In: Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol 13216, Springer, pp. 235-250, Birmingham (UK), 2022 | 2 |
| Anders M, "Relating User Feedback and Existing Requirements". In: Joint Proceedings of REFSQ-2023 Workshops, Barcelona (Spain), 2023 | 5, 6 |
| Anders M, Obaidi M, Specht A and Paech B, "What Can be Concluded from User Feedback? - An Empirical Study". In: IEEE 31st International Requirements Engineering Conference Workshops (REW), pp. 122-128. Hannover (Germany), 2023 | 2 |
| Anders M, Paech B, Bockstaller L, "Exploring the Automatic Classification of Usage Information in Feedback". In: Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol 14588, Springer, pp. 267-283, Winterthur (Switzerland), 2024 | 6 |
| Anders M, Paech B, "FeReRe: Feedback Requirements Relation Using Large Language Model". In: Requirements Engineering: Foundation for Software Quality, Lecture Notes in Computer Science, vol 15588, pp. 89-105, Barcelona (Spain), 2025 | 5 |

Chapter 2

# Fundamentals

This chapter explains fundamental concepts and terms used in this dissertation. Section 2.1 introduces feedback analysis as a general concept and discusses analysis goals, analysis coarseness, and feedback sources. Section 2.2 introduces the Task-oriented Requirements Engineering (TORE) framework used for usage information classification. Section 2.3 introduces relevant machine learning concepts, the classifiers used in this dissertation, and the metrics used to measure their performance. Section 2.4 introduces the development tools mentioned throughout the dissertation. Lastly, Section 2.5 presents the manually created datasets which are used for training and evaluation of the automatic classifiers designed in this dissertation.

Sections 2.2, 2.3.3 and 2.5 are partially based on previous publications (Anders et al., 2022) (Anders et al., 2024) (Anders and Paech, 2025).

## 2.1. Feedback Analysis

Feedback analysis refers to the process of evaluating feedback statements (also referred to as reviews synonymously) from users to improve a product. The goal is to continuously improve the product, in the case of this dissertation, software, to meet the users' needs and expectations and provide the best possible product. Analysis can either be performed on explicit or implicit user feedback (Maalej et al., 2009). Explicit feedback refers to written statements from users about a software product. Implicit feedback refers to indirect forms of feedback, such as monitoring information automatically logged by software. In this dissertation, we only analyze explicit feedback in textual form. Alternative forms of explicit feedback, such as drawings, for example, are not analyzed. The evaluation of feedback through analysis can be performed manually, automatically, or semi-automatically.

**Manual analysis** can either be performed in a structured or unstructured manner. A structured manual analysis approach assigns predefined classes to the feedback. This process is called coding (Saldana, 2021). An unstructured manual analysis approach does not follow these coding rules. Rather, the feedback statements are read, for example, by a developer while they take notes and highlight passages they deem important (Johanssen et al., 2019).

**Automatic analysis** refers to the process of using a classifier to perform a structured analysis of the feedback fully automatically. The classifier, designed for a specific analysis goal (Section 2.1.1), performs the feedback classification and provides the developers with an output that they can then process further (R. Santos et al., 2019).

**Semi-automatic analysis** is a hybrid between manual and automatic analysis. Here, a classifier is used as a recommender system, providing the developers with recommendations for the classification of the feedback. These recommendations are then manually either approved

or rejected by the developers. The semi-automatic analysis aims to reduce the time needed to analyze the feedback compared to fully manual analysis while not relying on the performance of the used classifier as much as during automatic analysis (Felfernig et al., 2013).

### 2.1.1. Analysis Goals

Any feedback analysis, be it manual, automatic, or semi-automatic, is performed with certain goals in mind. These goals define what the developers ultimately want to achieve with their analysis. There are numerous goals for feedback analysis. According to van Oordt et al. (Oordt and Guzman, 2021), practitioners collect and analyze feedback with seven different goals in mind with respect to requirements engineering (Figure 2.1). These are the identification of new features, the prioritization of existing requirements, spotting trends in the feedback, creation of bug reports, identification of pain points, identification of incompatibilities, issues with the usability of the software, and making assumptions about the users of the software.

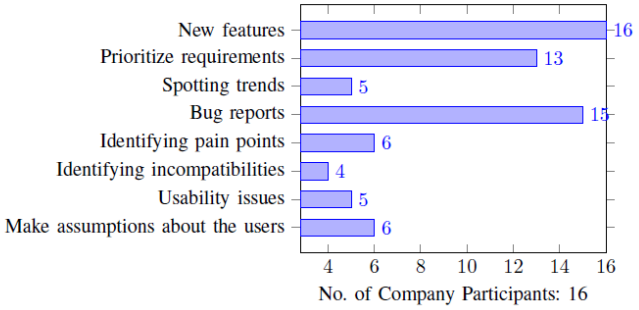



Figure 2.1.: Why do practitioners collect user feedback (Oordt and Guzman, 2021)

These seven general analysis goals can be achieved through the mining of information. Dabrowski et al. (Dabrowski et al., 2022), for example, differentiate between different types of mined information and analysis types from app reviews. This can be seen in Table 2.1. This list, however, is not exhaustive and merely an excerpt of the types of mined information found by Dabrowski et al. Different analysis goals can be achieved by combining a type of analysis combined with the mined information. To achieve a practitioner's goal of finding new features, for example, information extraction of user requests can be performed on the feedback. Other analyses are more supportive towards reaching a more general goal, such as the clustering of similar reviews, for example, to help with the spotting of trends.

For more information about potential analysis goals in research, refer to Chapter 4. The primary analysis goals that this dissertation aims to support, as classified by (Oordt and Guzman, 2021), are the spotting of trends and the identification of pain points. To achieve this, we perform two different types of analysis as classified by (Dabrowski et al., 2022). The first is the clustering of similar reviews by relating feedback directly to existing requirements. The second is the classification of usage information. Both of these analyses support the stated analysis goals. Trends can be spotted by clustering common feedback related to the same requirement and by looking at common usage information appearing in feedback. Pain points can be identified by looking at the users' statements about how they use the software through usage information. These pain points can then be traced back to the requirements through feedback requirements relation.

### 2.1.2. Statement, Sentence and Word Analysis

Table 2.1.: Analysis Types and Mined Information in App Reviews (Dabrowski et al., 2022)

| **App Review Analysis** | **Mined Information** |
|---|---|
| Information Extraction | Features<br>User Request<br>Non-Functional Requirements (NFR) |
| Classification | User Request Type<br>NFR Type<br>Issue Type |
| Clustering | Similar Reviews |
| Search and Information Retrieval | Feature-Specific Reviews<br>Review-Goal Links<br>Review-Issue Links<br>Review-Code Links |
| Sentiment Analysis | Feature-Specific<br>Review |
| Recommendation | User Request Priority<br>Review Response |
| Summarization | Review Summary |

User feedback can be analyzed at different levels of coarseness (Cambria et al., 2013). The level of coarseness selected can depend on multiple factors, such as the chosen analysis goals, available time for analysis, or the capabilities of the classifiers used. In general, there are three different levels of coarseness. Feedback can be analyzed as a complete statement, such as a whole app review submitted by a user. It can also be analyzed on a sentence level, where each sentence is analyzed individually. The least coarse level is the word-based analysis, where each word in a feedback statement is analyzed individually.

In this dissertation, we perform statement— and sentence-based analyses for the relation of feedback and requirements in Chapter 5. Word-based analysis is not practical for this goal because deciding to which requirement an individual word belongs is not feasible, nor does it provide meaningful information. For the usage information classification (Chapter 6), we perform sentence- and word-based analyses. A whole feedback statement can contain multiple types of usage information (Section 2.2). As a result, performing statement-based usage information classification would result in too much usage information being lost.

### 2.1.3. Feedback Sources

Feedback can be gathered from a large number of different sources. In their study, van Oordt et al. (Oordt and Guzman, 2021) identified 20 different sources of feedback that practitioners use to get explicit and implicit feedback from users and colleagues. These can be seen in Figure 2.2.

Implicit feedback sources such as usage data and error logs are irrelevant to this dissertation because we only handle explicit feedback. We also do not handle feedback that does not provide written statements from the software's users, such as internal blogs, colleagues, and phone calls. Of the remaining feedback sources, this dissertation performs evaluations using datasets from review platforms, social media, in-app feedback mechanisms, and surveys (Section 2.5).

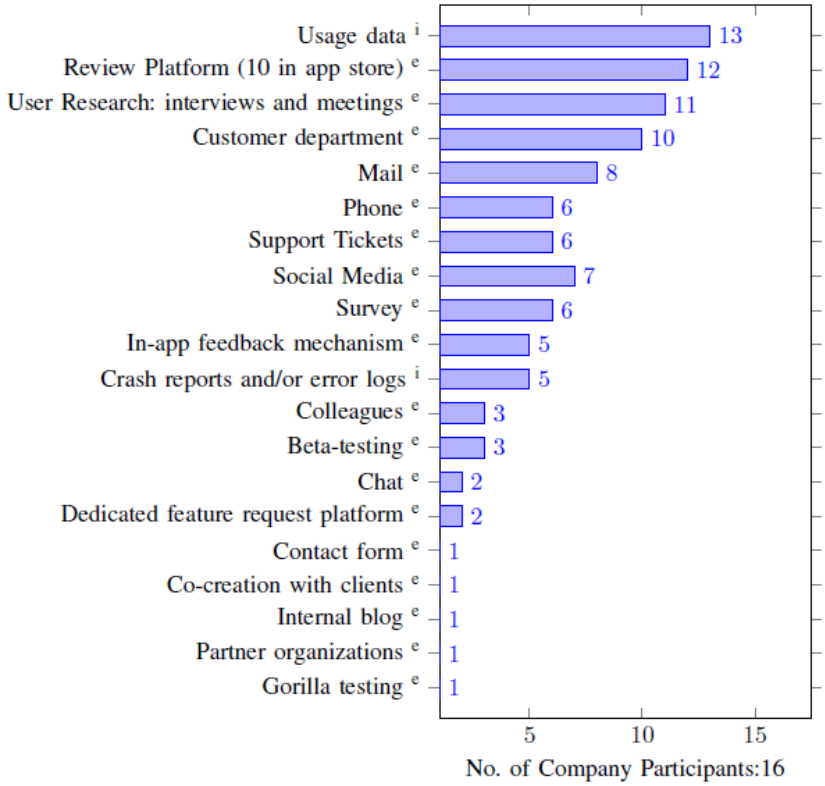



Figure 2.2.: Sources of feedback for practitioners, e denotes explicit feedback and i implicit feedback (Oordt and Guzman, 2021)

Review platforms are most commonly directly integrated into distribution platforms such as mobile application stores or other online stores. These review platforms provide functionalities to write feedback messages directly to software developers and often include the ability to rate the product. In research, these are among the most commonly used platforms (especially mobile reviews) (Wang et al., 2019). This dissertation uses reviews from the Google Play Store[1].

Social media refers to online platforms that facilitate social interactions between users. These include platforms such as Twitter, Facebook, or, in the case of this dissertation, the online forum reddit[2]. Here, users can exchange messages through forum posts. One user generates a post, giving it a name and a description, and other users can comment on this post or on previous comments. What separates social media from other sources, such as review platforms, is the fact that they are not necessarily designed to give feedback. They can serve as a platform for users to directly address developers if they have a presence on the platform, but they can also facilitate discussion between users about the software without developer input. In this regard, they are different from other sources, like review platforms (Williams and Mahmoud, 2017) (Iqbal et al., 2021).

In-app feedback mechanisms allow users to send feedback messages directly to the developers within the software they are using. They may also allow developers to ask specific questions for the users, allowing them to gather more targeted feedback, for example, for specific functionalities. This feedback source has the advantage of lowering the steps a user has to go through to submit feedback compared to other sources. This dissertation uses the SmartFeedback app integrated into the SmartAge-App-Suite (see Section 2.5 for further details).

[1] https://www.play.google.com/
[2] https://www.reddit.com

Surveys are a way for developers to get targeted feedback from their users. They can target specific groups of users and, by designing specific survey questions, target specific functionalities for which they wish to gain feedback. These surveys can either be sent directly to users or be published in online survey platforms such as Prolific[3], as is the case in this dissertation. Here, participants are paid for their participation. These platforms also allow users to target their survey towards specific demographics or people with specific knowledge backgrounds.

It should be noted that usage information classification and feedback requirements relation may also be applied to other feedback sources for which no datasets are presented in this dissertation. Many of these sources, such as support tickets, emails, and customer departments, are often only available internally within a company, which prohibits us from creating datasets and performing evaluations for them.

## 2.2. Usage Information Classification Framework TORE

We define **usage information** as the interactions users have with software, namely the UI elements and software functionalities they engage with, the data they enter, and the data the software outputs. These interactions occur within a broader usage context, encompassing the tasks and activities users perform in their daily lives, along with the data they interact with during these activities.

To classify the usage information, we use the TORE framework, which covers all these concepts (Paech and Kohler, 2004). TORE was developed for requirements elicitation and specification. It has also been applied in different industrial development projects in the past (Adam et al., 2009) to guide requirements engineers in their communication and decisions while eliciting and specifying requirements.

### 2.2.1. Original TORE Framework

The original TORE framework consists of 18 decision points. For each decision point, a part of the requirements is specified. TORE does not prescribe a specific template but gives some recommendations. The decision points are grouped into four abstraction levels, as shown in Figure 2.3. It is comprised of two stages. The first stage consists of the TORE *levels*, i.e. *Goal & Task Level*, *Domain level*, *Interaction level*, and *System level*. The levels are detailed by the second stage, the TORE *categories*, which specify the decision points.

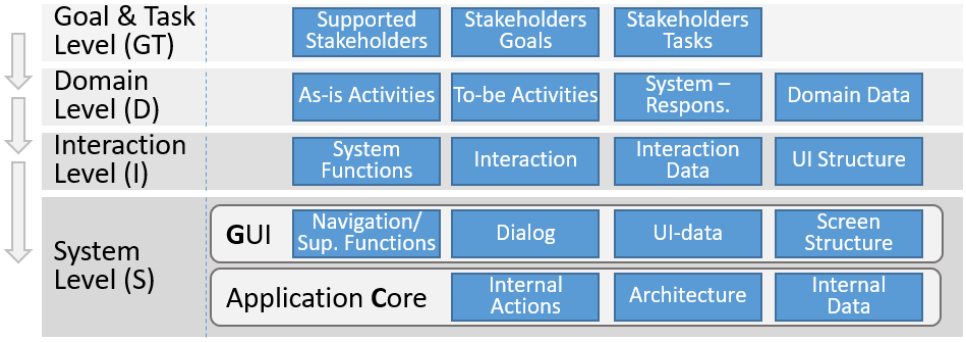



Figure 2.3.: Original TORE levels and categories (Kücherer, 2018)

The *Goal & Task level* captures the system context in terms of the persons and roles (stakeholders) who will be supported by the developed system and those who influence the resulting system. These stakeholders' goals and the tasks in the domain are also captured here.

[3]https://www.prolific.com/

The *Domain level* captures the system context in terms of *Activities*, *System Responsibilities*, and *Domain Data*. The *Activities* refine the *Stakeholder Tasks* into individual steps and are separated into *As-is Activities* and *To-be Activities*. *As-is Activities* describe how each step of a task is currently being performed. *To-be Activities* describe steps of *Stakeholder Tasks* in the future. *System Responsibilities* specify the *To-be Activities* that should be directly supported or even automated by the system. The *Domain Data* covers entities as well as their attributes and relations to one another, which are relevant to the *Stakeholder Tasks*.

The *Interaction level* captures decisions on how the software directly supports the users' tasks and activities. *System Functions* specify which functionalities the software will provide and to what extent. *Interactions* cover all actions between a user and the system. *Interaction Data* captures all data necessary for these user-system interactions. *UI Structure* groups the *Interaction Data* and *System Functions* through workspaces that represent the individual views of the software.

On the *System level*, the graphical user interface (GUI) is described by refining the *Interaction level*. The GUI is described through *Navigation / Support Functions*, which allows users to navigate between individual screens and realize parts of the *System Functions* that are visible to users. *Dialogs* refine *Interactions* by defining the sequence in which individual screens are navigated. *UI-Data* defines input data provided by users and output data provided by the system. *Screen Structure* refines the *UI Structure* by specifying the visual design of the user interface (UI).

Furthermore, the *Application Core* details are determined on the *System Level*. The *Application Core* covers the *Internal Actions* of the software, its' *Architecture*, and any *Internal Data* processed by the system.

In total, TORE comprises decisions ranging from the context through interactions to the level of an object-oriented design. We call the decision points *categories* in this dissertation, as we use them as coding and classification categories.

As shown in Table 2.2, Adam et al. (Adam et al., 2014) provide guiding questions for all decision points. These are intended to help make the individual decisions necessary for requirements engineering more explicit. According to Adam et al. *"the TORE framework does not prescribe a concrete requirements engineering process; rather, it guides and supports requirements engineers logically."*.

Table 2.2.: Original TORE and Guiding Questions

| Decision Point | Guiding Question (Adam et al., 2014) |
|---|---|
| **Goal & Task Level** | |
| Supported Stakeholders | Who are the stakeholders (e.g., departments, roles, persons, etc.) that are affected by the project? |
| Stakeholders Goals | What goals is the project supposed to achieve? What should be the benefit at the end? |
| Stakeholders Tasks | Which business processes and/or user tasks are to be analyzed and addressed in the context of the project? |
| **Domain Level** | |
| As-is Activities | How are the business processes and/or user tasks currently performed? What are their strengths and weaknesses? |
| To-be Activities | How should the business processes and/or user tasks be performed in the future in order to be able to achieve the project goals? |
| System - Responsibilities | Which parts/steps of the to-be business processes and/or user tasks should be supported or even automated by the system to be developed? |
| Domain Data | Which data and rules are relevant in the considered business processes and/or user tasks? |
| **Interaction Level** | |
| Interaction | How should users or external systems interact with the system to be developed for achieving the results of certain steps in the business processes and/or user tasks? |
| System Functions | Which system functions are needed for realizing the system responsibilities or interactions? |
| Interaction Data | Which data are exchanged in the interactions? Which interaction rules apply? |
| UI Structure | How should data and system functions be grouped logically within the user interface? |
| **System Level** (GUI) | |
| Navigation/Support Functions | Which additional functions are needed to support the user's navigation? |
| Dialog | How should the dialogs between user and system be designed? How do screen transitions take place? |
| UI-Data | Which UI-Element should be used and which data should they represent? |
| Screen Structure | How should the visual design of the user interface look like? |
| **System Level** (Application Core) | |
| Internal Actions | How should the system functions be realized by means of methods, procedures, etc.? |
| Architecture | How should the system be organized/structured internally? |
| Internal Data | How should the business data be represented in the data storage? |

### 2.2.2. Adapting TORE for Feedback Analysis

In order to apply TORE for user feedback classification, we made several changes to its levels and categories. We generally kept TORE's abstraction levels but reduced the number of categories and levels by combining those for which distinctions were not necessary for feedback analysis. We also changed some of the categories' names for clarity and ease of use. Figure 2.4 shows how the original classes were combined. The colors in the figure correspond to the levels of the categories. Orange for the *Goal & Task Level* and *Domain Level*, yellow for the *Interaction Level*, and blue for the *System Level*.

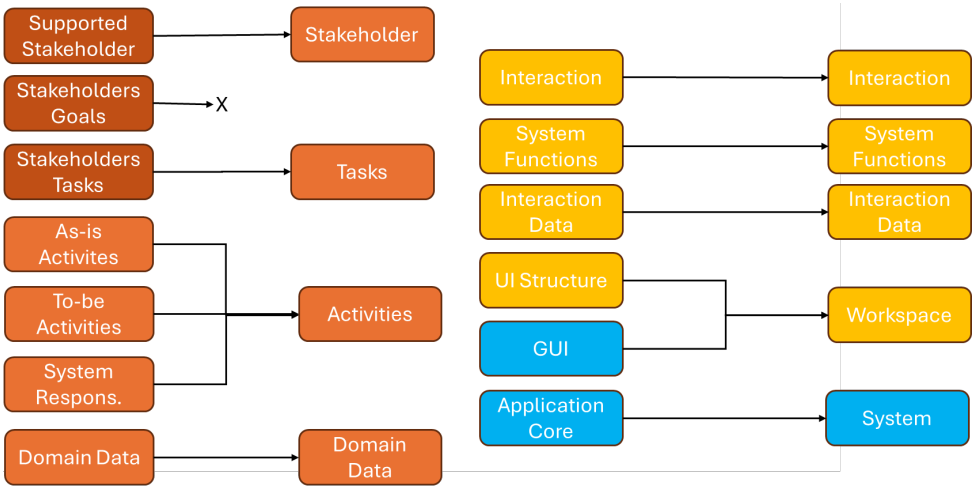



Figure 2.4.: Transition from original to adapted TORE categories

As a first step, we combined the original first two levels (*Goal & Task level* and *Domain level*) into the *Domain level* because there was no necessity to maintain the distinction between the two with the reduced number of classes. *Supported Stakeholder* and *Stakeholders Tasks* were renamed to *Stakeholder* and *Task* respectively. During multiple analyses of feedback with the TORE framework (Anders et al., 2022) (Anders et al., 2024), we found that the *Stakeholders Goals* category was almost never expressed by users in their feedback, occurring once in a feedback dataset of around 26.000 words. Consequently, we removed it from the adapted framework. In addition, the distinction between *As-is Activities* and *To-be Activities* is unnecessary for analyzing feedback because it concerns existing software. We also combined the *System-Responsibilities* with the two types of activities, as they are part of *As-Is Activities*. The resulting category is named *Activities*. No changes were made to the *Domain Data*.

On the *Interaction level*, we replaced the original *UI Structure* category with a *Workspace* category. This captures all statements related to specific UI elements, including GUI elements (originally on the *System level*), as they refine workspaces.

Lastly, we unified the categories on the *System level* into one category called *System* because users rarely have detailed knowledge of the inner workings of the software and thus do not reliably differentiate between *Internal Actions*, *Architecture*, and *Internal Data*.

Table 2.3 provides a description of the resulting levels and categories and lists examples of usage information extracted from feedback. The feedback excerpts are largely extracted from the datasets introduced in Section 2.5 and discuss the popular hiking app Komoot.

The examples for the levels show usage information assigned to complete sentences. The *Domain level* example highlights an everyday action within the domain of Komoot. The *Interaction level* example shows a user's description of an interaction with the software. The *System level* shows a remark concerning the system performance.

Table 2.3.: TORE Categories, their Definitions, and Examples

| Category | Definition | Example |
|---|---|---|
| **Domain level** | | "It allows me to discover new things" |
| Stakeholders | Roles supported by or influencing the developed software (e.g. Users or Developers) | "I'm more of a **power user** while my friend is an **infrequent user**" |
| Tasks | Responsibilities of the Stakeholder as part of larger processes in the domain | "I frequently **plan outdoor routes** for my adventures" |
| Activities | Steps in the Stakeholders' Tasks | "This is the best app for **running, hiking** or **biking** out there" |
| Domain Data | Data relevant to an Activity | "The app keeps letting me down with bad information on **camping grounds** about 35% of the time" |
| **Interaction level** | | "Easy to use and allows me to change different modes from hiking to biking." |
| Interaction | The interaction between a user and the software | "You can **track** your miles, **make** your own trails and even **get** directions" |
| System Functions | Functions executed by the software that consume, manipulate or produce data. | "It lacks **data sharing to Google Fit**" |
| Interaction Data | Data relevant for the System Functions or Interactions | "You deleted the **maps** I had, and now I can't even access **online maps**" |
| Workspace | Grouping of Interaction Data and System Functions which are relevant for one Task and specific UI elements | "The app shows completely different numbers than it records in the **Completed Rides Tab**" |
| **System level** | | "It should not keep reloading data if it has already been loaded" |
| System | Components of the software as well as data and actions processed internally | "Komoot's **app** has become horribly slow and I'm not sure if that is because of **loading data** from some **server**" |

For the categories, the *Stakeholder* example highlights two different roles that influence software. *Tasks* capture larger processes in the domain, which consist of multiple *Activities*. These *Activities* are individual steps in the domain. In the examples of Table 2.3, this is the Outdoor-Activity-Domain, for example, hiking and biking. *Domain Data* captures all entities that are relevant to the activities of the domain, for example, "camping grounds". *Interactions* describe actions performed by the user with the software. *System Functions* are functionalities that the software provides or, according to the user, should provide, as can be seen in the example. *Interaction Data* comprise data provided and used in the interaction. *Workspace* highlights which specific view in the UI or which element in a view the user is discussing. Lastly, the *System* category captures mentions of the software as a whole (in the example "app") or users' discussions about the inner workings of the software ("loading data" and "server").

### 2.2.3. Alternative Models

TORE is not the only framework capable of being applied to usage information classification. It has similarities to other requirements models, which could also be applicable.

Lauesen (Lauesen, 2002), for example, introduces Goal, Domain, Product, and Design levels. Goal and Domain levels correspond to the original TORE levels. The Product level focuses on the functions on TORE's *Interaction level*. The Design level focuses on the GUI details as in the original TORE *System level*.

Gorschek et al. (Gorschek and Wohlin, 2006) also introduce a requirements model with different abstraction levels. They use Product, Feature, Function, and Component levels. The Product level comprises *Goals* as in the original TORE, and the Features correspond to the *Activities*.

The Function level corresponds to our *Interaction level* and the Component level to our *System level*. However, we use more categories to distinguish data and UI information in addition to functions on the *Interaction level*.

In our view, the fine-grained categories of TORE allow us to extract more detailed usage information. The setup of levels and categories also allows us to investigate the potential of classifying different granularities. For these reasons, we chose TORE as the framework for defining the different types of information that are extracted in this dissertation. However, the classifiers presented in Section 6.2 could also be applied to any other usage information framework as long as sufficient training data is created. Thus, TORE serves more as a proof of concept for the feasibility of usage information classification in user feedback.

## 2.3. Machine Learning Concepts and Classifiers

This section introduces concepts related to machine learning, which are important for the dissertation, namely machine learning (ML), deep learning (DL), and large language model (LLM), as well as related terms.

Afterward, the classifiers used in Section 5.2 to automate the relation of feedback and requirements and the usage information classification in Section 6.2 are introduced.

### 2.3.1. Machine Learning

Michie et al. define machine learning as *"automatic computing procedures based on logical or binary operations, that learn a task from a series of examples"* (Michie et al., 1995).

While machine learning is also used in other application areas, such as computer vision, this dissertation focuses on ML in the context of natural language processing (NLP). NLP refers to the *"theoretically motivated range of computational techniques for analyzing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like language processing for a range of tasks or applications."* as defined by Liddy et al. (Liddy, 2001).

In simpler terms, this dissertation focuses on models designed to classify written segments of text corpora into predefined classes. A model is a machine learning algorithm designed to process and understand data. The classification by the model is based on information gained through previous text analysis. This process of feeding text data into a model to tune it for the desired classification is referred to as **training** (Yan, 2022).

Depending on the available data and the employed model, different types of training can be used. The two main types of training are supervised and unsupervised training (Goodfellow et al., 2016). **Supervised training** utilizes a set of labelled training data to tune the model. In this case, the provided text contains annotations labelling text segments in the data with pre-defined classes. This annotation is most often done manually before the model's training. On the other hand, **unsupervised training** does not use annotated data, meaning that rather than learning on text annotations, the model learns the data's inherent structure. A third type exists, called **semi-supervised training** (Goodfellow et al., 2016). This is a hybrid between supervised and unsupervised training. Here, a small set of annotated data is used to initally train the model, which then predicts annotations of larger unlabeled datasets. This process is repeated iteratively to create more accurate predictions by the model.

Training performed in this dissertation mainly focuses on supervised training. Pre-trained models, which partly utilize unsupervised training, are also employed.

In machine learning models, **features** refer to the measurable properties or characteristics of the data that the model analyzes to make predictions (Nargesian et al., 2017). When applied to text, features can represent various aspects of the text, such as its length, the frequency of

certain words, or the presence of specific keywords. These features are used as the model's input values, meaning that each feature is assigned a numerical value or a category that the model processes. In this context, **feature engineering** is the process of identifying and extracting these relevant features from the raw text to convert them into input values (Nargesian et al., 2017). The model uses these numerical or categorical values as the basis for its predictions or classifications.

The individual segments of a text that a model processes are called **tokens** (Ali et al., 2024). Commonly, these are individual words in the text that are processed subsequently. Some models like BERT can even process subword tokens, where complex words are broken into multiple tokens to allow for a more effective understanding of the words (see Section 2.3.6).

### 2.3.2. Deep Learning

Deep learning emerged as a distinct subgroup of machine learning in the NLP context around 2012 (Young et al., 2018). It is most commonly associated with the use of **neural networks**, which are a type of model that tries to emulate the human brain.

Neural networks generally consist of neurons that form individual nodes within the model. These neurons store values and are joined together by a set of biases and weights (Abdi et al., 1999). The neurons are separated over different layers within the model. While there is no universally applicable definition for how these networks are set up due to the various types of neural networks that exist, the layers can generally be separated into three types: the input and output layers, as well as the hidden layers in between. Figure 2.5 shows a simplified representation of this.

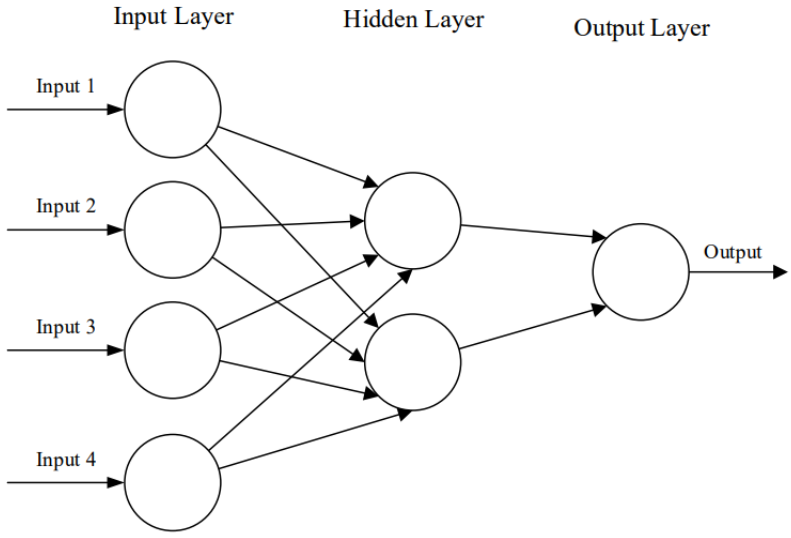



Figure 2.5.: Simple three layered neural network (O'Shea and Nash, 2015)

The input layer consists of feature vectors, which are mathematical representations of the input data. Inside the hidden layers, the model learns about the data by adapting the weights and biases associated with the neurons. The output layer then forms the output for the feature vectors.

Zhao et al. (Zhao et al., 2021) differentiate **"conventional" machine learning** and **deep learning** approaches by the latter's better capabilities to handle large amounts of raw data without the need for extensive feature engineering. They also attribute better capabilities to deep learning models to correctly process previously unseen data.

Further details on the specific deep learning approaches relevant to this dissertation can be found in Sections 2.3.5, 2.3.6, and 2.3.7

### 2.3.3. Large Language Models

A language model is a statistical or machine learning model that predicts the likelihood of a sequence of words in a language by understanding patterns, context, and structures within text data, enabling tasks like text generation, completion, and comprehension (Bengio et al., 2000).

LLMs are a form of language model trained on large text corpora using neural network architecture (Min et al., 2023). **Language model's** primary function is to predict words or characters succeeding a given input. As such, they are a form of probability model that calculates which word or character is most likely to occur next in a given text, considering the given input.

These models utilize extensive amounts of text collected for their training. The Llama 3 (Section 2.3.7) training dataset, for example, used over 15 trillion tokens of text and code to achieve its final model. As a result, these models learn to predict words reliably and are able to use extensive amounts of context in their predictions. **Context** refers to the surrounding text or sequence of words, sentences, or paragraphs that the model uses to understand and generate predictions (Brown, 2020). In the case of Llama 3, the maximum context that the model considers is 8192 tokens.

LLMs often come in different configurations differentiated by the number of parameters they include. These **parameters** typically refer to the number of weights and biases in the model. A larger number indicates an increase in the quality of the model but also an increase in complexity. The more complex the model, the more sophisticated the hardware required.

The training of the models is known as **self-supervised learning** (Goodfellow et al., 2016). This is a form of human-unsupervised learning where unlabeled data is fed into the model, but parts of the text are intentionally hidden. The model is then tasked with predicting the hidden parts. Its weights and biases are adjusted to achieve the best possible prediction results. This process is also referred to as *masking* (Devlin et al., 2018).

Generally, there is a distinction between two types of LLMs, generative and discriminative models (Bishop and Nasrabadi, 2006). **Generative LLMs** (such as GPT and LLama in Section 2.3.7) are designed to generate new data (text) similar to the input and training data. They are able to create semantically correct and contextually relevant text. **Discriminative models** (such as BERT in Section 2.3.6) focus on classifying the input data into classes. They are mostly trained to predict labels or features of an input text.

### 2.3.4. Stanford Named Entitiy Recognizer (SNER)

The Stanford Named Entity Recognizer (SNER) is a conventional machine learning classifier for named entity recognition (NER). NER is a natural language processing task aimed at extracting predefined entities in a text. This is mostly done on a word or term basis, meaning that individual words or sequences of words in the text are assigned a specific entity class. Common classes for NER include persons, organizations, dates, and locations (Sang and Meulder, 2003). The following is a simple example of the extraction of named entities from a sentence.

**Example Sentence:**
Jane and John Doe went to Berlin for a conference hosted by Microsoft.

**Named Entities:**

| | |
|---|---|
| Jane, John Doe | **PERSON** |
| Berlin | **LOCATION** |
| Microsoft | **ORGANIZATION** |

Stanford Named Entity Recognizer is an augmentation of a conditional random fields (CRF) (Lafferty et al., 2001) system. CRFs are a probabilistic model designed to predict labels in structured data. In our case, this data is the text we want to classify. They predict the probability of labels given the context of neighbouring labels. This context allows CRFs to capture the dependencies between labels, increasing the accuracy of predictions. SNER augments this approach by introducing long-distance dependencies, essentially increasing the number of neighbouring labels considered when predicting a label.

In this dissertation, SNER is used in Section 6.2 to classify usage information in user feedback. Usage information extraction does not strictly fit SNERs original design because it mostly focuses on extracting nouns, as in the example above. Usage Information also contains verbs in the case of actions performed in everyday life and with the software. However, the capabilities of NER classifiers like SNER to perform word-based classification while considering the context of surrounding labels also fit the requirements for a usage information classifier.

### 2.3.5. Bidirectional Long Short-Term Memory (Bi-LSTM)

Bidirectional long short-term memory (Bi-LSTM) models are a form of neural network that utilizes bidirectionality and long short-term memory (LSTM) to increase their capabilities of handling long data sequences. The concepts of bidirectionality (Schuster and Paliwal, 1997) and LSTM (Hochreiter and Schmidhuber, 1997) were first introduced in 1997.

Essentially, **LSTMs** are a form of recurrent neural network (RNN) (Pearlmutter, 1989). RNNs process data by including a "hidden state" that stores information from previous steps, such as previously processed words. This serves as the short-term memory of the network. As each word in a given input is processed, the network takes this hidden state information into consideration when making a prediction. The hidden state is continuously updated, thus allowing the network to retain information and improve the accuracy of predictions. However, RNNs can struggle with long-term dependencies, for example, when words that are far apart in a sentence relate to each other.

LSTMs are designed to address this issue of long-term dependencies. They do this by introducing a memory cell and so-called "gates" (Hochreiter and Schmidhuber, 1997). The memory cell stores the information from previous steps, serving as the long-term memory of the network. The gates handle the information in the memory cell. There are three different types of gates. The *Forget Gate* decides which information in the memory cell is retained or discarded. The *Input Gate* determines which information should be added to the memory cell. Lastly, the *Output Gate* decides which information in the memory cell should be used as the hidden state for the current processing step.

In standard LSTMs, information is processed in only one direction. In the case of text, this would mean that the words of a sentence are only processed in the direction that a human would read. As a result, only words that precede the currently processed word can be used as context. However, in actuality, words following the currently processed word can also influence the context and, thus, the correct prediction. **Bidirectionality** in a Bi-LSTM solves this issue by also processing the data from the other direction (Schuster and Paliwal, 1997). This allows

the network to include future context in its predictions by updating its hidden state according to the words that follow the currently processed word. Figure 2.6 highlights the concept of bidirectionality through the forward and backward layers in an LSTM.

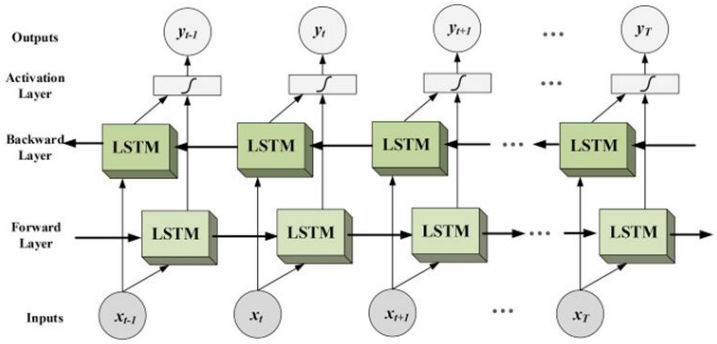



Figure 2.6.: Basic Bi-LSTM structure [4]

This dissertation utilizes a Bi-LSTM based on (N. Li et al., 2019) to classify usage information in feedback (Section 6.2). The approach feeds word embeddings of the text into the input layer of a Bi-LSTM. **Word embeddings** are mathematical representations of words in a text as vectors. These vectors capture the semantic meaning of the words and their relationships to other words in the input text. The approach utilizes word2vec (Mikolov et al., 2013) to compute these embeddings. The classification of the model is improved through the inclusion of these word embeddings.

### 2.3.6. Bidirectional Encoder Representations from Transformers (BERT)

The Bidirectional Encoder Representations from Transformers (BERT) model is a deep learning large language model developed by Google in 2018 (Devlin et al., 2018). The concept of **bidirectionality** was introduced in the previous section and functions similarly in BERT by processing input text from both directions (i.e. left-to-right and right-to-left). BERT can operate on subword tokens by splitting words into multiple tokens. This allows it to handle even complex and rarely used words more effectively. A word like "unhappiness" might be divided into up to three tokens, namely "un" as a prefix indication negation of the word, "happy" as the root word itself, and "ness" as a suffix indicating a condition.

BERT is pre-trained on a large corpus of text consisting of around 3.3 billion words using two training techniques: masking, as introduced in Section 2.3.3 and next-sentence-prediction, where the model trains to determine if one sentence follows another (Devlin et al., 2018). This pre-training allows the model to learn to generate accurate language representations. It can then be fine-tuned to specific tasks, such as classification tasks, using only a small amount of task-specific data.

BERT uses a transformer architecture (Vaswani et al., 2017). **Transformers** are a type of neural network architecture that uses self-attention to capture long-range dependencies in the input text. Essentially, **self-attention** allows the model to weigh the importance of words in an input text relative to each other (Vaswani et al., 2017). This is done by computing scores for each word by comparing them to every other word in the text. This is different from the processing of Bi-LSTMs because, unlike the Bi-LSTMs bidirectionality, the self-attention mechanism of the transformer considers every word in the input relative to every other word simultaneously, regardless of their positions in the sentence. This allows BERT to identify important contextual

[4] https://www.i2tutorials.com/deep-dive-into-bidirectional-lstm/ (Last Accessed: 30.07.2024)

relationships, even if words are far apart. The resulting values are a representation of the input text where the relationships between the words are captured, leading to more effective and accurate language understanding.

Figure 2.7 shows an overview of the transformer architecture. The **encoder** on the left consists of multiple layers of this self-attention. The input text is converted into embeddings and enriched by multiple layers of self-attention and feed-forward networks. A **feed-forward network** is a type of artificial neural network where information moves in one direction—from the input layer, through one or more hidden layers, to the output layer—without looping back. Each layer's neurons take inputs, process them, and pass the results to the next layer (Goodfellow et al., 2016).

The output of the encoder are contextual representations which capture the relationships and meanings of each token. The self-attention and feed-forward networks have residual connections (arrows on the left bypassing the blocks). These combine the original input of each layer with the output of the self-attention and feed-forward networks, ensuring that information from earlier layers is not lost. BERT does not use a Decoder, as shown on the right side of Figure 2.7 because, as a discriminative large language model, it does not generate output text. It merely generates the context-rich representations of text. In this dissertation, the context-rich representations generated by BERT are used for both feedback requirements relation and usage information classification.

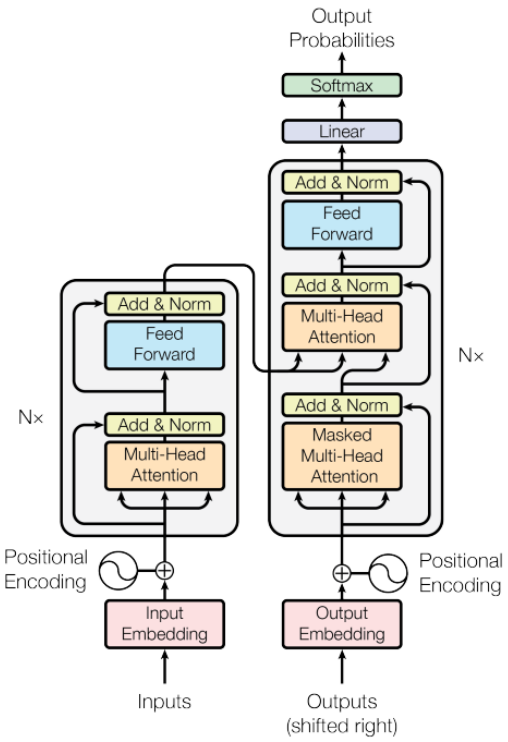



Figure 2.7.: The Transformer model architecture (Vaswani et al., 2017)

Several variations of BERT have been designed since its creation (Qiu et al., 2020). Next to the original BERT, this dissertation also uses DistilBERT (Sanh et al., 2020) and RoBERTa (Y. Liu et al., 2019). DistilBERT is a 40% smaller version of BERT created to run more efficiently and 60% faster while maintaining 97% of BERT's language understanding, according to the authors. RoBERTa is a variant of BERT that seeks to improve its performance by optimizing the pre-training process. It removes the next sentence prediction task, trains on a larger dataset, and uses longer sequences of text.

### 2.3.7. GPT & Llama

GPT and Llama are the two generative large language models used in this dissertation. GPT stands for Generative Pre-Trained Transformer and was first introduced by OpenAI in 2018 (Radford et al., 2018). At the time of writing, the most advanced model available is GPT4o (OpenAI, 2024). As a proprietary model, many details about its specific architecture and training data are not publicly available. The model is only available through the chat interface provided by OpenAI[5] or through its API.

Llama (**L**arge **La**nguage **M**odel Meta **AI**), on the other hand, is a semi-open-source model first released in 2023. The currently most advanced model available is Llama 3 (AI@Meta, 2024). While some details about the specific training data used are still proprietary information to Meta, in contrast to GPT, the model can be run locally. As previously mentioned, Llama 3 is trained on over 15 trillion tokens of text and code with a maximum context length of 8192 tokens.

Unlike BERT, GPT and Llama utilize the **Decoder** part of the Transformer architecture. For GPT, this architecture can be seen in Figure 2.8. The decoder generates output sequences from input information. It takes the text representations and produces one output token at a time, using what it has generated so far to help decide the next token. Essentially, the decoder converts abstract input representations into meaningful output sequences, in this case, generating text.

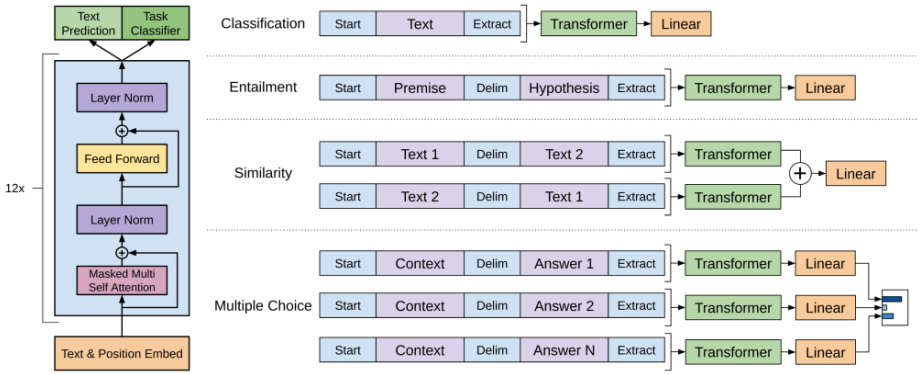



Figure 2.8.: GPT Transformer Architecture (Radford et al., 2018)

Working with these generative large language models is quite different from working with discriminative models such as BERT. Instead of fine-tuning the models on task-specific data and using the output representations to, for example, perform classifications, GPT and Llama generate output text based on the input text given to them. Thus, generative LLMs require the creation and refinement of prompts which are used as input into the model as clear text. These prompts describe the task the model is to complete and include the data on which the task is to be performed. These prompts can take different forms and are suited for different tasks depending on their intended usage (Ronanki et al., 2024)

### 2.3.8. Classifier Performance Metrics

In this dissertation, classifiers are evaluated with respect to multiple ground truths (Section 2.5). These were created via manual annotation through multiple annotators based on predefined and continuously updated coding rules. These ground truths are treated as the true classification. This means that it can be assumed that all labels in the ground truth are correctly assigned.

The aim of the evaluation is to determine to what degree a classifier is capable of identifying the correct classification. We refer to this as the classifier's performance. Precision (Formula

[5] https://chatgpt.com/

2.1), recall (Formula 2.2), and F1-Meassure (Formula 2.3) are used to evaluate this performance for the usage information classification (Sokolova and Lapalme, 2009). This is either done for individual classes or as a mean over all classes. Essentially, precision calculates how many of the instances classified as a certain class are correct. Recall measures how many correct instances of a class are found by the classifier. F1 calculates the harmonic mean of precision and recall.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.2}$$

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.3}$$

The metrics are calculated by comparing classifier results and ground truth annotation. True Positives (TP) are instances where the classifier correctly predicted the positive class. For usage information classification, the positive class is the class that is assigned to a given text segment in the ground truth. For feedback requirements relation, the positive class is a requirement that is related to a feedback statement in the ground truth. False Positives (FP) are instances where the classifier incorrectly predicted the positive class. In this case, the classifier incorrectly assigns a class to a text segment that is not classified as such in the ground truth and is thus false. False Negatives (FN) are instances where the classifier fails to predict the positive class. This means that the classifier does not assign a specific class to a text segment, even though it is classified as such in the ground truth. True Negatives (TN) (as shown in Formula 2.5) are instances where the classifier correctly does not assign the class to a segment.

In this dissertation, feedback requirements relation is proposed as a semi-automatic task. This means that the classifier serves as a recommender system for a human annotator to reduce their workload when performing the relation. Thus, as a classification task, feedback requirements relation benefits more from high recall (Berry, 2021) than from precision. High recall leads to most of the true positives being correctly assigned. This means that only the false positives have to be filtered out manually, which reduces the amount of manual labour necessary for the task. Thus, instead of F1, the F2 measure is used for feedback requirements relation tasks as it puts much greater emphasis on recall than on precision (Formula 2.4) (Sokolova and Lapalme, 2009). It should be noted that other F-Measures exist, which put even more weight on recall (increased $\beta$ in Formula 2.4). However, a recommender system, as proposed in this dissertation, still benefits from precision. Higher precision reduces the number of incorrect recommendations the system makes and thus reduces the manual labour required to correct them. As a result, F2 was chosen as a reasonable compromise to measure classifier performance. It balances a focus on high recall while maintaining the relevance of precision.

$$F\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}} \text{ where } \beta = 2 \tag{2.4}$$

For the usage information classification task, however, recall should not be more emphasized than precision due to the time-intensive labour necessary to classify usage information manually. The task of manually correcting the automatic classifier's labels would still be too labour-intensive for any potential users. This means that precision is also a critical measure for judging the classifiers' performance in this task. It is just as important to reduce the number of false positives as it is to reduce the false negatives for usage information classification. Thus, we decided to use F1 instead of F2 because it puts a similar emphasis on precision and recall.

Another potential measure is Accuracy (Formula 2.5), which measures the ratio of the number of correct predictions (TP and TN) to the total number of predictions (Sokolova and Lapalme,

2009). This can serve as a general performance indicator for a classifier because it shows the proportion of correctly predicted instances out of all instances. However, accuracy is only really useful when classes in the dataset are balanced (Chawla et al., 2004), which is not the case for any of our classification tasks. The metric is thus disregarded for this dissertation.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.5}$$

In summary, we utilize precision, recall, and F1-measure for the usage information classification and precision, recall, and F2-Measure for the feedback requirements relation.

## 2.4. Development Tools

During the course of this dissertation, five development and software deployment tools were used. These were mainly used to aid in the development and deployment of Feed.UVL (Chapter 7). These tools are Jira, Docker, Jenkins, Traefik and Portainer. Additionally, MLFlow was used to manage the machine learning experiments performed in Chapters 8 and 9.

Jira is an issue tracking system (ITS) published by Atlassian[6]. An ITS is a tool designed to help developers with the management of requirements and development processes. They can also help with the tracking of bug reports and any tasks related to requirements. In the context of this dissertation, Jira is used to document requirements for the creation of datasets (Section 2.5). It is also used by Feed.UVL to provide developers with easier access to the classification results of feedback requirements relation and usage information classifiers (Section 7.2.6).

Docker[7] is a platform that automates the deployment and management of software using containerization. Containerization is the process of encapsulating a software's functionalities into individual units called containers. This enables developers to package software into containers while ensuring consistent environments across development, testing, and production. In this dissertation, Docker is used as a framework for the containerization of individual Feed.UVL functionalities (Section 7.1).

Jenkins[8] is an open-source automation server used in this dissertation to facilitate the continuous deployment of the docker containers that make up Feed.UVLs functionalities. It automates the process of building, testing, and deploying the code in these containers. This ensures that code changes are automatically propagated to the production and testing environments of Feed.UVL without the need for manual intervention.

Traefik[9] is an open-source tool that acts as a reverse proxy, meaning it directs incoming requests to the right backend services. It's especially useful for applications running in containers because it automatically detects and configures services without needing manual updates. Traefik can handle tasks like securing connections with HTTPS and distributing traffic. It also comes with a dashboard for monitoring traffic and performance.

Portainer[10] is an open-source tool that provides a simple web-based interface to manage and monitor containers, making it easier to work with Docker and other container platforms. It helps with the deployment of containers, managing volumes and networks, setting up stacks, and monitoring resource usage. It also supports features like role-based access control to manage permissions and detailed activity logs.

MLFlow[11] is an open-source platform designed to manage the machine learning lifecycle. It is used in this dissertation to track experiments, manage models, and deploy trained classifiers to

[6] https://www.atlassian.com/software/jira
[7] https://www.docker.com/
[8] https://www.jenkins.io/
[9] https://www.traefik.io
[10] https://www.portainer.io/
[11] https://www.mlflow.org/

Feed.UVL. For this, it provides several functionalities to log experiments, including all parameters, metrics, and model files created during the training and evaluation of a classifier.

## 2.5. Training & Evaluation Datasets

This section explains all of the datasets, which were manually created for the automatic feedback requirements relation and usage information extraction classifiers. A total of seven datasets were created covering five different software products. Section 2.5.2 introduces the feedback requirements relation datasets. Section 2.5.3 introduces the usage information classification datasets. All datasets used in this dissertation can be accessed through the open-source Github repository for Feed.UVL[12].

### 2.5.1. SmartAge Project

A large part of the feedback data used in this dissertation was gathered in the context of the SmartAge project (Radeck and Paech, 2024). In this project, older German adults above the age of 67 were introduced to SmartVernetzt (translated: "SmartConnected"), which was designed to improve their daily lives. This app provides information about leisure activities and health-related topics. A second app, called SmartFeedback, then allowed users to send feedback about both SmartVernetzt and SmartFeedback itself. Users could answer questions about the app or freely send feedback messages.

We chose to use the SmartAge dataset because it offered the unique ability to analyze feedback resulting from specific questions posed directly to the software users. This characteristic is not present in any of the other datasets used in this dissertation. Also, proximity to the project's organizers and developers gave us access to both the feedback data and the requirements for the developed apps. However, it should be noted that the SmartAge data only became available towards the later stages of research on this dissertation. Thus, it was not initially present when work on the usage information classification and feedback requirements relation classifiers was started.

### 2.5.2. Feedback Requirements Datasets

To train and evaluate our FeReRe classifiers, we manually created three datasets based on feedback and requirements from three different apps: Komoot, SmartVernetzt, and SmartFeedback. We also include another dataset (ReFeed) from an independent publication (Kifetew et al., 2021) to validate our classifier further. Each dataset consists of requirements for a specific software, feedback from real users about the software and a ground truth in which the feedback is related to the requirements.

Our three ground truths were created by manually examining every feedback statement and relating it to all requirements addressed in the feedback. Two annotators examined every feedback statement in each dataset and independently performed the relation to requirements. Afterwards, all disagreements regarding feedback requirements relations between the two annotators were resolved through discussion. This created the interrater agreement, which guarantees a high-quality ground truth for training and testing.

The FeReRe Komoot dataset comprises feedback statements crawled from the Google Play Store[13] about the popular hiking app Komoot. We excluded feedback shorter than 25 characters because it is unlikely to provide meaningful information. We also removed all links and emojis from the feedback. We could not acquire the initial requirements for Komoot as it is a commercial

[12]https://www.github.com/feeduvl
[13]https://www.play.google.com/store/apps/details?id=de.komoot.android

product. Instead, we recreated the requirements in detail by specifying every function and view provided by Komoot. We specified these requirements using the Task-oriented Requirements Engineering framework (TORE) (Paech and Kohler, 2004) in the ITS Jira. Thus, our dataset specifies all functionalities of Komoot as *System Function* requirements. Figure 2.9 shows an example for such a *System Function* in the form of the "Start Navigation function of the Komoot app. The activities users use the app for in their daily lives are captured in so-called *Sub-Task* requirements. The user interface of Komoot is captured in *Workspace* requirements. These three types of requirements comprise the 79 requirements in the provided dataset.

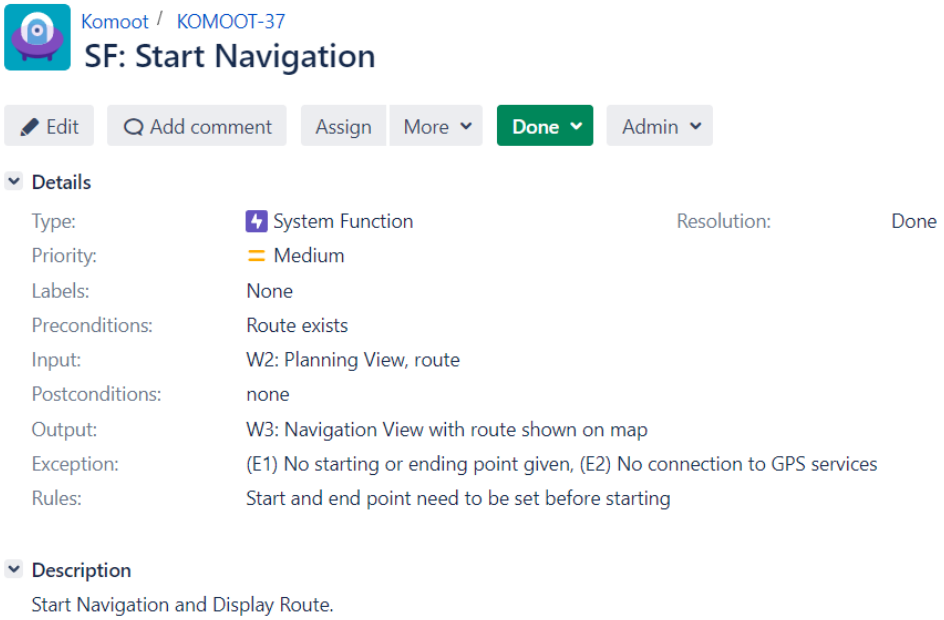



Figure 2.9.: SF: Start Navigation for Komoot App

The FeReRe SmartVernetzt and SmartFeedback datasets were gathered in the context of the SmartAge project (Section 2.5.1). Like Komoot, we excluded any feedback shorter than 25 characters and removed emojis from the text. To match the language of our other datasets, the German feedback was then translated into English using DeepL[14]. We then checked and corrected all automatic translations manually. Very few translations contained any errors apart from typos in the original German statements. The requirements for both apps were created by the app developers themselves, who also used the TORE framework to document the requirements. Figure 2.10 shows an example sub-task for the SmartVernetzt app.

The FeReRe ReFeed dataset was created as part of an independent study by Kifetew et al. (Kifetew et al., 2021). The software for this dataset supports home energy management by allowing users to monitor and analyze their home's energy consumption. The requirements for this software are documented as single sentence statements following the structure *"The system shall... <Functionality>"*. One example of this is the requirement: *"The system shall allow the user to calculate additional meters related to water and electricity consumption"*. More information on the creation of the dataset can be found in (Kifetew et al., 2021). We include this dataset to further evaluate our classifier on more independently created data. Additionally, it allows us to evaluate the performance on requirements not specified according to TORE.

Table 2.4 provides statistics about the number of feedback statements and requirements and the total number of sentences for each dataset. As shown in the table, the datasets have

[14]https://www.deepl.com/en/translator

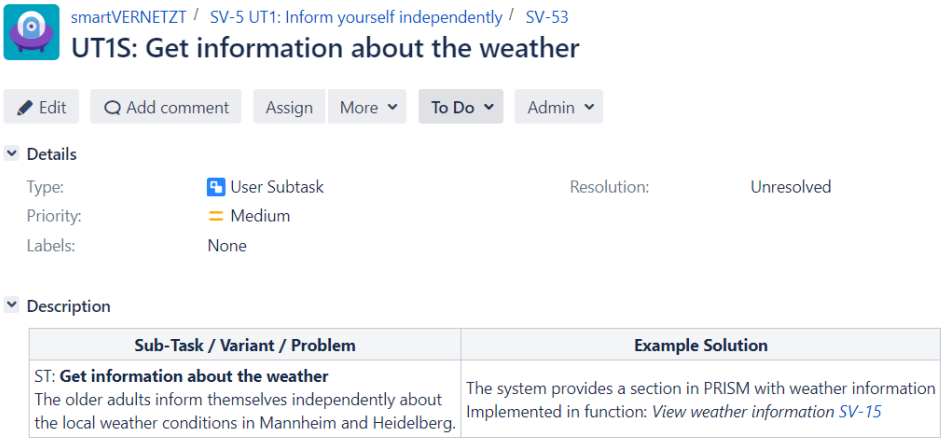



Figure 2.10.: UT1S: Get information about the weather for SmartVernetzt

Table 2.4.: Sizes of FeReRe Datasets

| Dataset | Feedback | Feedback Sentences | Requirements | Requirements Sentences |
|---|---|---|---|---|
| FeReRe_Komoot | 335 | 1485 | 79 | 218 |
| FeReRe_SmartVernetzt | 527 | 723 | 31 | 145 |
| FeReRe_SmartFeedback | 549 | 846 | 29 | 125 |
| FeReRe_ReFeed | 60 | 261 | 14 | 14 |

some noticeable differences. The average number of sentences per feedback statement is almost three times larger in Komoot and ReFeed than in SmartVernetzt and SmartFeedback. This is because the targeted feedback gathered for SmartVernetzt and SmartFeedback through the SmartFeedback app is more precisely focused on individual functionalities of the software. This is a result of the questions posed to users. This type of feedback is called *pull feedback* because it is gathered through questions actively posed to the users. Komoot and ReFeed, on the other hand, have feedback which is given by users without specific prompts. This is referred to as *push feedback*.

Also, the average number of sentences per requirement is higher on average in our datasets than in ReFeed. ReFeed consists only of single-sentence requirements, while TORE describes requirements in more detail. It should be noted that the ReFeed dataset does not contain all requirements of the software but only those provided by the original paper. We create datasets that fully specify each software.

For each dataset, a kappa value was calculated using Cohen's Kappa. Across the entire SmartVernetzt dataset, annotators reached a Cohen's Kappa of 0.92, while the kappa for the SmartFeedback dataset was even higher at 0.95. Annotators reported that the agreement for these two datasets was mostly due to the nature in which the feedback was collected through SmartFeedback. More than 90% of the feedback was received as answers to specific questions posed to the users. These questions made it much simpler for the annotators to relate a requirement to the feedback. The Cohen's Kappa for the Komoot dataset was much lower at 0.76. For Komoot no questions were posed, as the feedback came directly from the app store. Also, the number of requirements was much higher, with 79 requirements compared to 31 and 29 for SmartVernetzt and SmartFeedback, respectively, as can be seen in Table 2.4.

### 2.5.3. Usage Information Datasets

For our goal of studying usage information classification, we could not use any existing datasets. Over the course of multiple research projects, we had different opportunities to collect feedback data, which resulted in the creation of four different datasets. All come from different user feedback sources, allowing us to investigate each source's unique properties and transferability across feedback sources.

We created datasets to evaluate classifiers' performances for three different granularities of usage information classification using the TORE framework: sentence-based TORE Level classification, word-based TORE Level classification and word-based TORE Category classification. More details on the different granularities can be found in Section 2.2 and 6.1.

For the sentence-based TORE Level classification, data gathered from the feedback app SmartFeedback was used. This is partially the same feedback as is used for the feedback requirements relation datasets but includes more feedback statements in order to provide enough sentences for the classifiers' training. For the word-based classification, an online questionnaire, app reviews and an online forum were used to gather feedback. A significant amount of manual effort was required to label these datasets manually. Because of this, we could not manually label every dataset for every granularity. These datasets can also be accessed through the Feed.UVL Github repository[15]. Table 2.5 shows information on all four datasets.

Table 2.5.: Usage Information Dataset Characteristics

| | **Sentences** | **Words** | **Source** | **Software** | **Characteristics** | **Coding** |
|---|---|---|---|---|---|---|
| UIC_SmartAge | 3832 | 30706 | Feedback App | SmartFeedback & SmartVernetzt | Users' answers to developers questions | Sentence-based TORE Level |
| UIC_Prolific | 1146 | 26607 | Questionnaire | Komoot | Users' descriptions, likes, dislikes and ideas for improvement of a software | Word-based TORE Categories |
| UIC_Forum | 865 | 13775 | Reddit | Komoot, VLC, Chrome | 98 posts and their comments | Word-based TORE Categories |
| UIC_App Review | 901 | 14879 | Play Store | Komoot | 200 randomly crawled App Reviews | Word-based TORE Categories |

The dataset used for the sentence-based classification was gathered in the context of the SmartAge project (Section 2.5.1). Accordingly, we refer to this as the **UIC SmartAge** dataset. From this data, feedback collected over 3 1/2 months was then manually coded. This set consists of 3831 sentences, constituting a total of 1821 feedback messages. More information on the data, including questions asked to users, can be found in (Radeck and Paech, 2024). The data allows us to analyze whether including these questions as a classification feature improves usage information classification. We did not perform word-based classification on the SmartAge dataset because, as Section 6.2 will show that more training data is not necessary, and the additional coding would have involved another 70 hours of work for each coder involved. Given the fact that the SmartAge data only became available towards the later stages of the research on this dissertation, this was not feasible.

Since the project participants are older German adults, their feedback was also given and then coded in German. However, as classifiers are often trained on purely English data and monolingual mostly outperform their multilingual counterparts (Wu and Dredze, 2020)(Rönnqvist et al., 2019), the data was then translated into English using DeepL[16]. We then checked and

[15]https://www.github.com/feeduvl
[16]https://www.deepl.com/en/translator

corrected the automatic translations. The translated feedback was the same data also used for the FeReRe SmartAge datasets. For all automatic classification experiments, the English data was then used. Other than the translation, the only other pre-processing of the data was the removal of emojis.

The results of the coding for UIC SmartAge data are listed in Table 2.6. As can be seen, the dataset is very imbalanced, leaning heavily towards the *Domain Level*. As can be expected from older adults, the System Level was only rarely used at all. The *Interaction Level* also shows only about a quarter of the occurrences of the *Domain Level*, indicating that older adults tend to discuss the SmartAge software more towards the effects it has in their real life rather than specific interactions with the software. 1144 sentences were not assigned to any level. These sentences were often short answers stating, for example, that a user hadn't tried a functionality which they were asked about (e.g. *"I can't say yet"*) or talking about the context of the SmartAge project rather than the apps themselves (e.g. *"I have sent you a message that I will be travelling for the next week."*).

Table 2.6.: SmartAge dataset Number of Sentences assigned to each Code

| No Level | Domain Level | Interaction Level | System Level |
|---|---|---|---|
| 1144 | 2062 | 565 | 61 |

The datasets for the word-based classification are called the **UIC Prolific**, **UIC Forum** and **UIC App Review** dataset. The only pre-processing performed on the datasets was the removal of web links and emojis. Table 2.7 lists the number of words in each dataset assigned to each TORE Category code.

Table 2.7.: Number of Words Assigned to each TORE Category Code

| | Domain Data | Stake-holder | Activity | Task | Inter-action | Interaction Data | System Function | Work-space | System | No Category |
|---|---|---|---|---|---|---|---|---|---|---|
| **UIC_Prolific** | 1097 | 385 | 350 | 249 | 1048 | 952 | 482 | 81 | 749 | 21.214 |
| **UIC_Forum** | 280 | 33 | 85 | 1 | 572 | 443 | 5 | 345 | 678 | 11.333 |
| **UIC_App Review** | 363 | 77 | 113 | 69 | 844 | 746 | 325 | 90 | 420 | 11.832 |

The Prolific dataset was created through an online survey on the crowd platform Prolific[17]. A total of 100 participants took part in the survey. They were asked to answer four main questions about the Komoot[18] hiking app. Namely, "What is the Komoot App and what can the Komoot App do?", "What do you like about Komoot?", "What do you dislike about Komoot?" and "What could be improved about Komoot and why?". The survey was part of a prior research project. More information on the dataset can be found in (Anders et al., 2023).

The UIC Forum dataset consists of 98 threads, including comments crawled from the Reddit online forum. The threads were randomly sampled from the Google Chrome (40 threads), Komoot (20 threads), and VLC Video Player (38 threads) sub-forums on Reddit.

The UIC App Review dataset was the same feedback as the FeReRe Komoot dataset, Feedback crawled from the Google Play Store about the hiking app Komoot.

The coding process was the same for all four datasets. The datasets for the word-based classification were coded using Feed.UVL's custom coding tool, which is explicitly designed for TORE annotation (Chapter 7). The tool helped to reduce the large time cost of manual annotation on a word level and provided functionalities for easier interrater agreement. The dataset for the sentence-based classification was coded using Microsoft Excel. Multiple annotators

[17]https://www.app.prolific.com/
[18]https://www.play.google.com/store/apps/details?id=de.komoot.android

coded each part of a dataset. Coding was done in multiple steps. Each annotator coded a portion of a dataset, and then the annotators met to resolve all disagreements between annotations. After all disagreements were resolved, coding rules were improved, after which another portion of the dataset was coded. Note that there was no separate coding for word-based TORE Levels and TORE Categories because of the high amount of manual effort required when performing word-based coding. Instead, coders coded TORE Categories on the word level, after which TORE Levels were abstracted from the categories.

This means that for the word-based TORE Level dataset, all categories on each level (e.g. *Task, Activity, Domain Data and Stakeholder on the Domain Level* were transformed into the corresponding TORE Level. In total, three PhD and four master's students participated in the coding, with the author of this dissertation participating in each coding process.

In total, four PhD candidates and four master students participated in the coding, with the author of this dissertation participating in each coding process. Forum, App Review and SmartAge were annotated by two coders. Prolific was annotated by three coders. Except for the author of this dissertation, none of the coders were familiar with TORE other than through an introductory software engineering lecture some of them had attended. Every coder received a one-on-one introductory presentation, including an introduction to TORE, example codings, and a tool presentation. This was then followed by an interactive coding session, where participants coded example feedback, explained their reasons for assigning codes and discussed these with the author of this dissertation.

For each dataset, a kappa value was calculated using Brennan & Prediger Kappa (Brennan and Prediger, 1981). These can be seen in Table 2.8. We chose Brennan & Prediger partially due to constraints in our initially used coding software, MaxQDA. However, after switching to a different tool, we continued to use it due to its assumption of marginal distributions, which helps reduce the risk of the kappa paradox, where values can be low despite relatively high agreement due to the imbalanced nature of our datasets.

The kappa values for all three datasets where TORE Categories were assigned to words range from 0.59 to 0.65. Annotators found that the uncertainty over which subsequent words to assign codes to and which not significantly increased the number of disagreements, thus influencing the kappa values. In contrast, disagreements regarding sentence-based TORE Level coding were very low, reaching a kappa of 0.96. A contributing factor to this very high agreement was that there was no ambiguity over which segment to assign a code to compared to the word-based coding. Every part of a user's feedback was assigned either one of the three levels or the "0" category, indicating that the sentence did not contain any usage information. This, along with the reduced number of categories (3 levels compared to 9 categories) and the clear distinction between the levels, made it much easier for coders to agree. Meanwhile, for the word-based coding, different domain actions like *Activity* and *Task*, for example, were much more challenging to differentiate in users' statements due to the often ambiguous nature of user feedback.

Additionally, we calculated the precision and recall of each coder. To do this, we used the part of the human coders' annotations that was created with the final, revised coding rules. This was the last 50% of each dataset. We compared each coder's annotation to the ground truth after the interrater agreement had been done to calculate the metrics and averaged them over all coders for each dataset. This was done to compare human performance to the classifiers during treatment validation (Chapter 9). Table 2.8 shows the average precision and recall for each dataset across all human coders.

Table 2.8.: Brennan & Perediger Kappa, Average Precision and Average Recall of human coders

| dataset | Kappa | Avg. Precision | Avg. Recall |
|---|---|---|---|
| **UIC_SmartAge** | 0.96 | 0.98 | 0.97 |
| **UIC_Prolific** | 0.59 | 0.86 | 0.83 |
| **UIC_Forum** | 0.62 | 0.88 | 0.83 |
| **UIC_App Review** | 0.65 | 0.88 | 0.85 |
| **Average for Word-based** | 0.62 | 0.87 | 0.84 |

The four datasets differ in key ways, particularly in user intent, data characteristics, and structural organization. The online questionnaire dataset captures responses from users in a controlled, structured environment motivated by clear purposes, resulting in consistent, purpose-aligned data. In contrast, app store feedback reflects spontaneous user input, often emotionally charged and highly variable, aimed directly at app developers. Online forums provide yet another perspective, with community-driven discussions focused on problem-solving, experience sharing, and debates. This leads to unstructured, conversational data. Meanwhile, dedicated feedback apps blend characteristics of structured questionnaires and open-ended feedback, offering a mix of structured and spontaneous user input. These differences in data characteristics are compounded by variations in language style and structural organization. Forum feedback, for example, is more conversational and informal, often using colloquialisms and emotionally charged language, unlike the structured tone seen in other datasets. Additionally, forums feature nested discussions where context builds across posts, creating challenges for classifiers trained on standalone comments from other datasets.

Using these diverse datasets provides a more comprehensive evaluation for usage information classifiers across different contexts. Each dataset represents a distinct form of user input. This diversity can strengthen the classifier's robustness by ensuring it can handle both structured and unstructured data, formal and informal language, and varying levels of user intent clarity. However, these differences also introduce challenges, such as the need for classifiers that can generalize across disparate data sources. Despite these challenges, the inclusion of multiple datasets ultimately enhances the framework's applicability in real-world scenarios, as feedback rarely comes from one homogenous source.

# Part II.

# Solution Investigation

Chapter 3

# State of the Art - Software Artifact Relation

This mapping study was mainly conducted during a master thesis (Thakur, 2025), which this dissertation's author closely supervised. Research questions and synthesis criteria were given to the student by the supervisor, and the selection of papers was closely monitored. For a more thorough explanation of the search procedure and a detailed synthesis, please refer to the master thesis. This chapter will provide short explanations of the study design (Section 3.1), give an overview of the found publications (Section 3.2), discuss the threats to the validity of the mapping study (Section 3.3), and then draw conclusions for this dissertation (Section 3.4).

## 3.1. Study Design

Section 3.1.1 provides the research questions for the mapping study. Section 3.1.2 roughly explains the procedure for identifying relevant literature.

### 3.1.1. Research Questions

The main research question for this mapping study is focused on furthering knowledge goal 1 and is further refined into five sub-questions.

**RQ1:** Which approaches exist to relate different software artifacts with one another?

- **RQ1.1:** Which software artifacts are related by the approaches?
- **RQ1.2:** Which machine learning model is used to perform the relation?
- **RQ1.3:** How and on which data is the model trained for the relation?
- **RQ1.4:** Which pre-processing steps are necessary to use the approaches on their respective datasets?
- **RQ1.5:** How was the evaluation of the approaches conducted and what are the results of the evaluation in terms of metrics?

The main research question, **RQ1**, defines the focus to approaches related to knowledge goal 1. This mapping study aims to identify approaches that relate software artifacts to one another. We defined software artifacts as documents created by any stakeholder during software development or software maintenance. The artifacts we looked for were requirements, software specifications, bug reports, issue trackers and user feedback. We did not look for approaches relating source code, because it does not represent a purely natural language artifact. However, many of the approaches we found relate natural language artifacts to source code, which explains the presence of these approaches in the mapping study.

Sub-question **RQ1.1** identifies the different software artifacts the approaches use to relate to one another. Sub-question **RQ1.2** analyses which machine learning models are used by the different approaches to automate the relation process of two software artifacts. Sub-question **RQ1.3** analyses both how the identified models are trained and which data is used for this training process. Sub-question **RQ1.4** looks for any necessary pre-processing steps that the approaches perform. Lastly, sub-question **RQ1.5** analyses the evaluation methodology and results reported by the approaches.

### 3.1.2. Selection Procedure

Both literature studies discussed in this dissertation follow the guidelines laid out by (Kitchenham and Charters, 2007) for systematic mapping studies. A mapping study is different from a literature review. While the literature review identifies, analyzes and interprets all available evidence related to the research questions, the mapping study provides a broader overview (Petersen et al., 2008).

There are two ways of identifying relevant literature for a mapping study. The first is a search term-based search. Here, researchers begin with predefined keywords or phrases directly related to their research question. These terms are entered into databases or search engines to retrieve relevant papers. This method focuses on capturing a broad range of studies that contain or are tagged with these specific terms, allowing researchers to access a large volume of relevant literature quickly. By refining search terms, using operators like *AND, OR,* and *NOT*, and applying filters (e.g. publication date), researchers can narrow down results to more directly relevant studies.

The second method is Snowballing (Wohlin, 2016). This involves exploring references within previously identified relevant studies to uncover additional related sources. There are two types of snowballing: backward and forward. Backward snowballing refers to reviewing the reference list of a relevant study to find prior works that influenced it. Forward snowballing, meanwhile, involves finding newer studies that have cited the original article. This approach can help researchers uncover studies they may have missed through keyword searches alone.

The mapping study discussed here used both the search term-based search as a first step to identify relevant papers and the snowballing method to identify further approaches relevant to the research questions. For further details on the search terms, relevance criteria, and snowballing, refer to the relevant master thesis (Thakur, 2025). The search was performed in September 2024.

## 3.2. Overview of Publications

Using search terms and snowballing, the mapping study identified a total of 18 papers which deal with software artifact relation. It should be noted that one of these papers ((Lyu et al., 2023)) is a systematic literature review by Lyu et al., which contains another 40 approaches specifically dealing with the relation of issue tracker reports to other software artifacts. To keep the number of approaches manageable for the synthesis, the 40 approaches found by Lyu et al. are not discussed individually, but rather, the findings of the literature review as a whole are discussed. Table 3.1 provides a list of the 18 identified papers and the method through which they were discovered.

## 3.3. Threats to Validity

This section discusses the threats to the validity of this mapping study according to (Ampatzoglou et al., 2020).

Table 3.1.: Relevant Papers for Systematic Mapping Study of Software Artifact Relation

| ID | Title | Ref | Source |
|---|---|---|---|
| P1 | Recovering Traceability Links between Release Notes and Related Software Artifacts | (Nath et al., 2024) | Search term |
| P2 | Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models | (Lin et al., 2021) | Search term |
| P3 | Traceability Support for Multi-Lingual Software Projects | (J. Liu et al., 2020b) | Search term |
| P4 | Using Frugal User Feedback with Closeness Analysis on Code to Improve IR-Based Traceability Recovery | (Kuang et al., 2019) | Search term |
| P5 | Analyzing Requirements and Traceability Information to Improve Bug Localization | (Rath et al., 2020) | Search term |
| P6 | Towards Automatically Localizing Function Errors in Mobile Apps with User Reviews | (Yu et al., 2023) | Search term |
| P7 | A Software Requirements Ecosystem: Linking Forum, Issue Tracker, and FAQs for Requirements Management | (Tizard et al., 2022) | Search term |
| P8 | A Systematic Literature Review of Issue-Based Requirement Traceability | (Lyu et al., 2023) | Search term |
| P9 | IRRT: An Automated Software Requirements Traceability Tool Based on Information Retrieval Model | (H. Zhang et al., 2022) | Search term |
| P10 | Recovering Semantic Traceability Between Requirements and Source Code Using Feature Representation Techniques | (L. Zhang et al., 2021) | Search term |
| P11 | Recovering Trace Links Between Software Documentation and Code | (Keim et al., 2024) | Search term |
| P12 | Improving the Effectiveness of Traceability Link Recovery Using Hierarchical Bayesian Networks | (Moran et al., 2020) | Search term |
| P13 | Traceability in the Wild: Automatically Augmenting Incomplete Trace Links | (Rath and Maalej, 2018) | Search term |
| P14 | Automating User-Feedback Driven Requirements Prioritization | (Kifetew et al., 2021) | Search term |
| P15 | Towards Semantically Guided Traceability | (J. Liu et al., 2020a) | Snowballing P1 |
| P16 | User Review-Based Change File Localization for Mobile Applications | (Zhou et al., 2020) | Snowballing P6 |
| P17 | Automatically Matching Bug Reports with Related App Reviews | (Haering and Nadi, 2021) | Snowballing P7 |
| P18 | Requirements Traceability Recovery for the Purpose of Software Reuse: An Interactive Genetic Algorithm Approach | (Hamdi et al., 2022) | Snowballing P4 |

The first threat is the validity of the study selection. As explained, the search was conducted by a master's student during their thesis (Thakur, 2025). This poses the threat that the student might have missed certain publications due to a lack of experience when performing literature searches. However, we tried to reduce this threat by providing the student with comprehensive guidelines and close supervision. All papers that showed relevance in their title were discussed with one of the master thesis supervisors to form the final selection of papers. The student also documented all searches and paper selections to allow proper supervision. Still, the threat of missing relevant publications is present in all mapping studies, and search terms might not cover every single relevant publication. The threat was further reduced by performing snowballing to cover additional literature that was missed by the search terms.

The second threat concerns the data validity. This threat stems from the fact that the results reported in the mapping study may be incorrect due to incorrect reporting in the primary papers. This threat is reduced by only including peer-reviewed publications in this mapping study, resulting in a high standard for included publications. However, the presence of the literature review by Lyu et al. (Lyu et al., 2023) could also influence the data validity. Reporting on the 40 individual approaches found by Lyu et al. would have exceeded the scope of this mapping study, as a proper synthesis of 57 approaches (40 by Lyu and 17 found by the master thesis) creates problems both in the appropriate visualisation and comparison of the approaches. Instead, the approaches found by Lyu et al. were reported as one paper, which influences the observations of the synthesis.

The third threat concerns the research validity of the study. The results of this mapping study might not be perfectly repeatable because other researchers may choose other works as relevant.

This threat is present in all mapping studies. However, the student kept a log of the complete research procedure to facilitate reproducibility as much as possible.

## 3.4. Conclusion

The mapping study referenced here provided an overview of software artefact relation approaches. This section draws conclusions from this mapping study for the treatment design and treatment validation of the feedback requirements relation performed in this dissertation based on the mapping study's research questions.

**RQ1.1: Which software artifacts are related by the approaches?** The systematic literature review by Lyu et al. (Lyu et al., 2023) focuses specifically on issue reports (i.e. bug reports) and the software artifacts they are related to. Lyu et al. identified 40 relevant approaches. They found that issue reports were most commonly related to commit messages (18 out of 40), with the relation to source code being the second most common (7). Roughly 75% of the papers (29) relate issue reports to purely natural language artifacts. Of the further approaches identified by the master thesis, only two handled purely natural language artifacts. The other approaches relate source code to other natural language artifacts. Out of the 31 approaches handling natural language artifacts (29 by Lyu et al. and two further approaches), only one relates user feedback to software requirements. Kifetew et al. (Kifetew et al., 2021) relate the feedback to requirements by establishing a domain ontology and leveraging this to calculate the similarity between the two artifacts.

Feedback requirements relation appears to not have been a major research focus in the past. However, the number of works identified in total that establish relationships between two software artifacts indicate that creating connections between software artifacts is of great value as it creates traceability during software development. This creates further motivation to pursue the task of feedback requirements relation.

**RQ1.2: Which machine learning model is used to perform the relation?** Looking at the machine learning methods employed by the 31 approaches relating natural artifacts, we did not find any clear trend. More than 20 different machine learning models are used by the approaches, with no clear favourite in terms of usage frequency. Kifetew et al. (Kifetew et al., 2021), the only approach performing feedback requirements relation, do not use a complex machine learning classifier. Rather, they identify concepts, capture synonyms for these using WordNet[1] and calculate the Jaccard similarity between concepts in the feedback and concepts in the requirements. However, given the need to establish domain ontologies and the low precision of their approach, our conclusion is to instead adopt a machine learning classifier-based approach. The reason for this is further laid out in Chapter 5. Because of the variety of the used models, we conclude that it is necessary to experiment with multiple models to identify the best-performing one.

**RQ1.3: How and on which data is the model trained for the relation?** Because of the lack of feedback requirements relation approaches and the plethora of different software artifacts being related, used data is hard to compare between approaches. Some software artifacts are easier to obtain than others. Feedback and bug reports, for example, can easily be crawled from online repositories and are more numerous because they are often documented by the users themselves. Other artifacts, such as requirements and design documents, are harder to obtain because they are documented by the developers and are often not publicly available.

Additionally, no conclusions can be drawn on the size of datasets needed, as these also vary drastically between approaches. The most closely related approach by Kifetew et al. uses very

[1] https://www.wordnet.princeton.edu/

small datasets of only 14 requirements and 60 feedback. However, they do not need the data to train a classifier and only use it for evaluation.

Our main conclusion from this is to utilize the small dataset that is publicly available by Kifetew et al. (Kifetew et al., 2021) while also creating our own, much larger datasets to facilitate the training of any machine learning classifiers. Larger datasets also allow a more thorough evaluation of any designed approach.

**RQ1.4: Which pre-processing steps are necessary to use the approaches on their respective datasets?** Usage of pre-processing methods is heterogeneous across approaches. Some approaches use a plethora of different methods to treat their data. Others do not appear to process their data much at all before training their classifiers. Because there is no consistency in which pre-processing methods should be used, our main conclusion is to establish a baseline classifier first and then experiment with different combinations of preprocessing methods commonly used, such as stop word removal. This allows us to identify the combination of methods that provide the best results for feedback requirements relation.

**RQ1.5: How was the evaluation of the approaches conducted, and what are the results of the evaluation in terms of metrics?** Most papers empirically validate their approaches compared to a previously established ground truth. They utilize different metrics for this but most commonly report precision, recall and F-measure, as is standard in most machine learning approaches. Following the majority of approaches, we also perform empirical validation of the classifier to evaluate effectiveness. For reasons already laid out in Section 2.3.8, we decided to report precision, recall and F2-measure.

Kifetew et al., the only approach performing feedback requirements relation, do not provide average performance metrics of their classifier for the relation task. However, calculating the averages of the precision and recall provided in their two studies leads to an average precision of 0.34 and recall of 0.86. These numbers, however, are not representative because of the small data size and differences in the two experiments conducted in their work. They can still serve as a general comparison value for our evaluation.

Chapter 4

# State of the Art - Fine-Grained Feedback Classification

This chapter reports on the findings of a mapping study conducted to further knowledge goal 2: Understanding the current state of the art regarding automatic fine-grained feedback analysis. The mapping study provides an overview of current approaches in scientific literature. Section 4.1 describes the study design. Section 4.2 reports the results of the mapping study. Section 4.3 discusses threats to the study's validity. Section 4.4 draws conclusions for this dissertation from the mapping study.

## 4.1. Study Design

Section 4.1.1 provides the research questions for the mapping study. Section 4.1.2 explains the procedure for identifying relevant literature.

### 4.1.1. Research Questions

The main research question for this mapping study is focused on furthering knowledge goal 2 and is further refined into five sub-questions.

**RQ1:** Which classification approaches exist that employ methods to classify natural language user feedback into known requirements engineering related classes of fine granularity?

- **RQ1.1:** What are the overall goals of these approaches?
- **RQ1.2:** Which predefined classes are used by these approaches to classify the feedback?
- **RQ1.3:** Which automatic methods are used by these approaches?
- **RQ1.4:** Which data sources and data sizes are used by these approaches in which steps?
- **RQ1.5:** How was the evaluation of these approaches conducted, and what are its results?

The main research question, **RQ1**, focuses on the selection of approaches related to knowledge goal 2. As the goal is to transfer the findings of the mapping study to usage information classification, we are looking for approaches that *classify natural language user feedback*. This excludes approaches which classify other natural language artifacts, such as requirements. The question also asks for *known requirements engineering related classes*. This means that related approaches must predefine the classes they use for the classification of the feedback, and these classes must be related to the requirements engineering domain. We are not looking for approaches

that extract their classes from the feedback, such as is often done in topic modelling, for example (Blei et al., 2003). Lastly, the question is further specified by asking for classes of fine granularity. We define *fine granularity classes* as those referring to what users are specifically talking about in their feedback. This goes beyond general requirements classes like "Feature" or "Bug" to include, for example, specific rationale users state in their feedback.

Sub-question **RQ1.1** identifies the overall goals of the approaches. With this, we want to identify which tasks the authors of the approaches want to tackle, especially to find out if any approaches tackle a goal closely related to usage information classification.

Sub-question **RQ1.2** aims to provide further insights into the specific classes the approaches are using. We want to find out not only how many classes these approaches are using but also the granularity of the classes. Additional insights can be found here, such as whether approaches use multiple tiers of classes, where one class has further sub-classes that can be classified in multiple steps.

Sub-question **RQ1.3** focuses on the methods employed in each classification approach. These may include machine learning models, NLP techniques, or deep learning (Section 2.3). Answering this question clarifies how each approach automates classification. This helps us identify which methods are commonly used for fine-grained classification of user feedback.

Sub-question **RQ1.4** aims to detail the types and sizes of datasets each approach uses across training, validation, and testing. Data sources may include online reviews, user feedback from app stores, issue-tracking systems, or custom datasets from specific software products (Section 2.1.3). Information on data size is also essential as it affects the generalizability of the classification model.

Sub-question **RQ1.5** focuses on the evaluation methods and the results reported for each approach. Specifically, we are looking for the common evaluation metrics precision, recall and F1 (Section 2.3.8) in order to compare the results of the approaches. This also sets a frame of reference for the performance of our fine-grained user feedback classification approach.

### 4.1.2. Selection Procedure

As explained in Section 3.1.2, the studies introduced in this dissertation are both systematic mapping studies. In the previous study on the relation of software artifacts (Chapter 3) we were able to perform a search-term based search first, after which we performed snowballing on the relevant papers. However, for this study on fine-grained user feedback classification approaches, we were unable to perform a search term-based literature search because the terminology within this field is highly inconsistent. Specifically, "fine-grained" is not an established term commonly used in papers, meaning it does not reliably appear as a keyword or in titles and abstracts. Conducting a comprehensive search would require us to identify and include every potential research objective or focus area related to "fine-grained" classes. This is both impractical and unlikely to yield consistent results. Consequently, a search term-based method is ineffective for this review, as it would miss relevant studies due to the varied and non-standardized language used in the literature.

Instead, we focused our efforts on backwards snowballing a comprehensive set of six existing literature studies that deal with crowd-sourced requirements and user feedback analysis. This search was mainly conducted from late 2022 to early 2023. The literature studies we investigated ((Wang et al., 2019), (T. Zhang and Ruan, 2020), (R. Santos et al., 2019), (Lim et al., 2021), (Khan et al., 2019), and (Dabrowski et al., 2022)) contained a total of 341 individual papers. The idea behind this was to utilize the search-term based search that these reviews had already performed in a more general context, like feedback analysis or automatic classification and narrow the selection of 341 papers to those related to our research question. This process of "outsourcing" the search-term based search to existing literature avoided the previously stated

Table 4.1.: Number of Overlapping Articles Cited by the Literature Reviews

| | **Wang et al.** | **Zhang et al.** | **Santos et al.** | **Lim et al.** | **Khan et al.** | **Dabrowski** |
|---|---|---|---|---|---|---|
| **Wang et al.** | | 9 | 12 | 11 | 6 | 23 |
| **Zhang et al.** | 9 | | 7 | 12 | 10 | 11 |
| **Santos et al.** | 12 | 7 | | 13 | 8 | 19 |
| **Lim et al.** | 11 | 12 | 13 | | 11 | 15 |
| **Khan et al.** | 6 | 10 | 8 | 11 | | 5 |
| **Dabrowski et al.** | 23 | 11 | 19 | 15 | 5 | |

problems of designing search terms. Table 4.2 lists the six literature studies, along with their research questions, the search terms they used, the years in which they searched for literature and the number of papers they found.

Table 4.1 shows the number of overlapping articles found by the relevant literature studies. Wang et al., for example, cite 23 articles which are also cited by Dabrowski et al. As can be seen, there is considerable overlap between the studies. In total, of the 341 papers in the six literature studies, 273 only appear in one of the studies. 68 articles are cited by two or more literature studies.

To identify which of the 341 papers were relevant to our research questions, we defined a set of criteria of relevance, all of which had to be met for the papers to be included in this mapping study. These were:

**CoR1:** The article must analyze users' explicit software feedback.

**CoR2:** The article must present an approach for (semi-) automatic classification.

**CoR3:** The article must be related to requirement engineering.

**CoR4:** The article must use predefined classes.

**CoR5:** The article must use fine-grained classes

As can be seen, the criteria are analogous to the research questions. This ensured that the articles were both relevant and able to provide answers to our research questions. **CoR1** restricts articles to those dealing with explicit feedback about software. This meant excluding articles that handled implicit feedback as well as those dealing with feedback not related to software, such as product reviews. **CoR2** removed articles that did not introduce semi-automatic or fully automatic classification approaches. Manual approaches or meta-studies, which reported on the characteristics of user feedback, for example, were excluded. **CoR3** excluded articles that were not directly related to requirements engineering. This criteria excluded articles that analyzed user feedback for the purposes of, for example, content moderation (i.e. detecting spammers or fake accounts). **CoR4** and **CoR5** had the purpose of restricting the selection to articles which use both predefined and sufficiently fine-grained classes. As previously mentioned, we defined fine-grained classes as those that analyze what users are specifically talking about with regard to the software or their rationale behind the feedback.

Table 4.2.: Literature Studies used for Snowballing

| | Research Questions | Search terms | Search date | Papers Found |
|---|---|---|---|---|
| **(Wang et al., 2019) A systematic mapping study on crowdsourced requirements engineering using user feedback** | RQ1 What sources of implicit and explicit crowdsourced user feedback have been reported in RE activities according to published literature?<br>RQ2 What metadata of crowdsourced user feedback are reported in published literature as being useful for RE?<br>RQ3 In which RE activities has the crowdsourced user feedback been applied?<br>RQ4 What are the demographics of the research on applying crowdsourced user feedback for crowd-based RE according to published literature? | (ALL (requirements) AND TITLE (user OR app OR software) AND TITLE (review OR comment OR feedback)) AND (TITLE (requirements) OR TITLE (crowd OR crowdsourced OR crowdsourcing OR data-driven)) | 2006-2017 | 44 |
| **(T. Zhang and Ruan, 2020) The challenge of data-driven requirements elicitation techniques** | RQ1: What is the state of the art in data-driven requirements elicitation?<br>RQ2: What are the challenges with data-driven requirements elicitation?<br>RQ3: What is the improvement solution for a particular techniques' challenge? | Snowballing | 2007-2018 | 44 |
| **(R. Santos et al., 2019) An Overview of User Feedback Classification Approaches** | RQ1: Which automated techniques have been used in research to classify requirements-relevant content in user feedback?<br>RQ2: Which classification algorithms and features have been used most often?<br>RQ3: Which algorithm-feature pairs have yielded the highest precision and recall values? | (("CrowdRE" OR "Crowd RE") OR ((("User Review" OR "User Feedback" OR "App Review" OR "Feature Requests" OR "User Opinions" OR "User Requirements")) AND (Classif* OR Framework OR Tool OR "Text Analysis" OR Mining OR "Feature Extraction") AND "Requirements Engineering")) | 2013-2018 | 43 |
| **(Lim et al., 2021) Data-Driven Requirements Elicitation: A Systematic Literature Review** | RQ1: What types of dynamic data are used for automated requirements elicitation?<br>RQ2: What types of techniques and technologies are used for automating requirements elicitation?<br>RQ3: What are the outcomes of automated requirements elicitation? | Requirements elicitation: "Requirements elicitation" OR "requirements analysis" OR "requirements identification" OR "requirements discovery" OR "requirements gathering" OR "requirements determination" OR "requirements collection" OR "requirements engineering" OR "system requirements"<br>Automation: Automat* OR "computer aided" OR "computer assisted"<br>Big Data sources and related analytics: "Big data" OR sensor* OR "Internet of Things" OR IoT OR "natural language processing" OR "data mining" OR "artificial intelligence" OR "data processing" OR "data science" OR "data analysis" OR "machine learning" OR "data driven" OR "data oriented" OR "graph analytics" | 2012-2020 | 51 |
| **(Khan et al., 2019) Crowd Intelligence in Requirements Engineering: Current Status and Future Directions** | RQ1: What are the current foci of CrowdRE research?<br>RQ2: How traditional RE activities are mapped with CrowdRE activities and how crowd-based techniques support RE activities?<br>RQ3: What is a possible future role of intelligence in CrowdRE? | ("CrowdRE" OR "Requirements Crowdsourcing" OR ("Crowd" AND "Requirements Engineering") OR "Crowd-based Requirements Engineering" OR ("Crowd intelligence" AND "Requirements Engineering") | 2010-2018 | 97 |
| **(Dabrowski et al., 2022) Analysing app reviews for software engineering: a systematic literature review** | RQ1: What are the different types of app review analyses?<br>RQ2: What techniques are used to realize app review analyses?<br>RQ3: What software engineering activities are claimed to be supported by analysing app reviews?<br>RQ4: How are app review analysis approaches empirically evaluated?<br>RQ5: How well do existing app review analysis approaches support software engineers? | ('app review mining' OR 'mining user review' OR 'review mining' OR 'review analysis' OR 'analyzing user review' OR 'analyzing app review') AND ('app store')<br>('app review' OR 'user review' OR 'app store review' OR 'user feedback') AND ('software engineering' OR 'requirement engineering' OR 'software requirement' OR 'software design' OR 'software construction' OR 'software testing' OR 'software maintenance' OR 'software configuration' OR 'software development' OR 'software quality' OR 'software coding') AND ('app store') | 2010-2020 | 182 |

Table 4.3.: Number of Papers matching the criteria of relevance

| Criterium | Wang et al. | Zhang et al. | Santos et al. | Lim et al. | Khan et al. | Dabrowski et al. | Unique Articles |
|---|---|---|---|---|---|---|---|
| **CoR1:** Software Feedback | 40 | 26 | 38 | 36 | 28 | 135 | 227 |
| **CoR2:** (Semi)-automatic | 21 | 17 | 29 | 25 | 12 | 65 | 144 |
| **CoR3:** Requirements Engineering | 13 | 12 | 25 | 14 | 11 | 38 | 67 |
| **CoR4:** Predefined Classes | 10 | 8 | 23 | 5 | 9 | 30 | 50 |
| **CoR5:** Fine-grained Classes | 3 | 0 | 7 | 3 | 3 | 6 | 12 |

Table 4.3 shows how many papers out of the 341 extracted from the literature studies match the criteria. The table lists both the number of papers per literature study as well as the total number of unique papers across all six studies. Note that because of the article overlap between the studies, the number of unique articles is lower than the sum of each row. Each criterium reduces the number of matching papers from the previous criterium. As can be seen, 114 papers did not deal with explicit software feedback. Of the remaining 227 papers, 144 provided some form of semi- or fully-automatic approach, meaning that 83 did not. 67 of the 144 were related to requirements engineering. Of these, 50 use predefined classes, while only 17 do not. Of the 50 approaches, 38 used classes that were not of sufficiently fine granularity. Some examples are approaches classifying feedback into functional and non-functional requirements (e.g. (H. Yang and Liang, 2015)) or into bug reports, suggestions or an "other" class (e.g. (Villarroel et al., 2016)). In total, out of the 341 papers, we identified 12 that matched all criteria of relevance. Out of the six studies, (T. Zhang and Ruan, 2020) is the only one that did not yield a relevant paper.

To further our selection, we then performed another round of snowballing on these 12 papers, applying the same criteria. We performed both backwards and forward snowballing for these papers. This resulted in another nine papers that matched our criteria of relevance. These papers were not included in the literature studies because they were all published after 2020. As can be seen in Table 4.2, the literature studies performed their searches no later than 2020. In total, we were able to identify 21 papers which perform (semi-)automatic classification of user feedback into predefined, fine-grained, requirements engineering related classes.

## 4.2. Results and Comparison

This section will report on the results of this mapping study and discuss the findings. Section 4.2.1 gives an overview of the identified papers, and Section 4.2.2 compares the approaches through synthesis.

### 4.2.1. Overview of Publications

Table 4.4 lists all 21 relevant papers for this mapping study and the sources through which the papers were found. This source can either be one or multiple of the six literature surveys we analyzed or one of the other papers found through the literature surveys. As mentioned before, we discovered 12 approaches through the six surveys we analyzed and another 9 through further snowballing of the 12 approaches.

Figure 4.1 shows an overview of the publication dates of the relevant approaches. Note that the six literature surveys we analyzed were published between 2019 and 2022, which could influence these numbers. The oldest publication found is by (Guzman et al., 2015). The timeline shows an uptick in 2017, followed by no publications in 2018. Afterwards, publications range between 1 and 4 in the following years.

Table 4.4.: Relevant Papers for Systematic Mapping Study of Fine-grained User Feedback Classification

| ID | Title | Ref | Snowballing Source |
|---|---|---|---|
| P19 | Analyzing reviews and code of mobile apps for better release planning | (Ciurumelea et al., 2017) | Literature Survey<br>(Wang et al., 2019)<br>(R. Santos et al., 2019)<br>(Dabrowski et al., 2022) |
| P20 | Automatic Classification of Accessibility User Reviews in Android Apps | (Aljedaani et al., 2022) | (Ciurumelea et al., 2017) |
| P21 | Ensemble Methods for App Review Classification: An Approach for Software Evolution | (Guzman et al., 2015) | Literature Survey<br>(Wang et al., 2019)<br>(R. Santos et al., 2019)<br>(Dabrowski et al., 2022) |
| P22 | Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews | (McIlroy et al., 2016) | Literature Survey<br>(Wang et al., 2019)<br>(R. Santos et al., 2019) |
| P23 | Fine-Tuning Pre-Trained Model to Extract Undesired Behaviors from App Reviews | (W. Zhang et al., 2022) | (McIlroy et al., 2016) |
| P24 | A BERT and Topic Model Based Approach to reviews Requirements Analysis | (J. Yang et al., 2021) | (McIlroy et al., 2016) |
| P25 | Requirements knowledge acquisition from online user forums | (Ali Khan et al., 2020) | Literature Survey<br>(Lim et al., 2021) |
| P26 | Can end-user rationale improve the quality of low-rating software applications: A rationale mining approach | (Ullah et al., 2022) | (Ali Khan et al., 2020) |
| P27 | Can a Conversation Paint a Picture? Mining Requirements In Software Forums | (Tizard et al., 2019) | Literature Survey<br>(Lim et al., 2021) |
| P28 | Which Feature is Unusable? Detecting Usability and User Experience Issues from User Reviews | (Bakiu and Guzman, 2017) | Literature Survey<br>(R. Santos et al., 2019)<br>(Lim et al., 2021)<br>(Khan et al., 2019)<br>(Dabrowski et al., 2022) |
| P29 | Automatic Classification of Non-Functional Requirements from Augmented App User Reviews | (Lu and Liang, 2017) | Literature Survey<br>(R. Santos et al., 2019) |
| P30 | A Practical User Feedback Classifier for Software Quality Characteristics | (R. d. Santos et al., 2021) | (Lu and Liang, 2017) |
| P31 | Evaluating pre-trained models for user feedback analysis in software engineering: A study on classification of app-reviews | (Hadi and Fard, 2023) | (Lu and Liang, 2017) |
| P32 | Classifying User Requirements from Online Feedback in Small Dataset Environments using Deep Learning | (Mekala et al., 2021) | (Lu and Liang, 2017) |
| P33 | UUX-Posts: a tool for extracting and classifying postings related to the use of a system | (Mendes and Furtado, 2017) | Literature Survey<br>(R. Santos et al., 2019) |
| P34 | Listening to the Crowd for the Release Planning of Mobile Apps | (Scalabrino et al., 2019) | Literature Survey<br>(R. Santos et al., 2019)<br>(Dabrowski et al., 2022) |
| P35 | Mining User Rationale from Software Reviews | (Kurtanović and Maalej, 2017) | Literature Survey<br>(Khan et al., 2019) |
| P36 | Cslabel: An approach for labelling mobile app reviews | (L. Zhang et al., 2017) | Literature Survey<br>(Dabrowski et al., 2022) |
| P37 | Mining non-functional requirements from app store reviews | (Jha and Mahmoud, 2019) | Literature Survey<br>(Dabrowski et al., 2022) |
| P38 | Extracting Arguments Based on User Decisions in App Reviews | (Kunaefi and Aritsugi, 2021) | (Jha and Mahmoud, 2019) |
| P39 | Empirical Evaluation of ChatGPT on Requirements Information Retrieval Under Zero-Shot Setting | (J. Zhang et al., 2023) | (Jha and Mahmoud, 2019) |

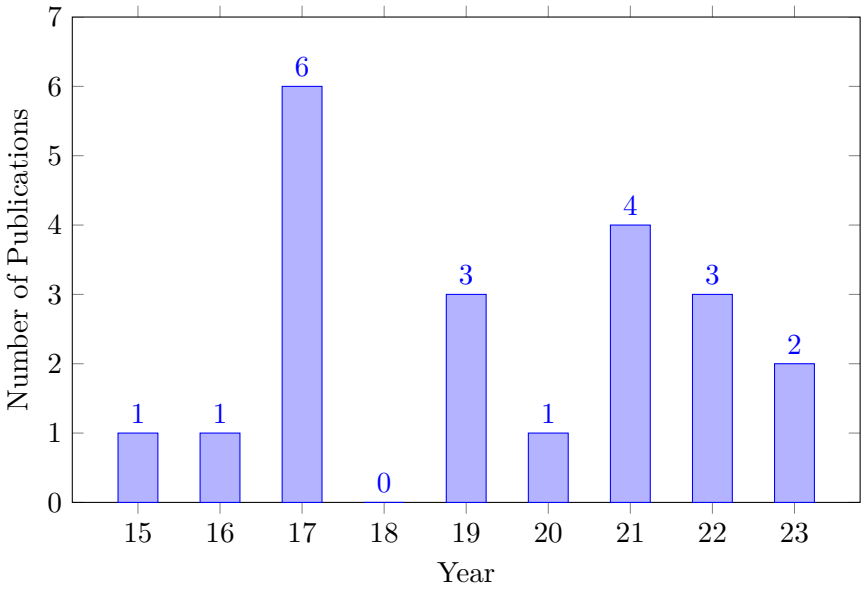



Figure 4.1.: Fine-grained Classification Approach Publications Per Year

### 4.2.2. Synthesis

The synthesis compares the relevant papers by introducing synthesis criteria derived from the research questions. Table 4.5 lists the criteria and research questions from which they were derived.

Table 4.5.: Relevant Papers' Evaluation Metrics and Results

| ID | Criterium | Relevant RQ |
|---|---|---|
| SC1 | Approach's Goal | RQ1.1 |
| SC2 | Number of Classes | RQ1.2 |
| SC3 | Used Classes | RQ1.2 |
| SC4 | Used Methods | RQ1.3 |
| SC5 | Methods per No. Classes | RQ1.2; RQ1.3 |
| SC6 | Dataset Sizes | RQ1.4 |
| SC7 | Dataset Source | RQ1.4 |
| SC8 | Evaluation Methodology | RQ1.5 |
| SC9 | Evaluation Metrics | RQ1.5 |
| SC10 | Evaluation Results | RQ1.5 |

**SC1** investigates the overall goal of each approach. This helps identify for which purposes the approaches were designed and compare similar goals across the different approaches. **SC2** compares the number of classes each approach classifies in the user feedback, while **SC3** compares the specific classes of the approaches with one another. **SC4** compares all NLP, ML and DL methods used by the approaches. The idea behind this is to find out which methods are most commonly used for fine-grained user feedback analysis. Building on this, **SC5** does not compare individual approaches but rather groupings of approaches. The approaches are clustered into clusters of similar numbers of classes, and the methods employed by these are investigated. The goal behind this is to see whether a correlation exists between the number of classes used

by an approach and the methods they employ to classify these classes. **SC6** compares the sizes of the datasets used by the approaches for training and testing, while **SC7** identifies the sources of these datasets. **SC8**, **SC9**, and **SC10** investigate the evaluations of the approaches. Their methodology, with regard to the creation of a gold standard, interrater agreement and cross-validation testing, is compared, as well as the metrics they use to report performance and their results.

The second column of Table 4.6 lists the overall goals of each approach (**SC1**). The goals can be grouped into four categories: Taxonomies, non-functional requirements (NFR), Rationale and Others. The most common goal category here is the classification into NFR, which seven approaches perform. Four of these approaches (P24, P29, P30, P37) perform a general classification of NFRs categories, while three (P20, P28, P33) focus on specific NFRs and their sub-categories. Five approaches (P19, P21, P22, P34, P36) classify custom taxonomies they introduce within their work. The taxonomies of P19 and P22 are custom-tailored to app review classification, while the taxonomies of P21, P34 and P36 are more generic for software feedback in general. Four approaches (P25, P26, P35, P38) classify user rationale to gain a better understanding of the users' reasoning in their feedback. Three approaches (P23, P27, P32) offer unique analysis goals. P23 classifies feedback into categories of undesirable behaviour, P27 classifies different types of information that can be found in online forums, and P32 classifies feedback according to how useful it is.

Two approaches (P31, P39) can not be sorted into the above categories because their overall goal is not to perform a specific classification but rather to evaluate the performance of specific classifiers for feedback analysis in general. P31 evaluates the performance of certain pre-trained models while P39 evaluates the performance of ChatGPT for classification. Both approaches use datasets from multiple other publications which have their own goals and use these to evaluate the performance.

Additionally, we see that only three approaches (P22, P4, P36) perform multi-label classification, where one feedback can be assigned multiple classes. The other 18 approaches all assign only one class per feedback. It should also be noted that none of the approaches we identified performed word-based analysis of the feedback (Section 2.1.2). Instead, all approaches either classified complete statements or individual sentences.

Looking at the third column of Table 4.6 (**SC2**), we see that the number of classes is very similar across the rationale approaches, with most using five classes and one using four. NFR approaches, on the other hand, can differ a lot, with five approaches using 4-5 classes and two using 23 and 26 different classes. Both of the latter are looking at specific NFR categories. Taxonomy approaches can also differ more widely, with two approaches using seven classes, one using 14 and two using 17 classes. Both approaches with the primary goal of evaluating classifiers use 16 classes, while the approaches in the *other* category use 23, 8 and 9 classes. On average, across the 21 approaches, the number of used classes is approximately 11.

The used classes in column four of Table 4.6 (**SC3**) show a lot of similarities, especially for the NFR approaches as these most often use standardized definitions for NFRs such as (ISO/IEC, 2010). P22 and P36, both classifying taxonomies, also use the same classes. P36, however, extends the taxonomy by three more classes.

Table 4.7 (**SC4**) lists all methods used by the 21 approaches in descending order of usage frequency. Most of the approaches utilize multiple methods to create their classifiers. In total 22 different methods are used, with 12 of these only being used by a single approach.

The most used method is a support vector machine, which 12 approaches utilize. This is followed by Naive Bayes which nine approaches use. "Newer" methods like Neural Network or transformer based models such as BERT are less frequent, placing only as the 7th and 6th mosed used methods respectively.

Table 4.6.: Relevant Papers' Goals and Classes (SC1, SC2, SC3)

| ID | Approach's Goal | No. Classes | Used Classes |
|---|---|---|---|
| P19 | Classify into app taxonomy | 17 | *High and Low-Level Taxonomy*<br>Compatibility into Device, Android version, Hardware<br>Usage into App usability, UI<br>Resources into Performance, Battery, Memory<br>Pricing into Licensing, Price<br>Protection into Security, Privacy |
| P20 | Classify into app accessibility categories | 4 | Principles, Audio/Video, Design, Focus |
| P21 | Classify into software evolution taxonomy | 7 | Bug report, Feature strength, Feature shortcoming, User request, Praise, Complaint, Usage scenario |
| P22 | Classify into custom app multi-label taxonomie | 14 | Additional Cost, Functional Complaint, Compatibility Issue, Crashing, Feature Removal, Feature Request, Network Problem, Other, Privacy and Ethical Issue, Resource Heavy, Response Time, Uninsteresting Content, Update Issue, User Interface |
| P23 | Classify into undesirable behavior categories | 23 | Ad disruption, App ranking fraud, App repacking, Bad performance, Drive-by download, Excessive network traffic, Fail to delete, Fail to exit, Fail to install, Fail to login or register, Fail to retrieve content, Fail to start, Hidden app, Illegal redirection, Illegal background behavior, Inconsistency between functionality and description, Payment deception, Permission abuse, Praise, Privacy of information leak, Virus, Vulgar content, Special |
| P24 | Classify into multi-label NFR categories | 5 | Usability, Dependability, Performance, Supportability, Miscellaneous |
| P25 | Classify into user rationale categories | 4 | Feature, Claim-supporting, Claim-attacking, Issue |
| P26 | Classify into user rationale categories | 5 | Claim-attacking, Claim-supporting, Claim-neutral, Decisions, Issues |
| P27 | Classify into different types of information found in online forums | 8 | Application usage, Non-informative, Apparent bug, Application guidance, Question on application, Help seeking, Feature request, User setup |
| P28 | Classify into usability and user experience categories | 23 | Memorability, Learnability, Efficiency, Errors, Satisfaction, Likeability, Pleasure, Comfort, Trust, Anticipation, Overall Usability, Hedonic, Detailed Usability, User Differences, Support, Impact, Affect and Emotion, Enjoyment and Fun, Aesthetics and Appeal, Engagement, Motivation, Enchantment, Frustration |
| P29 | Classify into NFR categories | 5 | Reliability, Usability, Portability, Performance, Others |
| P30 | Classify into NFR categories | 4 | Functional suitability, Performance, Compatibility, Usability |
| P31 | Evaluate pre-trained models for feedback analysis | 16 | Performance, Portability, Usability, Reliability, Usage scenario, Feature strength, User experience, Feature shortcoming, Inquiry, Problem, Rating, Bug report, Feature request, Aspect evaluation, Praise, Irrelevant |
| P32 | Classify into usefulness categories | 9 | Useless, Helpful<br>Helpful, Useless (again on previous helpful)<br>None, Feature, Stability, Quality, Performance |
| P33 | Classify into usability and user experience (UUX) categories | 26 | Posting related to Use (PRU); Non-PRU<br>PRU into Usability, UX or UUX<br>Usability into Efficacy, Efficiency, Memorability, Learnability, Safety, Utility<br>UX into Satsifaction, Affect, Trust, Esthetics, Frustration, Motivation, Usability, Pleasure, Anticipation, Impact, Hedonic quality, Comfort, Engagement, Enchantment, Accessibility, Support |
| P34 | Classify into taxonomie, cluster and prioritize reviews | 7 | Functional bug report, Suggestion for new feature, Report of performance problems, Report of security issues, Report of excessive energy consumption, Request for usability improvements, Other |
| P35 | Classify into user rationale categories | 5 | Issue, Alternative, Criteria, Decision, Justification |
| P36 | Classify multi-label issue type taxonomy | 17 | Additional Cost, Compatibility Issue, Content Complaint, Crashing, Feature Removal, Feature Request, Functional Complaint, Installation Issue, Network connection issue, Privacy and ethical issue, Property safety, Resource heavy, Response time, Traffic wasting, Update issue, User interface, Other |
| P37 | Classify into NFR categories | 4 | Dependability, Performance, Supportability, Usability |
| P38 | Classify into user rationale categories | 5 | Acquiring, Recommending, Requesting, Rating, Relinquishing decision |
| P39 | Evaluate ChatGPT classification for feedback analysis | 16 | Usability, Security, Operational, Performance, Dependability, Supportability, Business, Tool, Travel, Social, News, Navigation, Music, Life, Education, Entertainment |

Table 4.7.: Relevant Papers' Used Methods (SC4)

| | P19 | P20 | P21 | P22 | P23 | P24 | P25 | P26 | P27 | P28 | P29 | P30 | P31 | P32 | P33 | P34 | P35 | P36 | P37 | P38 | P39 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **SVM** | | X | X | X | | | X | X | | X | | X | | X | | | X | X | X | X | |
| **Naïve Bayes** | | | X | X | | | X | X | X | | | | | X | | | X | | X | X | |
| **Logistic Regression** | | X | X | | | | X | X | | | | X | | | | | X | | | X | |
| **Random Forest** | | X | | X | | | X | X | | | | | | | | X | | | | X | |
| **TF-IDF** | X | | | | | | | | | | X | X | | | | | | | X | | |
| **BERT** | | | | | X | X | | | | | | | X | X | | | | | | | |
| **Neural Network** | | | X | | | | | X | | | | | | | | | | | | X | |
| **Decision Tree** | | X | | | | | | | | | | X | | | | | | | | | |
| **Bag of Words** | | | | | | | | | | | X | X | | | | | | | | | |
| **Rule-Based Approach** | | | | | | | | | | X | | | | | | | | | X | | |
| **Regression Tree** | X | | | | | | | | | | | | | | | | | | | | |
| **Extra Tree Classifier** | | X | | | | | | | | | | | | | | | | | | | |
| **K-Nearest Neighbor** | | X | | | | | | | | | | | | | | | | | | | |
| **J48** | | | | X | | | | | | | | | | | | | | | | | |
| **CHI2** | | | | | | | | | | | X | | | | | | | | | | |
| **XLNet** | | | | | | | | | | | | | X | | | | | | | | |
| **FastText** | | | | | | | | | | | | | | X | | | | | | | |
| **ELMO** | | | | | | | | | | | | | | X | | | | | | | |
| **Pattern Matching** | | | | | | | | | | | | | | | X | | | | | | |
| **AdaBoost** | | | | | | | | | | | | | | | | | | | | X | |
| **XGBoost** | | | | | | | | | | | | | | | | | | | | X | |
| **ChatGPT** | | | | | | | | | | | | | | | | | | | | | X |

Table 4.8.: Relevant Papers' Overlap Between Number of Classes and Used Methods (SC5)

| **No. Classes** | **20+** | **10-19** | **6-9** | **5** | **4** |
|---|---|---|---|---|---|
| **Occurrences** | 3 | 5 | 4 | 5 | 4 |
| **Papers** | P23, P28, P33 | P19, P22, P31, P36, P39 | P21, P27, P32, P34 | P24, P26, P29, P30, P35 | P20, P25, P30 P37 |
| **Methods** | BERT<br>Pattern Matching<br>SVM | BERT<br>ChatGPT<br>J48<br>Naive Bayes<br>Random Forest<br>Regression Tree<br>SVM (3x)<br>TF-IDF<br>XLNet | BERT<br>ELMO<br>FastText<br>Logistic Regression<br>Naive Bayes (3x)<br>Neural Network<br>Random Forest<br>SVM (2x) | AdaBoost<br>BERT<br>Bag of Words (2x)<br>CHI2<br>Decision Tree<br>Logistic Regression (4x)<br>Naive Bayes (3x)<br>Neural Network (2x)<br>Random Forest (2x)<br>SVM (4x)<br>TF-IDF (2x)<br>XGBoost | Decision Tree<br>Extra Tree Classifier<br>K-Nearest Neighbor<br>Logistic Regression (2x)<br>Naive Bayes (2x)<br>Random Forest (2x)<br>SVM (3x)<br>TF-IDF<br>Rule-Based Approach |

Table 4.8 shows the comparison of approaches for **SC5**. When clustering the approaches by the number of used classes we see which methods are used for which number of classes. The approaches are clustered in such a way that relatively equal amounts of approaches (3-5) are present in each. There appears to be no clear correlation between the number of classes and the methods used to classify them. SVMs are used consistently across all clusters. Methods such as BERT and Random Forest are also used in four of the five clusters. The cluster with 20+ classes has a much lower variance of methods being used, with only three different methods being applied. However, it is also the smallest group with only three approaches.

Table 4.9 lists the size of each approach's dataset used for training and evaluation (**SC6**) as well as the sources of these datasets (**SC7**). Comparison of some dataset sizes is difficult because some approaches report the number of sentences (P24) or number of forum posts (P27), which does not directly correlate to the number of reviews, as reported by most of the approaches. Figure 4.2 further visualizes the number of reviews as a dot plot. For simplicity, all approaches are tracked as "Number of Reviews" in Figure 4.2. We see that most approaches have less than 5000 reviews in their datasets. Five of these have less than 2000. Ten approaches, however, have more than 5000 reviews, with three (P27, P31, P26) having 49000 or more. P38 uses the lowest number of reviews, with only 500. It should be noted that the approaches with 49000 or more reviews do not use the full size of their dataset for training or testing. They mostly create random samplings of reviews from the dataset. However, in the absence of actual numbers given by the approaches, the size of the complete dataset is listed here.

Looking at **SC7** in Table 4.9, we see that by far, the most common source for these reviews is app stores such as the Google Play Store[1] or the Apple App Store[2]. Fifteen approaches use reviews collected from this source. Three approaches use software reviews collected from the Amazon store. Reddit, Twitter, a dedicated review website, and dedicated online forums are each used by only one approach.

[1]https://www.play.google.com/store
[2]https://www.apple.com/app-store/

Table 4.9.: Relevant Papers' Dataset Size and Sources (SC6, SC7)

| **Paper** | **Dataset Size** | **Source** |
|---|---|---|
| P19 | 7754 Reviews | App Store |
| P20 | 2663 Reviews | App Store |
| P21 | 4550 Reviews | App Store |
| P22 | 7290 Reviews | App Store |
| P23 | 10358 Reviews | App Store |
| P24 | 6759 Sentences from Reviews | App Store |
| P25 | 3319 Forum Comments | Reddit |
| P26 | 77202 Reviews | Amazon |
| P27 | 49000 Forum Posts | Forums |
| P28 | 3491 Reviews | Review Website |
| P29 | 6696 Reviews | App Store |
| P30 | 1500 Reviews | App Store |
| P31 | 55933 Reviews | App Store |
| P32 | 1000 Reviews | App Store |
| P33 | 18545 Tweets | Twitter |
| P34 | 3000 Reviews | App Store |
| P35 | 1020 Reviews | Amazon |
| P36 | 3902 Reviews | App Store |
| P37 | 6000 Reviews | App Store |
| P38 | 500 Reviews | App Store, Amazon |
| P39 | 1800 Reviews | App Store |

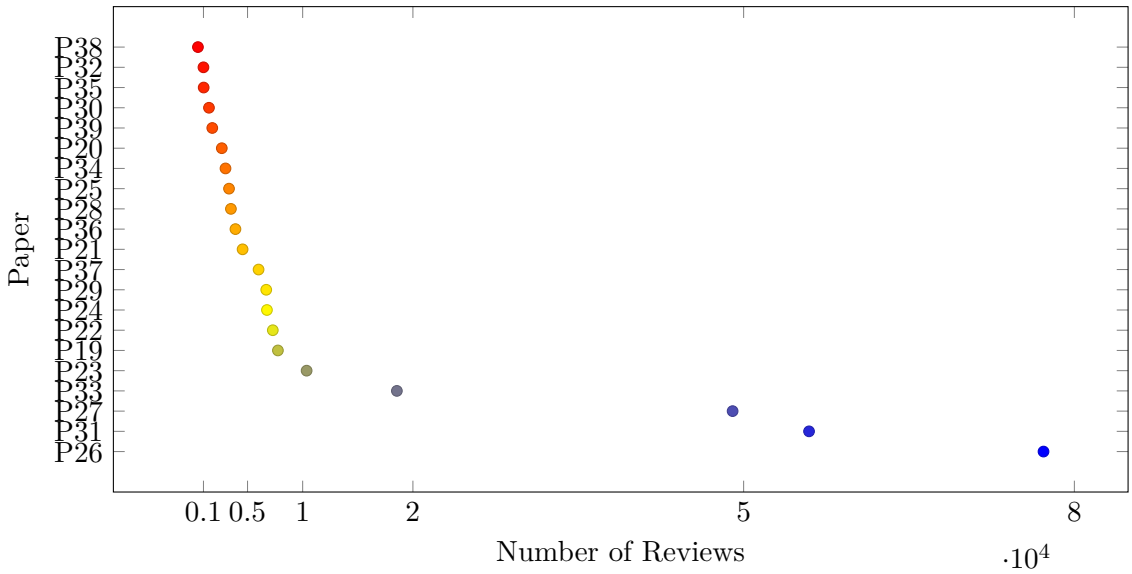



Figure 4.2.: Dot Plot of the Dataset Sizes per Relevant Paper (SC6)

Table 4.10.: Relevant Papers' Evaluation Methodologies (SC8)

| Paper | Manual Ground Truth | Interrater Agreement | Cross Validation |
|---|---|---|---|
| P19 | (X) | X | |
| P20 | X | X | |
| P21 | X | X | X |
| P22 | X | X | X |
| P23 | X | | |
| P24 | X | | |
| P25 | X | X | X |
| P26 | X | X | X |
| P27 | X | X | X |
| P28 | X | | |
| P29 | X | X | X |
| P30 | X | (X) | X |
| P31 | X | (X) | X |
| P32 | X | | |
| P33 | X | X | |
| P34 | X | X | X |
| P35 | X | X | X |
| P36 | X | X | X |
| P37 | X | X | |
| P38 | X | X | X |
| P39 | X | (X) | |

Table 4.10 shows the comparison for **SC8**, the evaluation methodology. For this, we investigate three factors: whether the authors of each approach manually created a ground truth to which they compare their classifier, whether they performed interrater agreement to create the ground truth and whether the classifiers were evaluated using k-fold cross validation. X marks an approach that used the respective methodology. (X) marks that the methodology was only partially applied.

We see that all authors manually created a ground truth or used a manually created ground truth from another source. P19 only does this partially because, while a manual ground truth was created, this was used to build the taxonomy and not evaluate the classifier. Instead, the classifier is evaluated by manually labelling its output as correct or incorrect.

Interrater agreement was done by 14 out of 21 approaches. Four approaches did not report performing any interrater agreement. In this case, their dataset or parts of it were labelled by only a single person with no overlap with another annotator. Three papers partially performed interrater agreement. Rather than having two people annotate the same data and then check for inconsistencies, these papers had the data annotated by one person while another checked those annotations to see if they agreed with them or not.

Cross validation was done by 12 papers. These were either 5- or 10-fold cross validations. The other 9 approaches used a split between their training and testing data. Most commonly 80% of the data was used for training of the classifier and 20% was used for testing. All papers made sure not to evaluate on the same data that the classifier was trained on.

The second column in Table 4.11 shows the metrics used in the papers to measure the performance of the classifiers (**SC9**). Most of the papers used the common metrics precision (P), recall (R) and F1 (see Section 2.3). Five approaches also used other categories to measure the performance. These depend on the goal of the approach and are thus not really comparable across the 21 relevant approaches. Two approaches (P30, P38) only provided F1 results without

Table 4.11.: Relevant Papers' Evaluation Metrics and Results (SC9, SC10)

| Paper | Evaluation Metrics | Precision (P) | Recall (R) | F1 Score |
|---|---|---|---|---|
| P19 | P, R, F1 | 0.89 | 0.99 | 0.94 |
| P20 | P, R, F1, Other | 0.97 | 0.99 | 0.98 |
| P21 | P, R, F1 | 0.71 | 0.62 | 0.64 |
| P22 | P, R, F1, Other | 0.65 | 0.64 | 0.64 |
| P23 | P, R, F1, Other | 0.75 | 0.74 | 0.75 |
| P24 | P, R, F1 | 0.70 | 0.65 | 0.66 |
| P25 | P, R, F1 | 0.64 | 0.59 | 0.54 |
| P26 | P, R, F1, Other | 0.98 | 0.94 | 0.96 |
| P27 | P, R, F1 | 0.93 | 0.87 | 0.90 |
| P28 | P, R, F1 | 0.68 | 0.79 | 0.73 |
| P29 | P, R, F1 | 0.71 | 0.72 | 0.72 |
| P30 | F1 | / | / | 0.60 |
| P31 | P, R, F1 | 0.96 | 0.91 | 0.92 |
| P32 | P, R, F1 | 0.95 | 0.96 | 0.93 |
| P33 | Other | / | / | / |
| P34 | P, R, F1 | 0.87 | 0.86 | 0.86 |
| P35 | P, R, F1 | 0.87 | 0.99 | 0.82 |
| P36 | P, R, F1 | 0.67 | 0.70 | 0.68 |
| P37 | P, R, Other | 0.62 | 0.54 | / |
| P38 | F1 | / | / | 0.95 |
| P39 | P, R, F1, Other | 0.96 | 0.95 | 0.95 |

precision or recall, while one approach (P37) did not provide F1 but precision and recall. P33 is the only approach that provided neither precision, recall, nor F1.

The third, fourth, and fifth column of Table 4.11 shows the precision, recall and F1 values for each approach. These values are difficult to compare, however, due to several factors. Approaches have different goals and evaluation methodologies, and, most importantly, they report different granularities of performance metrics. Some approaches may report the precision, recall and F1 of individual classes, while others only report average values across all classes. They may also calculate the metrics using different averaging methods. This means that all comparisons between approaches are highly subjective. To provide some consistency in this mapping study, we report the highest values for each metric as reported by each approach. This, however, may make approaches which only report averages appear worse in comparison to approaches only reporting individual class performance. This also explains why the precision and recall values in Table 4.11 may not directly correlate to the F1 values.

Because of these problems, instead of comparing the individual performance metrics of each approach, we calculate the distributions across all approaches. Figure 4.3 shows the distributions of the metric values seen in Table 4.11. We see that the median precision is 0.81 across the approaches, with quartile ranges of 0.95 to 0.68. The lowest precision is 0.62, as reported by P37. The median recall is 0.83, ranging from a lower quartile of 0.65 to an upper quartile of 0.95. The

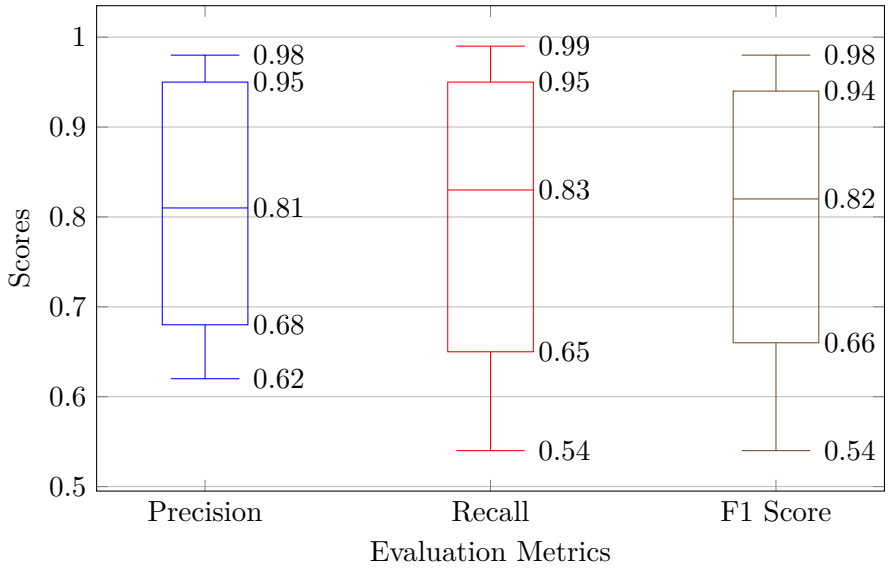



Figure 4.3.: Distribution of Precision, Recall, and F1 Scores Across Papers (SC10). Shown Values from Top to Bottom are: Maximum, Upper Quartile, Median, Lower Quartile, Minimum

lowest recall is lower than the lowest precision at 0.54, as also reported by P37. Median F1 is between median precision and recall at 0.82, with an upper quartile of 0.94 and a lower quartile of 0.66. The lowest F1 is reported by P25 at 0.54.

A complete synthesis matrix combining the synthesis criteria can be found in Appendix B.1.

## 4.3. Threats to Validity

This section discusses the threats to the validity of this mapping study according to (Ampatzoglou et al., 2020).

The first threat is the validity of the study selection. As explained in 4.1.2, we were not able to perform a keyword search due to the specificity of this mapping study's research questions. This creates the threat of missing relevant publications because we depend on the six SLRs from which we used snowballing to find relevant approaches. We tried to mitigate this threat as much as possible by investigating all 341 papers in the SLRs and performing another round of snowballing on the relevant papers, both forward and backward. However, even though the SLRs we used were broadly positioned in the field of feedback analysis, it is still possible that neither the SLRs' authors nor we captured some publications.

The second threat concerns the data validity. This threat stems from the fact that the results reported in this mapping study, especially those of the evaluation results, may be incorrect due to incorrect reporting in the primary papers. As mentioned before, evaluation methodology and calculation of metrics can differ from paper to paper, which makes comparing the reported metrics difficult. We decided to always report the best-reported values for each metric to reduce the threat. However, this could result in less rigorously evaluated approaches being perceived as superior because their reported values may be higher. The threat is further reduced by only including peer-reviewed publications in this mapping study.

The third threat concerns the research validity of the study. The results of this mapping study might not be perfectly repeatable because other researchers may choose other works as relevant, especially with regard to CoR5, the fine-grained classification criteria. We tried to provide an objective definition for *fine-grained* to reduce this threat, but it still remains somewhat subjective.

To facilitate reproducibility, we kept a log of the complete research procedure, including a complete list of the 341 studies and the reason for their exclusion and inclusion into this mapping study.

## 4.4. Conclusion

This study provided an overview of fine-grained feedback classification approaches in the literature. This section draws conclusions for treatment design and treatment validation of the usage information classification performed in this dissertation based on this mapping study's research questions.

**RQ1.1: What are the overall goals of these approaches?** We were not able to find other approaches that had the goal of classifying usage information in feedback. The overall goals can be sorted into four categories: Taxonomies, NFRs, Rational and Others. Usage information classification falls into the taxonomy group because it uses a taxonomy to identify the different types of usage information in the feedback.

The taxonomies we identified in this mapping study all focus on classifying characteristics of the software or the intent of the feedback itself. None incorporate classes focused on the users or their behaviour. In that, their purpose is very different from usage information classification. The rationale group is more user-focused by identifying the users' intent when giving feedback. However, this group does not focus on the behaviour of the software or the user when using the software. The NFR group does look at characteristics of the software, but by definition, these are only non-functional characteristics, so they do not capture the interactions of the users with the software, as usage information does.

We also could not identify any approaches that perform word-based classification (Section 2.1.2). Instead, all approaches in this mapping study either classify complete statements or individual sentences to achieve their goals. Thus, it is difficult to draw any conclusions from the overall goals of the approaches we found that would help with usage information classification. Our main takeaway is the necessity to evaluate the performance of usage information on different granularities, both word- and sentence-based.

**RQ1.2: Which predefined classes are used by these approaches to classify the feedback?** On average, the approaches we identified use approximately 11 classes for the fine-grained analysis. However, fine-grained analysis approaches can differ widely in the amount of classes they use. The approach we found with the lowest amount of classes had only four and the highest had 26 classes.

When using the adapted TORE framework, as introduced in Section 2.2.2, to classify usage information, nine classes are used, which is slightly below the average number of classes we found. However, as the approaches in the mapping study show, even approaches with a lower number of classes can perform fine-grained analysis.

**RQ1.3: Which automatic methods are used by these approaches?** No clear trend for specific machine learning models could be identified, as there seems to be no correlation between the number of classes an approach uses and the model they use to classify these classes. Some models are used more often than others (e.g. SVM, Naive Bayes, Logistic Regression). However, a total of 22 different models were employed.

What makes drawing conclusions from this even more difficult is that the evaluation metrics reported in Table 4.11 show that even when using the same model, approaches can have very different performances. The performance appears to be more dependent on the goal and classes used by the approaches than the machine learning models they employ. For example, approaches P31 and P32 report very good results when using BERT, while P24 and P25 show much lower results when using the same model.

There appears to be no one model, which can universally be applied to fine-grained usage information classification. Instead, experiments with different machine learning and deep learning models are needed to identify the one which performs the usage information classificaton best.

**RQ1.4: Which data sources and data sizes are used by these approaches in which steps?** App stores appear to be the primary source of feedback for most approaches. This is likely due to how easy it is to access feedback for a plethora of different applications with large feedback datasets for each. However, other sources are also used, such as review websites and social media (e.g., Twitter and Reddit). We draw from this the conclusion that the usage information classification should not be evaluated on a single data source. Instead, we gather our data from multiple different sources to best evaluate the generalizability of the approach across different feedback sources (Section 2.5).

Comparing dataset sizes across approaches is difficult. They differ widely between 500 reviews and close to 80.000. This means that there is not really a conclusion to draw for the dataset size we need in order to perform usage information classification, especially with regard to the fact that word-based or sentence-based annotation is considerably more effort than labelling entire feedback statements, as most approaches do.

**RQ1.5: How was the evaluation of these approaches conducted, and what are its results?** 12 of the 21 approaches perform all three of our evaluation methodology criteria: the creation of a manual ground truth, using interrater agreement to create the ground truth and performing k-fold cross validation. In order to perform the best validation possible and provide high-quality datasets for usage information classification, all three methodologies are also used for the datasets used in this dissertation.

The most common evaluation metrics are precision, recall and F1. While some approaches report only part of these metrics or include other metrics, we focus on these three metrics to evaluate the performance of our classifiers. To circumvent the problems encountered during this mapping study that approaches report these metrics very differently, either only as averages or as class-specific metrics, we report on both the overall metrics of the classifier as well as the individual class performances.

Evaluation results are difficult to compare across papers due to the differences in goals, classes and evaluation methodology. However, the distributions shown in Figure 4.3 can serve as a general guideline to compare the results of our evaluation to other fine-grained analysis approaches.

# Part III.

# Treatment Design

Chapter 5

# Feedback Requirements Relations

This chapter introduces the feedback requirements relation approach in Section 5.1. Afterwards, the experiments towards automation of the approach through machine learning classification are explained and discussed in Section 5.2.

The content of this chapter is partially based on a previous publication by the author of this dissertation (Anders and Paech, 2025). However, new experiments have been performed since the writing of this paper, mainly concerning experiments with different machine learning models, resulting in different values being reported compared to the publication.

## 5.1. FeReRe Approach

In Section 5.1.1, an overview is provided of the FeReRe approach towards feedback requirements relation. Section 5.1.2 then explains the intended use case of FeReRe.

### 5.1.1. Overview

The underlying idea of FeReRe is that When feedback is directly related to predefined requirements, developers can easily identify whether they are meeting expectations and understand what needs improvement. The relation helps to understand which functionalities are the primary focus of users' feedback and which themes appear related to existing functionalities.

One of the problems in the relation process is that developers and users have very different vocabularies when discussing software (Kuttal et al., 2020). This can make it difficult to determine whether a feedback statement and a requirement are related, therefore introducing the risk of missing important information. We performed early experiments in which we tried directly relating complete feedback statements to requirements by calculating the cosine similarity of the word embeddings of both. This was motivated by the work of Haering et al. (Haering and Nadi, 2021), who related bug reports to feedback. Our experiments with this approach had poor results. This can be attributed partially to the fact that a single feedback statement can discuss any number of software functionalities (Anders et al., 2023). As a result, feedback can often not only be related to a single requirement. Thus, feedback might be misclassified because parts of it might be related to a requirement while other parts are not. Also, as shown by Kuttal et al. (Kuttal et al., 2020), developers and users use very different vocabularies when discussing software. Therefore, the language used in feedback and requirements differs significantly.

We designed FeReRe to handle these two challenges: the fact that feedback can be related to any number of requirements and the different vocabularies found in feedback and requirements. As seen in Figure 5.1 the FeReRe approach consists of three steps to determine if a feedback and a requirement are related.

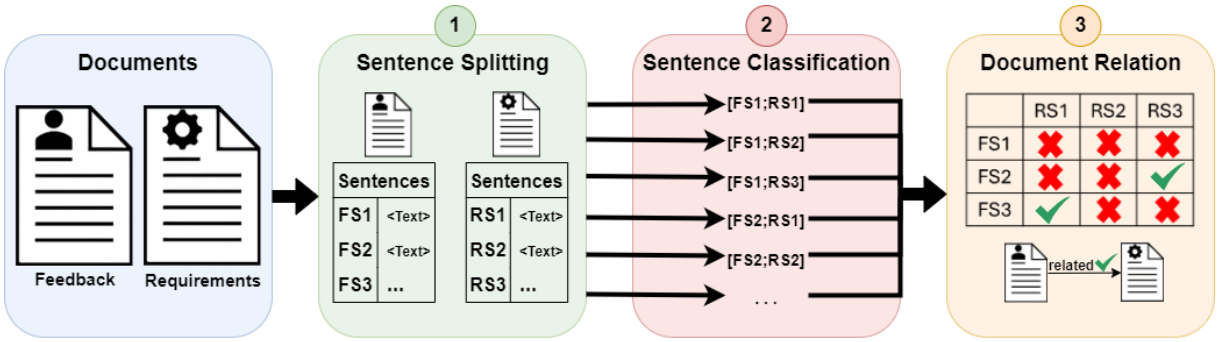



Figure 5.1.: FeReRe Approach

The first step splits the natural language text of both the requirement and the feedback into individual sentences (RS and FS). We use the *nltk* sentence splitter[1] to perform this task.

In the second step, every sentence from the feedback is paired with every sentence from the requirement. A classifier then performs a binary classification for every feedback-requirements-sentence pair by deciding whether the two sentences in the pair are related to each other. FeReRe does not require a specific classifier. The approach itself is applicable to any classifier capable of performing binary classification. Different classifiers are evaluated in Section 5.2. Our experiments identified BERT-Large (Devlin et al., 2018) as the best-performing classifier for FeReRe. To cope with the two previously mentioned challenges, we use fine-tuning by generating positive samples (sentence pairs that are related) and negative samples (sentence pairs that are not related). The positive and negative samples are combined, and their order is randomized and fed into the classifier for training. This fine-tuning reduces the risk that vocabulary differences affect the classification.

Finally, in the third step, it is determined whether a feedback and a requirement are related. We define the two as related if at least one sentence in the feedback is related to one sentence in the requirement. This mitigates the risk that complete feedback statements, which can be related to multiple requirements, affect the classification.

### 5.1.2. Use Cases

The key use case we see for FeReRe is requirements validation. Feedback that is directly linked to requirements helps determine whether a feature meets predefined specifications, reducing ambiguity and subjective opinions. In this regard, FeReRe serves as a grouping mechanism by grouping feedback related to the same functionalities and creating a connection between the feedback and the requirements which specify the functionalities of the software. This grouping is further expanded upon in Section 6.1.3.

Another possible use case is prioritizing issues and improvements. When feedback is mapped to requirements, it can help developers assess which parts of the software could be affected by feedback-driven changes by seeing the different requirements a feedback is related to. Changes to the software, motivated by feedback, rarely affect just a single functionality. Identifying which requirements are affected is simplified if developers gain access to the information to which requirements a feedback is related.

[1] https://www.nltk.org/api/nltk.tokenize.html

## 5.2. FeReRe Classifier

The FeReRe task involves a considerable amount of manual labour when performed without machine learning support. Consequently, this section describes the research towards the creation of FeReRe classifiers. Classifier selection and experiment configuration draw upon the conclusions taken from the mapping study conducted as part of the solution investigation (Section 3).

Section 5.2.1 describes the research questions guiding classifier experimentation. Section 5.2.2 describes how the experiments with the classifiers were configured and conducted. Section 5.2.3 discusses the creation of prompts for experiments with generative large language models. Section 5.2.4 presents the results of the conducted experiments and provides answers to the research questions. Section 5.2.5 discusses the findings before the threats to validity are discussed in Section 5.2.6. Lastly, Section 5.3 provides a final conclusion on the feedback requirements relation.

### 5.2.1. Research Questions

To analyze the performance of our classifier, we focus on the following research questions (RQ):

**RQ1:** *How well can FeReRe perform feedback requirements relation?*

**RQ2:** *Are classification results transferable across software domains?*

**RQ3:** *How does the FeReRe classifiers' performance compare to generative AI?*

**RQ4:** *Does the incorporation of already assigned feedback improve the results?*

RQ1 analyzes the overall performance of the FeReRe classifier. For this, we train and test the classifier on individual datasets as well as a combination of all datasets. We evaluate performance using precision, recall, and F2 measures (Section 2.3.8). First, we identify the best-performing model using a combination of all available data. Subsequent experiments are investigated using the best-performing classifier identified in these experiments. We compare the performance of the best-performing model on individual datasets to that of a combination of all data to see whether it is feasible to utilize a classifier trained on only one dataset and whether this classifier performs better than one trained on multiple datasets.

Classifier performance can be highly dependent on the software domain (Devine and al., 2023). The domain is the specific area of application or problem space that the software is designed to address or operate within. Even though all of our datasets concern apps, every app has a different application area (see Section 2.5). This means that a classifier trained on feedback and requirements from one software might not perform well on data from another software. With RQ2, we analyze the transferability of the classifier across software domains by testing it on datasets that are not used for training.

RQ3 analyzes the performance of our approach compared to the state-of-the-art generative LLM GPT4o. We performed our initial prompt engineering using ChatGPT (the dialogue-tuned web interface for GPT4o) and then evaluated the performance of the model with our prompts using the GPT4o API. These results are then compared to the FeReRe classifier.

In a practical usage scenario, relating feedback to requirements is an iterative process. As new feedback comes in, it must be related to requirements. RQ4 leverages the knowledge of previously assigned feedback when relating new feedback. The goal is to use the already assigned feedback as a classification feature. We investigate whether leveraging existing information improves results further for the classifier. Instead of just comparing sentences from requirements to sentences from new feedback (as described in Section 5.1), sentences from requirements are combined with sentences from already assigned feedback and compared to sentences from new feedback.

### 5.2.2. Experiment Configuration

Table 5.1 shows a list of all experiments conducted in this paper to evaluate the classifier for the FeReRe approach, along with the used model, datasets, and relevant RQs for each experiment.

Table 5.1.: FeReRe Experiments, Used Model and Dataset and Relevant RQ

| Experiment | Model | Datasets | Research Question |
|---|---|---|---|
| Model Comparison | BERT-Base | Komoot, SmartVernetzt, SmartFeedback, ReFeed | RQ1 |
| | BERT-Large | | |
| | DistillBERT-Base | | |
| | RoBERTa-Base | | |
| | Sentence-BERT | | |
| | Bi-LSTM | | |
| Individual Datasets | BERT-Large | Komoot, SmartVernetzt, SmartFeedback | RQ1 |
| Transferability | BERT-Large | Komoot, SmartVernetzt, SmartFeedback, ReFeed | RQ2 |
| GenAI Comparison | BERT-Large | Komoot | RQ3 |
| | GPT4o | | |
| Incorporate Feedback | BERT-Large | Komoot, SmartVernetzt, SmartFeedback, ReFeed | RQ4 |

For the *Model Comparison* experiments, four different BERT variants are trained and evaluated on a combination of all four datasets. Based on our data and preliminary experiments, we decided to use the large language model (LLM) BERT (Devlin et al., 2018) and its most commonly used derivatives as a classifier because it is already trained on extensive amounts of natural language data. We also experiment with another BERT derivative, namely Sentence-BERT (Reimers, 2019), a BERT variant specifically designed for the classification of sentence similarity. We also include an experiment with a non-BERT Bi-LSTM model (Hochreiter and Schmidhuber, 1997) for comparison.

To optimize the parameters for the classifiers, we first performed a grid search for each hyperparameter configuration with a data split of 80% training and 20% testing data across our datasets. This was done for every experiment individually. The classifiers were then evaluated using 5-fold cross-validation. This means that the combined datasets are split into 80% training and 20% testing data. This process is then repeated five times, ensuring that the testing data is different in every fold.

For the *Individual Datasets* experiment related to RQ1, the best-performing classifier from the model comparison, BERT-Large, is trained and tested on each of our datasets individually. For this, one dataset (e.g., Komoot) is split into training and testing data, again using 5-fold cross-validation. We excluded the ReFeed dataset from the individual experiments because our experiments showed that it was not large enough to train and test the model. Based on 60 feedback and 14 requirements, there were not enough positive samples for the model to learn adequately. However, in our opinion, a classifier would not be required for such a limited dataset anyway in real-world applications.

For RQ2, we investigate the transferability of the classifier to unseen software domains. We perform leave-one-out experiments by training on three of our datasets and testing on the fourth. For these experiments, 5-fold cross-validation does not apply because we split training and test datasets among all combinations of datasets. Since we test on entire datasets, no split is necessary.

Given the capabilities of generative large language models, we also use GPT4o, one of the most popular models, to perform the relation of feedback and requirements in RQ3. We develop multiple prompts to perform the task (see Section 5.2.3). GPT4o was chosen because it frequently outperforms open-source models (Zheng and al, 2024). Due to this difference in performance on common benchmarks and the resources needed to fine-tune open-source LLMs, we could not

perform any fine-tuning of generative Models; instead, we used a purely prompt-based approach. We only use the Komoot dataset to perform the GPT experiments. Due to privacy concerns related to the SmartVernetzt and SmartFeedback datasets, we cannot send these to the GPT servers abroad. However, the poor results of GPT for this task (see Section 5.2.4) lead us to believe that further experiments, for example, with the ReFeed dataset, would not be necessary.

### 5.2.3. Prompt Engineering

Working with generative LLMs like GPT requires the creation and refinement of prompts, which are used as input into the model. These prompts can take different forms and are suited for different tasks depending on their intended usage. Ronanki et al. (Ronanki et al., 2024) propose five distinct patterns of prompts, namely "Cognitive Verifier", "Context Manager", "Persona", "Question Refinement", and "Template". We experimented with these patterns as well as Zero-Shot, Few-Shot, and Chain-of-Thought prompt patterns (Rodriguez et al., 2023). Starting with an initial prompt, we refined the prompt further to achieve the output we desired from the model. Refinement of initial prompts is commonly used by other approaches like (Vogelsang and Fischbach, 2024) and (Rodriguez et al., 2023). A complete list of all prompts can be found in Table C.1 in Appendix C.1.

### 5.2.4. Results

In this section, we provide the results of our three research questions.

#### RQ1: How well can the FeReRe classifier perform feedback requirements relation?

Table 5.2.: RQ1: Best Performing Model for FeReRe approach

| Model | Precision | Recall | F2 |
|---|---|---|---|
| BERT-base | 0.82 | 0.94 | 0.91 |
| BERT-Large | **0.84** | **0.95** | **0.92** |
| DistillBERT | 0.72 | 0.89 | 0.85 |
| RoBERTa | 0.81 | 0.92 | 0.90 |
| SBERT | 0.68 | 0.73 | 0.72 |
| Bi-LSTM | 0.65 | 0.80 | 0.77 |

Table 5.2 lists the performance metrics for all six tested models when trained and tested on all available data using 5-fold cross-validation. BERT-Large achieves the highest F2 score of 0.92 as well as the highest precision and recall. This is followed closely by BERT-base with 0.91. RoBERTa reports the third-highest score of 0.90. DistillBERTs F2 drops to 0.85. The Bi-LSTM has the second lowest F2 at 0.77, and SBERT performs the worst with an F2 of 0.72. To maintain a manageable number of reportable experiments and to reduce unnecessary use of ressources, all subsequent experiments are performed with BERT-Large as the used model.

Table 5.3 shows the performance of the BERT-Large classifier on individual datasets, the average of these individual performances, and the classifier's performance when trained and tested on a combination of all four available datasets (as seen in Table 5.2). The individual performance is calculated by training and testing the classifier on only a single dataset. As discussed in Section 5.2.2, the ReFeed dataset can not be used for the individual experiments because, with 60 feedback messages and 14 requirements, it is simply too small to train a classifier on and still have enough data remaining to perform reliable testing.

Table 5.3.: RQ1: BERT-Large Classifier Trained and Tested on Individual Datasets

| Dataset | Precision | Recall | F2 |
|---|---|---|---|
| Komoot | 0.71 | 0.86 | 0.82 |
| SmartVernetzt | **0.88** | **0.98** | **0.96** |
| SmartFeedback | 0.85 | 0.95 | 0.93 |
| **Average** | 0.81 | 0.90 | 0.90 |
| AllCombined | 0.84 | 0.95 | 0.92 |

The average individual performance reaches an F2 score of 0.90. The highest F2 for individual performance is 0.96 for the SmartVernetzt dataset. The lowest F2 is achieved on the Komoot dataset, with 0.82. For the SmartFeedback and SmartVernetzt datasets, F2 is higher than the AllCombined classifier, which uses all data for training and testing. For the Komoot dataset, F2 is lower.

**Answering RQ1:** The FeReRe BERT-Large classifier can perform the feedback requirements relation with an F2 of 0.92 when trained and tested on a combination of all four datasets. Individual performance is higher for some datasets (up to 0.96 F2), while it is lower for others (0.82 F2).

#### RQ2: Are classification results transferable across software domains?

Table 5.4.: RQ2: Leave-one-out Experiments for Classifier Transferability

| Training | Testing | Precision | Recall | F2 |
|---|---|---|---|---|
| SV, SF, ReFeed | Komoot | 0.30 | **0.99** | **0.68** |
| Komoot, SV, ReFeed | SF | 0.71 | 0.58 | 0.60 |
| Komoot, SF, ReFeed | SV | **0.79** | 0.65 | **0.68** |
| Komoot, SF, SV | ReFeed | 0.64 | 0.56 | 0.57 |
| **Average** | | 0.61 | 0.76 | 0.63 |

Table 5.4 shows the results of the transferability experiments. In these experiments, the classifier was trained on three of our datasets and tested on the fourth. The highest F2 was achieved when testing on the Komoot and SmartVernetzt (SV) datasets, which resulted in a score of 0.68 for both. The Komoot experiment achieved the highest recall of 0.99 but a low precision of 0.30. The high recall is likely a consequence of the classifier simply related most feedback to most requirements, resulting in the low recall. The lowest F2 was achieved with the ReFeed dataset as a test set with a score of only 0.57. The highest precision was achieved by testing on the SmartVernetzt (SV) dataset with a precision of 0.79. The average F2 across all four leave-one-out experiments was 0.63, which is much lower than the results achieved in RQ1 when training and testing on all datasets.

**Answering RQ2:** Classification of unseen software domains results in a 0.29 lower F2 than a classifier trained on all datasets.

Table 5.5.: RQ3: Classifier Performance Compared To GPT4o

| Classifier | Prompt | Precision | Recall | F2 |
|---|---|---|---|---|
| GPT4o | Zero-Shot | 0.33 | 0.21 | 0.23 |
| | Few-Shot | 0.10 | 0.36 | 0.24 |
| | Chain-Of-Thought | 0.12 | 0.33 | 0.24 |
| | Predefined Structure | 0.17 | 0.34 | 0.28 |
| | Cognitive Verifier | 0.23 | 0.38 | 0.34 |
| | Context Manager | 0.18 | 0.33 | 0.28 |
| | Template | 0.20 | 0.33 | 0.29 |
| | Question Refinement | 0.16 | 0.35 | 0.28 |
| | Persona | 0.15 | 0.40 | 0.30 |
| **Average** | | 0.18 | 0.34 | 0.28 |
| BERT_Komoot | | **0.71** | 0.86 | **0.82** |
| BERT_LOO_Komoot | | 0.30 | **0.99** | 0.68 |

**RQ3: How does the FeReRe classifiers' performance compare to generative AI?**

Table 5.5 shows the best result achieved by the different GPT4o prompts for the feedback requirements relation as well as the average across all used prompts. The specific prompts can be found in Table C.1 in Appendix C.1. Because the classification was only performed on the Komoot dataset, the results are compared to the BERT classifier trained and tested exclusively on Komoot (BERT_Komoot) and to the leave-one-out (LOO) classifier from RQ2, which is not trained on Komoot (BERT_LOO_Komoot). We include the latter to compare the performance of GPT to a classifier that is also not trained on Komoot data.

On average, GPT4o achieves a low F2 of 0.28. The prompt with the highest F2 is the *Cognitive Verifier* with 0.34. Compared to the classifier trained and tested exclusively on Komoot data (BERT_Komoot), the *Cognitive Verifer* achieved a 0.48 lower F2. The classifier not trained on any Komoot data (BERT_LOO_Komoot) outperforms the prompt by an F2 of 0.34. The highest recall prompt is the *Persona* prompt with 0.40, which is still lower than either BERT_Komoot or BERT_LOO_Komoot.

**Answering RQ3:** GPT4o performs the classification worse than any of our classifier experiments. A BERT classifier with no knowledge of the Komoot software domain still outperforms GPT4o.

**RQ4: Does the incorporation of already assigned feedback improve the results?**

Table 5.6.: RQ3: Classifier Performance Compared To GPT4o

| Experiment | Precision | Recall | F2 |
|---|---|---|---|
| BERT_Large | 0.84 | 0.95 | 0.92 |
| BERT_Large_FeedbackIncorporated | 0.80 | 0.95 | 0.92 |

Table 5.6 shows the results of the incorporation experiments. For each requirement, a random feedback statement from the ground truth is selected as *"already assigned"*. This means that the feedback is treated as if it had already been related to the requirement. The remaining feedback is then related to the classifier by comparing the feedback to the requirement and the *"already*

*assigned* feedback. Due to a lack of data it was not possible to perform successive experiments where more than one feedback per requirement was treated as *"already assigned"*. The table shows that the incorporation of already assigned feedback did not affect the F2 score, which remains unchanged at 0.92. Recall was also not affected. The only difference is a slight drop in precision to 0.80 from 0.84.

**Answering RQ4:** Incorporation of already assigned feedback does not improve the classification results. F2 and recall remain unchanged, while precision drops by 0.04.

### 5.2.5. Discussion

#### Non-Generative LLM Performance:

The results of our experiments demonstrate that the FeReRe BERT-Large classifier performs well, particularly in terms of recall, which is the most critical metric for the task. Achieving an F2 of 0.92 when trained and tested on a combination of all datasets, the classifier reliably identifies relevant feedback requirements relations. All BERT derivatives except for SBERT perform fairly well. SBERT, despite being a model trained specifically for sentence relation tasks, is not able to relate feedback and requirements as well. BI-LSTM has a similarly worse performance.

The classifier's performance on individual datasets differs by an F2 of 0.14, ranging from 0.82 on the Komoot dataset to 0.96 on the SmartFeedback dataset. These differences are likely to be due to the differences in feedback across the datasets. The classifier performs similarly well on SmartVernetzt and SmartFeedback. Both datasets contain feedback gathered through a feedback app, which asks the users specific questions about the apps. As such, the feedback is more targeted to individual functionalities of the software and likely to be related to fewer different requirements. The Komoot dataset, on the other hand, contains feedback crawled from an app store. As Table 2.4 shows, this feedback is much longer than the feedback for SmartVernetzt and SmartFeedback and, thus, more likely to be related to multiple different requirements. This explains why the classifier struggles more with this type of feedback, as the chances for misclassification are higher.

#### Transferability:

The classifier's transferability across different software domains is limited. Performance drops sharply when classifying feedback from unseen domains. This indicates that domain-specific training is necessary to maintain high classification performance, as the classifier's knowledge does not transfer well across domains. These findings are in line with other research (Devine and al., 2023). The poor transferability poses a challenge for developers working in multiple software domains, as continuous retraining or adaptation of the classifier would be required to ensure accurate relations. We also notice that in the case of the Komoot testing experiment, the classifier achieves exceptionally high recall at 0.99 at the cost of a low precision of 0.30. This indicates that for the data, the classifier defaults to relating most feedback to most requirements, which produces a high recall but also a high number of false positive classifications. In general, the findings indicate that a classifier, specifically trained on the software domain it is classifying, is required for practical usage.

#### Feedback Incorporation:

Our experiments with using already assigned feedback as a classification feature (RQ4) to improve the relation were based on the observation that two feedback texts were more likely to be semantically similar than feedback and requirements text. This is because users tend to write very

differently in their feedback than developers do when specifying requirements. They use different words and may pay less attention to spelling and grammar. The idea was to improve the relation by combining feedback and requirements text when performing the relation. Our experiment, as shown in Table 5.6, shows that this improvement did not occur. Incorporating the feedback did not create different results. A qualitative analysis of 200 randomly chosen sentence-pair classifications of the classifier showed that in almost all cases in which the classifier decided that a feedback sentence was related to an incorporated feedback sentence, the classifier also decided that the feedback sentence was related to a requirement sentence as well. The analysis also showed that in some cases, the classifier related a feedback statement to a requirement solely because of the incorporated feedback, even though that classification was incorrect. This could explain the slight drop in precision. Consequently, the qualitative analysis likely indicates that the classifier is already performing the relation as well as possible and that already assigned feedback may even introduce noise into the classification, which could also negatively affect the results.

#### Generative LLM Performance:

Our experiments show that GPT4o performs worse than the FeReRe BERT classifier. The results suggest that GPT-based models may currently not be suitable for the relation task. This is particularly notable as the BERT-Large classifier, even when not trained on the specific software domain, still provides more accurate results than GPT4o. It is difficult to say why GPT performs so poorly. Our chain-of-thought (Rodriguez et al., 2023) experiments and asking the model for explanations on its classification did not provide any meaningful insights. The most likely explanation seems to be that generative LLMs do not perform such classification tasks systematically as they merely predict the most likely tokens to respond with, given an input. A non-generative LLM like BERT can be fed data systematically, and it can be ensured through code that the model behaves in the intended way and performs the classification.

### 5.2.6. Threats to Validity

#### Reliability

The reliability of the results is threatened because of the marginal differences observed between the BERT-Large, BERT-base and RoBERTa classifiers across the initial experiments in RQ1 (F2 +-0.02). Such small variations make it challenging to draw definitive conclusions about which model performs best under specific conditions. This also affects successive experiments where the best-performing classifier (BERT-Large) was used instead of alternative classifiers. To mitigate the risk of our results being influenced by chance, we employed 5-fold cross-validation whenever feasible. The absence of statistical testing, however, introduces a limitation. Without formal statistical analysis, we cannot confirm whether the observed differences in performance are genuinely significant or merely the result of random variation.

#### Construct Validity

A threat to the construct validity is the evaluation of the classifier's performance based on the presented metrics precision, recall, and F2. In Section 2.3.8, we explained our reasoning behind these metrics. Different conclusions might be drawn when different evaluation metrics are used. However, as precision, recall and F-Meassure are very commonly used for machine learning evaluation, we feel that this risk is minimal. Another threat to the construct validity exists because the SmartFeedback and SmartVernetzt apps are both highly related. They were designed as part of the same study and feedback for both apps is given in SmartFeedback. SmartFeedback

also is a dedicated feedback app which may not be representative of other apps. This may pose a threat to the conclusions drawn for the transferability experiments.

#### Internal Validity

One threat to internal validity is the implementation of our approach. We have used widely used libraries for sentence splitting and models to alleviate this. We also performed our testing with an established machine learning experiment management platform in MLFlow. Additionally, our code is publicly available to allow replication. We alleviated the internal threat of manual coding by employing multiple coders for every dataset and ensuring that two people independently performed the relation, after which an interrater agreement was established (see Section 2.5). Another threat to the internal validity is due to the prompt design. Given the non-deterministic nature of generative LLMs, different prompts often lead to different results. We tried to minimize this threat by carefully revising our prompts using existing research towards prompt engineering and evaluating many different prompt patterns.

#### External Validity

Threats to the external validity are mainly due to our datasets. For the Komoot dataset, we had to recreate requirements ourselves as the original requirements for the software were not publicly available. This means that the software wasn't designed with the specific requirements we present in our dataset. However, we tried to reduce the threat by systematically recreating all requirements using the requirements framework TORE. We iteratively refined the requirements to reproduce requirements that match the software as closely as possible. The study context in which the SmartAge dataset is gathered also presents a threat. The feedback is gathered from study participants, who were selected based on their age and the location in which they lived. While these participants were not paid, they did receive tablets on which they used the apps. This could influence the feedback they give. We made our datasets publicly available to allow independent investigation and replication. An external threat to all our datasets is the fact that all feedback statements are related to at least one requirement. This might not be representative of other feedback datasets where certain statements may not be relatable to any feedback. Consequently, it is unclear how well the classifier would perform with feedback that can not be related to any requirement at all, as this did not occur in our data.

## 5.3. Conclusion

BERT-base achieved an F2 that is only 0.01 lower than that of BERT-base. Given the very similar performance of BERT-Large and BERT-base and the significantly higher computation cost of BERT-large (almost double the amount for training and classification time for our datasets), BERT-base can be recommended as the preferred classification model, especially for larger datasets. We performed our experiments with BERT-large nonetheless, as computational power was not a restriction for the data available to us. However, this recommendation may change depending on the available datasets, as other data might result in different model performances where BERT-large could outperform BERT-base.

In summary, the FeReRe approach, using a BERT classifier, is able to perform the classification accurately. It also fulfils its designed purpose of achieving high recall, promoting its use as a semi-automatic approach. The semi-automatic approach is further supported by the functionalities provided by the Feed.UVL tool, which is explained in Chapter 7. The effectiveness of the FeReRE approach is evaluated in the treatment validation in Chapter 8.

Chapter 6

# Usage Information Classification

This chapter introduces the usage information classification approach in Section 6.1 and identifies the best-performing classifier towards usage information classification automation in Section 6.2.

The contents of this chapter are largely based on a journal paper that has, at the time of writing, not been published by the author of this dissertation.

## 6.1. UIC Approach

This section first provides an overview of the approach towards UIC in Section 6.1.1. Section 6.1.2 then discusses the different granularities of usage information in more detail. Section 6.1.3 introduces the two use cases for the application of UIC. Section 6.1.4 then provides concrete examples for these use cases.

### 6.1.1. Overview

As an interview study with software practitioners by Li et al. (Z. S. Li et al., 2024) shows, user feedback in industry is often manually grouped by themes. These themes represent common complaints or often desired features. These themes appear when a growing amount of feedback discusses the same specific issues. This allows developers to identify common issues or concerns users express. FeReRe is a first step towards grouping feedback through the functionalities it discusses. Further refining these groups through themes allows for more fine-grained identification of issues, as developers can clearly see which issues users commonly express for which functionality. The feedback discussing these themes can be grouped by the various types of different usage information we analyze. Usage information can also help developers look for more feedback on a specific theme via the type of usage information it is related to. A developer can, for example, sort feedback into a user interface (UI) theme or a specific system function theme by looking at whether that type of usage information is present in the feedback. In this context, FeReRe serves as the first grouping mechanism by sorting the feedback into related requirements, and UIC serves as the more fine-grained grouping mechanism by then sorting the feedback of each requirement into usage information themes.

We investigate increasingly fine-grained granularities of usage information classification. We can analyze the feedback by classifying which type of usage information is primarily discussed in whole sentences, or alternatively, we can classify individual words in the feedback by the type of usage information. We can also analyze coarse levels of usage information and finer categories.

To define the levels and categories of usage information, we use the TORE framework (see Section 2.2). It encapsulates usage information on three coarse levels. The first level captures the domain in which the software is used. It encapsulates the different user roles that use

the software as well as the actions users perform in real life for which they require software support and related data. The second level encapsulates users' direct interactions with software, including the different views they might mention and the functions of the software. The third level encapsulates individual software components and the architecture of the software, such as discussions of different servers a software might have. The TORE framework also allows us to split these levels into more fine-grained categories. These include categories that capture aspects of the UI, the users' descriptions of interactions performed in the software or the data exchanged between the user and the software.

The combination of sentence and word analysis, as well as level and category classification, yields three different types of granularity:

- **Sentence-based TORE Level classification**, where entire sentences are analyzed to determine the primary TORE Level they belong to.
- **Word-based TORE Level classification**, where individual words are labelled with TORE Levels.
- **Word-based TORE Category classification**, which further refines the classification by assigning words to specific TORE Categories.

Classifying the fine-grained TORE categories is impossible on a sentence-level, because it is impossible to assign such fine-grained categories to a whole sentence.

### 6.1.2. Granularities

**Sentence-Based TORE Level Classification:** Sentence-based classification assigns an entire sentence to one of the three TORE Levels: Domain Level, Interaction Level, or System Level. This approach captures the overall context of user feedback without focusing on individual words, making it useful for understanding broad themes within feedback. One of its key advantages is that it simplifies classification, reducing the complexity compared to word-level approaches, as a single label is assigned per sentence. However, it lacks precision since a sentence can contain multiple types of usage information, making it difficult to capture finer distinctions. This approach is particularly beneficial when developers need a high-level overview of how users discuss different usages of the software.

**Word-Based TORE Level Classification:** In word-based TORE Level classification, each individual word in the feedback is analyzed and assigned a corresponding TORE Level. This approach provides a more fine-grained understanding of how users express their interactions with the software by distinguishing between different parts of the same sentence. A major advantage is its ability to separate multiple types of usage information within a single piece of feedback, allowing developers to identify specific mentions of user tasks, interactions, or system-related aspects. Word-based classification is a more complicated task for both humans and machine learning classifiers than sentence-based classification. It is, however, particularly useful when developers want to extract detailed insights without being restricted to the overall sentence context.

**Word-Based TORE Category Classification:** The most detailed approach, word-based TORE Category classification, goes beyond TORE Levels and assigns each word to a specific category within its respective level. This means that words are labelled as Stakeholder, Task, Activity, Domain Data, Interaction, System Function, Interaction Data, Workspace or System (Table 2.3). This level of granularity allows for an in-depth analysis of how users discuss particular features and functionalities, which is useful for identifying trends, recurring issues, or design inconsistencies. However, this level of classification is also the most challenging, as words often carry ambiguous meanings, and the extended amount of classes compared to TORE Levels

complicates class assignments. The primary advantage of this approach is that it provides the most detailed information, helping developers understand exactly how users are engaging with the software or how they are struggling to use it.

### 6.1.3. Use Cases

We see two potential use cases for usage information classification. These go hand-in-hand with the use cases identified for FeReRe in Section 5.1.2. The first use case is the above-mentioned grouping of feedback by common usage information themes. In this case, after relating the feedback to requirements (Chapter 5), the feedback for each requirement is further grouped by the usage information it contains. Feedback discussing, for example, certain workspaces, tasks or system components can be grouped together to establish common themes.

By systematically coding user statements using the TORE framework, developers can extract structured insights from user feedback. This analysis allows for the identification of frequent user concerns, patterns of software interaction, and potential usability issues by filtering the feedback via common usage information themes. Once a developer has identified a problematic software behaviour, they can easily search for other feedback statements raising the same issues by searching for the same type of usage information. For this use case, TORE categories are the targeted granularity as they provide the detailed insights needed to group the feedback.

It must be noted that usage information does not automate the process of identifying issues or finding solutions for them. Usage information helps in identifying important terms or sentences that users use to express their usage of the software. It serves as a support mechanism for developers when finding common issues.

The second potential use case for UIC application is towards requirements validation. Usage Information can help identify whether the documented requirements cover the users' actual needs or whether certain functionalities are missing. For this use case, TORE Levels are mostly sufficient. Again, FeReRe provides the first step by relating the feedback to the relevant requirements. As shown in the example in the following Section, in order to identify whether a user task covers the user's needs, feedback containing the domain level needs to be analyzed. For cases where developers want to validate that their system functions are designed the way users are actually using them, the Interaction Level needs to be analyzed. To validate the requirement, the developer can analyse whether the highlighted usage information is present in the requirements documentation as well, i.e. the requirement covers the domain or interaction the user intends to use it for.

### 6.1.4. Approach Application

This section provides concrete examples for the application of the UIC use cases

#### 1st Use Case: Feedback Grouping

To illustrate possible use cases for the UIC approach, the following example of coding a user feedback statement is presented. The coding process involves segmenting the text into meaningful linguistic units, assigning appropriate labels, and categorizing them within the TORE framework. The example use cases provide a simplified version of the process. In actuality, while performing UIC, annotators mentally perform steps 1 and 2 while reading the statements rather than physically.

Example statement:

*"I wish one could turn off automatic replanning. When the GPS signal is lost in the navigation view, it starts to automatically replan, thereby replacing my originally planned tour."*

**Step 1:** Segmentation of the Statment into Text Units

Depending on the granularity. The statement is divided into distinct parts to isolate different aspects of usage information. Each segment represents either a sentence or a term. As the following example shows, terms can either be composed of multiple or individual words. They represent semantically distinct expressions such *"turn off"* or *automatic replanning.*

*I* / wish / one / could / turn off / automatic replanning. / When / the / GPS signal / is / lost / in / the / navigation view, / it / starts / to / automatically replan, / thereby / replacing / my / originally planned tour.

**Step 2:** Identification of Relevant Segments

In the second step, segments relevant to usage information are identified. This involves identifying verb and noun terms and deciding whether they contain relevant usage information.

*I* / wish / one / could / turn off / automatic replanning. / When / the / GPS signal / is / lost / in / the / navigation view, / it / starts / to / automatically replan, / thereby / replacing / my / originally planned tour.

**Step 3:** Assignment of TORE Codes

Each of the relevant segments is mapped to a corresponding TORE Level based on the nature of the information it conveys. When assigning the codes, the context of each segment is taken into account. An example of this is *originally planned tour* below, which in a different context might be considered as part of the *Domain Data* category. However, as it is put into the context of the software's interaction of *replacing* a previous tour, it becomes part of the *Interaction Data* category. The same is true for the *GPS signal* in the example. Because the user puts it into context with the *navigation view losing* the signal, it is part of the *Interaction Data category.*

*I* / wish / one / could / turn off {Interaction} / automatic replanning. {System Function} / When / the / GPS signal {Interaction Data} / is / lost{Interaction} / in / the / navigation view, {Workspace} / it / starts / to / automatically replan, {Interaction} / thereby / replacing {Interaction} / my / originally planned tour. {Interaction Data}

**Step 4:** Implications and Use of Coded Data

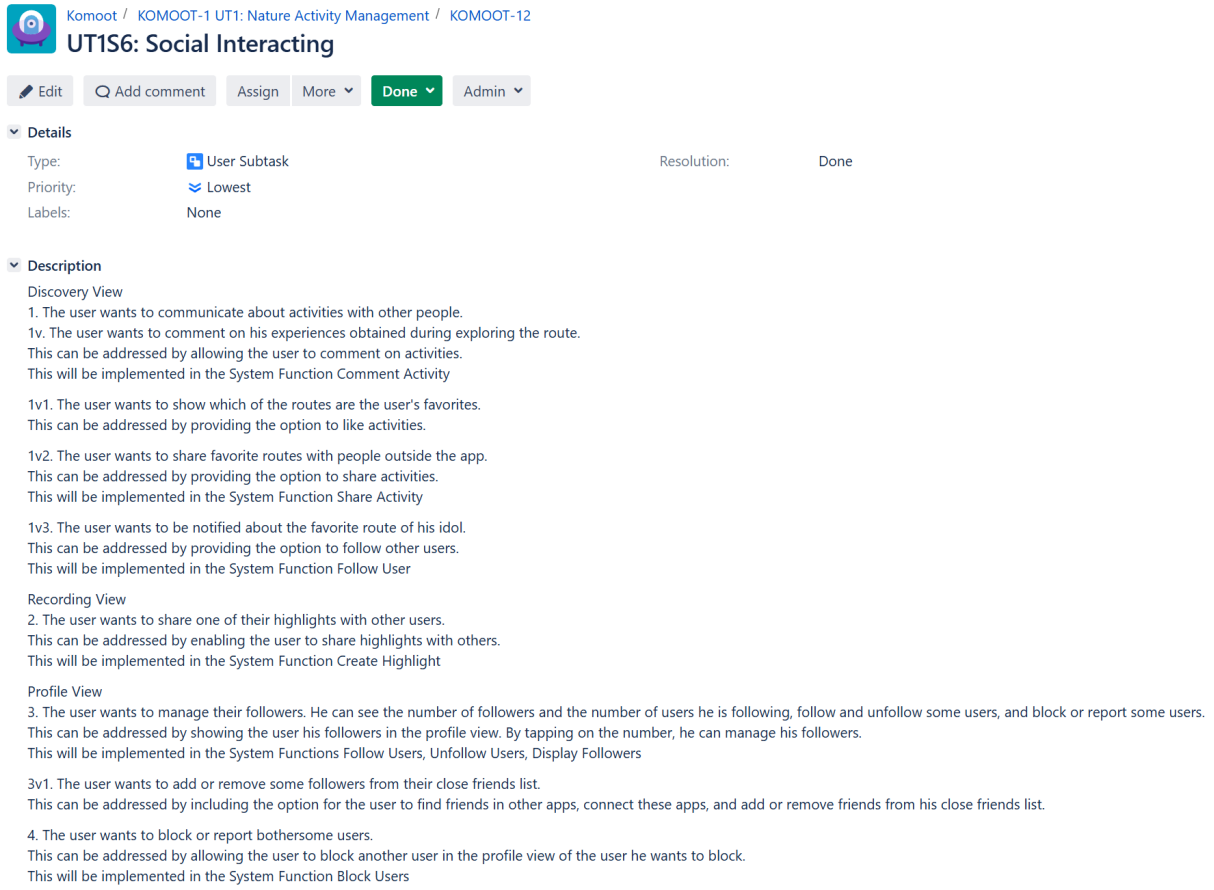
Looking at the above example, a software behaviour is described by the user, which they find undesirable. It does not constitute a fault in the program as such, as the software behaves as it is designed. The user, however, desires a different behaviour. Using the identified usage information terms, like *automatic replanning*, *turn off*, and *navigation view*, a developer is now able to quickly filter the other feedback statements they have collected to see if other users are also complaining about this part of the software's behaviour.

### 2nd Use Case: Requirements Validation

Figure 6.1 shows one of the user sub-tasks of the Komoot hiking app. In it, the user's social interaction activities are described, for which the software should offer support.

Applying the previously introduced steps, this time using TORE Levels instead of categories, on another example feedback statement results in the following coding.

"I'm a tour guide {Domain Level} so I want to use this app to inform {Domain Level} my customers {Domain Level} on which routes {Domain Level} we can go."


Komoot / KOMOOT-1 UT1: Nature Activity Management / KOMOOT-12

UT1S6: Social Interacting

Edit | Add comment | Assign | More | Done | Admin

Details

Type: User Subtask
Resolution: Done
Priority: Lowest
Labels: None

Description

Discovery View
1. The user wants to communicate about activities with other people.
1v. The user wants to comment on his experiences obtained during exploring the route.
This can be addressed by allowing the user to comment on activities.
This will be implemented in the System Function Comment Activity

1v1. The user wants to show which of the routes are the user's favorites.
This can be addressed by providing the option to like activities.

1v2. The user wants to share favorite routes with people outside the app.
This can be addressed by providing the option to share activities.
This will be implemented in the System Function Share Activity

1v3. The user wants to be notified about the favorite route of his idol.
This can be addressed by providing the option to follow other users.
This will be implemented in the System Function Follow User

Recording View
2. The user wants to share one of their highlights with other users.
This can be addressed by enabling the user to share highlights with others.
This will be implemented in the System Function Create Highlight

Profile View
3. The user wants to manage their followers. He can see the number of followers and the number of users he is following, follow and unfollow some users, and block or report some users.
This can be addressed by showing the user his followers in the profile view. By tapping on the number, he can manage his followers.
This will be implemented in the System Functions Follow Users, Unfollow Users, Display Followers

3v1. The user wants to add or remove some followers from their close friends list.
This can be addressed by including the option for the user to find friends in other apps, connect these apps, and add or remove friends from his close friends list.

4. The user wants to block or report bothersome users.
This can be addressed by allowing the user to block another user in the profile view of the user he wants to block.
This will be implemented in the System Function Block Users

Figure 6.1.: Example User Sub-Task for Komoot Hiking App

In order to validate that the software provides the necessary functionalities for the user, as stated in their feedback, a developer can look at the highlighted section containing usage information and cross-reference them with the documented requirement. The relation to the requirements can be automated using FeReRe (Chapter 5). The usage information shows that the user is a *tour guide* and wants to *inform customers*. Looking at the requirement, the developer can see that this use case is already captured in *1v2: The user wants to share favourite routes with people outside of the app*. Thus, this need, as stated by the user, is covered by the software. A similar process is also possible with other types of requirements, such as User Stories, which directly express the requirements as users' desires and needs. By checking the overlap of requirements and expressed usage information, gaps in the documented requirements can be identified.

The examples shown in this section highlight usage information in the form of TORE Levels. However, if it is the desire of the developer to gain further insights, such as whether, for example, different types of *Stakeholders* are mentioned, the use of TORE Categories could provide more information.

## 6.2. UIC Classifier

The usage information classification task described in the previous section involves a large amount of manual labour in order to code the feedback. This is especially so for the finer word-based granularities. In order to reduce the manual effort required, design goal 2 is supported

by knowledge goal 4 *"Find the best-performing classifier for usage information classification"* (Section 1.3). Consequently, this section describes the research towards the creation of automatic usage information classification classifiers. Classifier selection and experiment configuration draw upon the conclusions taken from the mapping study conducted as part of the solution investigation (Section 4).

Section 6.2.1 describes the research questions for the creation of automatic classifiers. Section 6.2.2 describes how the experiments with the machine learning classifiers were configured and conducted. Section 6.2.3 discusses the creation of prompts for experiments with generative large language models. Section 6.2.4 presents the results of the conducted experiments and provides answers to the research questions. Section 6.2.5 discusses the findings before the threats to validity are discussed in Section 6.2.6. Lastly, Section 6.3 provides a final conclusion on the usage information classification.

### 6.2.1. Research Questions

As described in Section 6.1.2, different granularities of usage information can be identified in feedback. To explore the automatic classification of these different granularities of usage information, the following research questions are defined. All research questions except those pertaining to binary classification and preprocessing were formulated before analyzing any data and are therefore independent of any assumptions derived from the dataset. This approach ensures that the data itself does not influence the core inquiries guiding the research. Those questions pertaining to binary classification (RQ1.2) and the use of preprocessing techniques (RQ1.3, RQ2.2, RQ3.3) were introduced later as a result of specific observations made during data collection.

**RQ1:** How well can automatic sentence-based TORE Level classification be performed?

- **RQ1.1:** Which classifier performs best?
- **RQ1.2:** Does binary classification improve classification results?
- **RQ1.3:** Does preprocessing of the data improve classification results?

**RQ2:** How well can automatic word-based TORE Level classification be performed?

- **RQ2.1:** Which classifier performs best?
- **RQ2.2:** Does preprocessing of the data improve classification results?
- **RQ2.3:** Are classification results transferable across feedback sources?

**RQ3:** How well can automatic word-based TORE Category classification be performed?

- **RQ3.1:** Which classifier performs best?
- **RQ3.2:** Does multi-stage classification improve results compared to single-stage classification?
- **RQ3.3:** Does preprocessing of the data improve classification results?
- **RQ3.4:** Are classification results transferable across feedback sources?
- **RQ3.5:** Does the specificity of the TORE Categories influence classification results?

All three research questions share some common sub-questions. First, we investigate which classifier performs the given task best (RQ1.1, RQ2.1 and RQ3.1). Second, because our datasets are highly imbalanced (see Section 2.5) and online feedback presents some common problems, like incorrect spelling or irrelevant statements (Section 2.1.3), we investigate whether preprocessing of the feedback data increases performance (RQ1.3, RQ2.2 and RQ3.3). The imbalance in the data

is inherent to usage information in feedback because users do not discuss every type of usage information equally. To combat the incorrect spelling, we apply spellchecking algorithms to the feedback before training. To reduce the amount of irrelevant statements, we perform relevance classification on the feedback, excluding sentences which do not contain any information related to the software.

RQs 2.3 & 3.4 investigate whether the classifier transfers between the different feedback sources. Other studies have also shown that classifier performance can be highly dependent on the feedback source (Novielli et al., 2020) (Devine and al., 2023). Thus, we train on one or more of the sources and test on another source. Note that we do not investigate the generalizability across different software products; rather, we investigate how the platform through which the feedback is given affects classifier performance. Due to a lack of data, this is unfortunately impossible for the sentence-based TORE Level classification in this dissertation.

As described in Section 2.2, TORE is composed of two stages, the levels and categories. In RQ3.2, we want to investigate the possibility of performing multi-stage classification on the word level to improve classification results. During multi-stage classification, TORE Levels (first stage) are classified by one classifier specifically trained for Level classification. This result is then used as a classification feature for the TORE Category (second stage) classification by another classifier.

For example, a word identified as being on the *Domain Level* in the first stage must, in the second stage, then be either a *Stakeholder, Task, Activity* or *Domain Data*. Thus, using multi-stage classification could improve results by limiting the number of possible classes in the second stage. However, it also introduces a risk of misclassification in the first stage, leading to incorrect classification in the second stage. Multi-stage classification is only feasible for TORE Category classification, as it requires a first stage (TORE Levels) and second stage (TORE Categories), so it only appears as a sub-question to RQ3.

Some sub-questions differ between the three main research questions. RQ1.2 analyzes whether training one classifier per sentence-based TORE Level that decides whether a sentence is on the respective level or not improves classification results. This question does not apply to RQ2 and RQ3, as some of the TORE categories simply did not occur frequently enough to train a binary classifier on them. Classifiers trained for classes like *Task* with only 319 words out of around 55.000 would mostly not label any words, making binary classification useless for a lot of the classes.

In order to understand the limits of the specificity of TORE Categories, which our classifiers are able to tell apart, we investigate in RQ3.5 the effects of combining classes which the classifier struggles to discern from one another. This form of reducing the specificity only makes sense for the TORE Categories, as the TORE Levels are already of lower specificity and significantly more dissimilar from one another than the individual categories.

We do not perform sentence-based TORE Category classification, as it is impossible to assign individual TORE categories to entire sentences. Sentences almost never mention only one TORE category. Instead, this would require performing multi-label classification, where one sentence is assigned multiple TORE categories. This would require the creation of new dataset codings which are not currently available.

### 6.2.2. Experiment Configuration

Table 6.1 shows a list of all experiments conducted in this paper, along with the models, datasets and relevant RQs for each experiment. "Basic Classification" refers to experiments where no additional methods were used to try and improve the results of the classifier. After experimenting with multiple classifiers in the basic classification experiments, we perform subsequent experiments with the best-performing model of the basic classification. In our case, this was BERT-Large

Table 6.1.: Experiments, Used Model and dataset and relevant RQ for each Task

| Classification Task | Experiment | Model | Datasets | RQ |
|---|---|---|---|---|
| Sentence-Based TORE Level | Basic Classification | BERT-Large | SmartAge | RQ1.1 |
| | | RoBERTa-Large | | |
| | | Llama3-70B | | |
| | Include Questions as Feature | BERT-Large | | |
| | Binary Classification | | | RQ1.2 |
| | Synthetic Oversampling | | | RQ1.3 |
| | Spellchecking | | | |
| | Relevance Classification | | | |
| Word-Based TORE Level | Basic Classification | BERT-base | Prolific, Forum, App Review | RQ2.1 |
| | | BERT-Large | | |
| | | Bi-LSTM | | |
| | | RoBERTa-Large | | |
| | | SNER | | |
| | | Llama3-70B | App Review | |
| | Synthetic Oversampling | BERT-Large | Prolific, Forum, App Review | RQ2.2 |
| | Spellchecking | | | |
| | Relevance Classification | | | |
| | Transferability | | | RQ2.3 |
| Word-Based TORE Category | Basic Classification | BERT-base | | RQ3.1 |
| | | BERT-Large | | |
| | | Bi-LSTM | | |
| | | RoBERTa-Large | | |
| | | SNER | | |
| | | Llama3-70B | App Review | |
| | Multi-Stage Classification | BERT-BERT-Large | Prolific, Forum, App Review | RQ3.2 |
| | | BiLSTM-BERT-Large | | |
| | | SNER-BERT-Large | | |
| | Synthetic Oversampling | BERT-Large | | RQ3.3 |
| | Spellchecking | | | |
| | Relevance Classification | | | |
| | Transferability | | | RQ3.4 |
| | Specificity | | | RQ3.5 |

for all classification tasks. We decided not to perform every experiment with every model for multiple reasons. As Table 6.1 shows, we performed 32 experiment-model combinations. The inclusion of every model with every experiment would have resulted in a much larger amount of combinations. In our view, this amount is too high to properly report and would not have been feasible in terms of time and resource consumption. For sustainability reasons, we decided not to spend these resources.

### Classifier Selection

In total, we experimented with six different classifiers. The BERT-Base classifier (Devlin et al., 2018), the Stanford Named-Entity Recognition classifier (SNER) (Finkel et al., 2005), and a Bi-LSTM based classifier (N. Li et al., 2019) were selected because all three had been used in the past to perform word-based classifications such as ours. They also cover three overlapping areas, namely pre-trained large language models (LLMs) with BERT, neural network deep learning (DL) with Bi-LSTM and more traditional machine learning (ML) with SNER. We decided to use BERT-Base as it is widely used in the related literature (Section 4.2.2). Because

BERT-Base performed better than SNER and Bi-LSTM in our research, we expanded upon that selection by also experimenting with BERT-Large (Devlin et al., 2018) and the BERT derivative RoBERTa-Large (Y. Liu et al., 2019).

We also investigated the possibility of using a BERT model specifically trained for Named-Entity-Recognition tasks such as this one provided by the hugging face library [1]. However, we found that these models were trained to recognize entities such as locations and organizations. This does not fit our use case, especially for TORE Categories such as *Interaction* and *Activity*, which describe actions and are thus mapped to verbs. Using such a NER model on our data effectively removes the fine-tuning for entity classes, turning it into a standard BERT model.

#### Generative LLM Experiments

Given the capabilities of generative large language models, we also employed the Llama3-70B model (AI@Meta, 2024) to perform all three classification tasks. We decided to use Llama as opposed to the more widely used GPT (OpenAI, 2024), even though the newest version, at the time of writing, *"GPT4o"* uses a significantly larger amount of parameters indicating better capabilities. This decision was primarily driven by specific constraints on the SmartAge data. The dataset used in this study contains sensitive information that could, at the time, not be made publicly available due to data security laws. Consequently, cloud-based models like GPT4o, which require data to be uploaded to external servers via APIs or web interfaces, are unsuitable for our analysis. Llama3, as an open-source model, can be run locally, ensuring compliance with data privacy regulations while still allowing us to utilize the SmartAge dataset. Additionally, accessing GPT models, particularly GPT-4o, incurs costs which would have been too expensive for us.

Due to the poor results achieved and issues related to working with generative large language models as reported in RQ1.1, RQ2.1 and RQ3.1 in Section 6.2.4, we did not conduct any but the "Basic Classification" experiments with Llama.

#### Preprocessing Experiments

For the synthetic oversampling experiments, we applied both SMOTE (Chawla et al., 2002) and ADASYN (He et al., 2008) to the training data to create an artificially balanced dataset. We only report the results using SMOTE because ADASYN provided slightly worse results in all experiments.

The spellchecking experiments utilize the *LanguageTool*[2] python library to automatically correct spelling errors in the users' feedback. LanguageTool was chosen based on a comparative study by Näther et al. (Näther, 2020), which benchmarked 14 different spelling tools. Applying LanguageTool to our datasets resulted in a total of 906 corrections compared to the total of 95.967 words in the datasets. Clearly, the amount of incorrect spellings was much lower than expected from online feedback sources. We, nonetheless, included the experiments for completeness.

For the relevance classification, we utilized a definition of what constitutes relevant feedback from the work of Mekala et al. (Mekala et al., 2021), adapted it for the purpose of usage information classification, and manually removed sentences that did not meet the definition from the feedback. This removal was conducted by the author of this dissertation and a master's student during the course of their thesis (Knorr, 2024). In total, around 24% of the 6744 sentences in the datasets were removed as irrelevant. The definition for relevant feedback was as follows:

*"A review sentence is relevant if it mentions pertinent aspects of the App like features, bug reports, performance issues, etc., that would help developers to improve the App. A review*

[1] https://www.huggingface.co/dslim/bert-large-NER
[2] https://www.pypi.org/project/language-tool-python/

*sentence is also relevant if it refers to the domain in which the App is used, such as the role of the user, the tasks and activities in which the user utilizes the App, and the data associated with the activities. It is not relevant if it does not relate to the App, its functions or the domain in which the App is used or if it only contains user feelings, jokes, etc."*

#### Additional Experiments

Because the feedback collected in the SmartAge dataset was created by posing specific questions about the software to users, we could include the experiment "Include Questions as Feature". Here, the questions posed to users are used as additional classification features for their feedback. We investigate how this affects the classification results.

For the multi-stage classification (RQ3.2), we experiment with all classifiers (excluding Llama for reasons stated in Section 6.2.3) as a first stage to classify TORE Levels. These results are then fed as a classification feature towards a second-stage BERT-Large classifier, which decides the specific TORE Category. We additionally create Perfect-BERT, which simulates a perfect first-stage classifier, in order to see how dependent the second-stage classifier is on the performance of the first stage.

For the TORE Category specificity experiments (RQ3.5), we analyzed the individual class performance of our best-performing classifier to see which categories the classifier struggles to tell apart. We then combined the classes *Task & Activity* as well as *Interaction Data & Domain Data* and completely retrained the best single- and multi-stage classifier configurations with the combined classes using 5-fold cross-validation.

#### Experiment Setup

We ran our experiments using the MLFlow machine learning platform[3]. The complete source code, including the final hyperparameter setup and the code to reproduce all experiments, is provided in the online Appendix A. All training and testing was done on a desktop computer using an AMD Ryzen 7 7800X3D CPU and an NVIDIA RTX 3080Ti GPU using CUDA to facilitate GPU training.

To optimize the parameters for each classifier, we performed a grid search for each classification task. We ran experiments for each hyperparameter configuration with a data split of 80% training and 20% testing data across all available datasets for each task.

For all experiments except those related to RQ2.3 and RQ3.4 and all Llama classifications, we performed 5-fold cross-validation with the previously established best hyperparameter configuration. Due to multiple issues in working with Llama, explained in Section 6.2.3, we did not perform cross-validation for any Llama experiment. For the transferability experiments investigating RQ2.3 and RQ3.4, 5-fold cross-validation does not apply because we split training and test datasets among all combinations of feedback sources. We then train our best-performing classifier for each word-based task on one feedback source (i.e. Prolific, App Review or Forum) and test on another. Additionally, we perform leave-one-out-validation for the word-based tasks by training on two of our three datasets and testing on the third.

### 6.2.3. Prompt Engineering

#### Prompt Development

Working with generative LLMs like Llama requires the creation and refinement of prompts, which are used as input into the model (Section 2.3.7). These serve to describe the task the model is to complete and include the data on which the task is to be performed. These prompts can take

[3]https://www.mlflow.org/

different forms and are suited for different tasks depending on their intended usage. Ronanki et al. (Ronanki et al., 2024) propose five different patterns of prompts, namely "Cognitive Verifier", "Context Manager", "Persona", "Question Refinement", and "Template".

Due to the large amount of manual effort required to run the experiments (as explained in the following Section 6.2.3), we were unable to conduct the generative LLM experiments with all prompt patterns. Instead, we performed initial experiments with a 20% sampling of the data to identify the best-performing pattern for the UIC classification tasks with the patterns presented by Ronanki et al. We achieved the best results using the "Persona" pattern (see Section 6.2.4). Here, the model is instructed to behave like a specific role, in our case, a "requirements engineer". It is then presented with the task it is to complete.

Starting with an initial prompt, we refined the prompt further to achieve the output we desired from the model. Refinement of initial prompts is commonly used by other approaches like (Vogelsang and Fischbach, 2024) and (Rodriguez et al., 2023). In the first step, we added TORE definitions because the model was not familiar with TORE. These definitions were extracted directly from the coding rules used by the manual annotators and can be seen in Table 2.3.

Because the model was still performing the classification poorly and could not produce reasonable definitions of TORE when asked, we also added examples from the ground truth into the prompt. This helped both with understanding the classes as well as the desired output format. While the number of examples was initially low, we noticed that increasing the number of examples increased the reliability of the output both in terms of formatting as well as the correctness of the output classifications. As a result, for both sentence-based and word-based classification tasks, we included 2% of the data as examples in the prompt. We did not see further improvement after 2%. Classification by Llama was then done on the remaining data, which were not presented as examples.

Lastly, we added a more comprehensive description of the specific task for Llama, including a description of how the output by the model was to be formatted. The final prompt for both sentence- and word-based classifications used in the experiments can be seen in Figure 6.2.

#### Difficulties in Working with Generative LLMs

Despite the extensive refinement of the prompts, we encountered several problems with the classifications using Llama. The limit on a context length of 8192 tokens meant that we had to split our dataset into multiple pieces and feed every piece into the model in a new conversation. Exceeding context length by only a few tokens resulted in the model immediately generating nonsensical outputs (e.g. just outputting seemingly random chains of characters and punctuation). We were able to come close to the context length for the sentence-based classification, requiring fewer data splits. However, we found that inputting more than around 250 words for the classification of the word-based tasks resulted in the model only assigning a few codes (i.e. 1-2 codes in every other sentence). Consequently, data input for word-based tasks was limited to no more than 250 words per conversation.

Furthermore, despite being told not to change the text or leave out any parts, the model had a tendency to correct spelling, change contractions, or leave out whole sentences or words in all three tasks. This caused significant problems for the calculation of precision and recall because it made mapping the model's classification to the ground truth impossible. Model outputs had to be manually compared to the ground truth, and left-out statements had to be fed into the model again for re-classification. This meant a considerable manual effort.

This also meant that alternative ways of formatting the model's output were too unreliable. For example, asking the model only to output the index of a sentence or word next to its code would have sped up the testing. Given the tendency of the model to ignore some words or

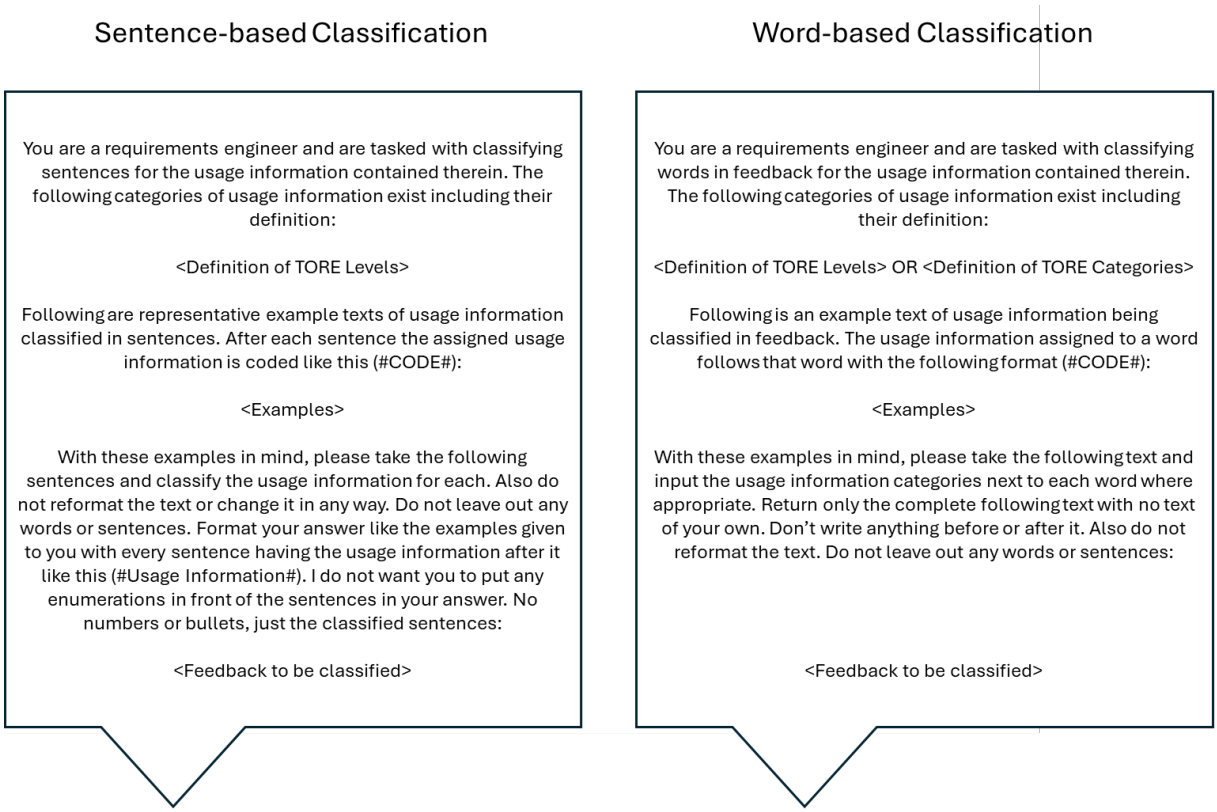



Figure 6.2.: Final Llama Prompts Used For TORE Classification

sentences, however, this would have been impossible, as there was no way of checking which index the model was matching to which sentence or word.

Due to these problems, especially the considerable manual effort and the low performance of the model as reported in Section 6.2.4, we decided to only perform the word-based classification tasks on the *App Review* dataset instead of all three datasets.

### 6.2.4. Results

In this section, we provide the results of all the research questions.

#### Sentence-based TORE Level Classification

***RQ1.1:*** *Which classifier performs best?*

Table 6.2.: Sentence-based TORE Level Classifier Results (RQ1.1 & RQ1.3)

| **Experiment** | **Model** | **Mean Prec** | **Mean Recall** | **Mean F1** |
|---|---|---|---|---|
| Basic Classification | **Llama3-70B** | 0.56 | 0.50 | 0.53 |
| | **RoBERTa-Large** | 0.71 | 0.71 | 0.71 |
| | **BERT-Large** | **0.72** | **0.75** | **0.73** |
| Incl. Questions as Features | **BERT-Large** | **0.78** | **0.77** | **0.77** |
| Synthetic Oversampling | **BERT-Large** | 0.71 | 0.72 | 0.71 |
| Spellchecking | **BERT-Large** | 0.72 | 0.75 | 0.73 |
| Relevance Classification | **BERT-Large** | 0.73 | 0.75 | 0.74 |

Table 6.2 shows the results for the basic classifications of sentence-based TORE Levels. The bottom three rows of the Table are discussed in RQ 1.3. Of the three tested classifiers, BERT-Large achieved the highest weighted mean F1 of 0.73. RoBERTa achieved almost identical but slightly lower results with F1 0.71. Llama has the lowest weighted F1 with 0.53. Including the questions posed to users when collecting feedback as a classification feature yields slight improvements for the BERT-Large classifier with F1 0.77 (+0.04).

Table 6.3.: Class Performance for Sentence-based TORE Level (RQ1.1 & RQ1.3)

| **Experiment** | **TORE Level** | **Mean Prec** | **Mean Recall** | **Mean F1** |
|---|---|---|---|---|
| BERT-Large Basic Classification | Domain Level | 0.74 | **0.85** | **0.79** |
| | Interaction Level | 0.57 | 0.46 | 0.51 |
| | System Level | 0.00 | 0.00 | 0.00 |
| | No Level | **0.83** | 0.76 | **0.79** |
| BERT-Large Binary Classifier | Domain Level | 0.72 | 0.79 | 0.75 |
| | Interaction Level | 0.62 | 0.46 | 0.53 |
| | System Level | 0.13 | 0.05 | 0.09 |
| | No Level | **0.84** | **0.83** | **0.84** |

The class-specific performance of the BERT-Large classifier can be seen in Table 6.3. We see a large disparity between the performance of the classes with more and less than 1000 occurrences in the dataset (Table 2.6). The *No Level* class (indicating that a sentence does not contain usage information relevant to any of the three levels) and the *Domain Level* class share an F1 score of 0.79. *Domain Level* has the highest mean recall, while *No Level* has the highest mean precision. F1 for the *Interaction Level* is lower. The *System Level* is not classified at all by the classifier. This is likely due to a lack of training data (only 61 occurrences).

**Answering RQ1.1:** BERT-Large achieves the highest performance for the sentence-based TORE Level classification with a weighted mean F1 of 0.73. This is increased further by including the questions posed to users as a classification feature. The *No Level* and *Domain Level* classes have the highest class performance with an F1 of 0.79.

***RQ1.2:*** *Does binary classification improve classification results?*

For binary classification, separate classifiers were trained to identify whether each sentence was on a specific TORE Level or not. The *No Level* binary classifier decides whether a sentence contains any usage information at all. The results are shown in Table 6.3. We compare this to the class-specific performance of the basic classification using the BERT-Large model shown in Table 6.3. The *No Level* classifier has the highest F1 of 0.84, which is 0.05 higher than that of the basic classification. The performance for the *Domain Level* is worse, while the performance for the *Interaction Level* is improved. The *System Level* is classified but with low performance.

**Answering RQ1.2:** Binary classification shows small improvement for three out of four classes. *Domain Level* however performs worse.

***RQ1.3:*** *Does preprocessing of the data improve classification results?*

The last three rows of Table 6.2 show the results of using preprocessing techniques before training and evaluation of the classifier.

SMOTE, as a method of synthetically oversampling the training data to achieve a balanced training set between the classes, did not increase the weighted mean F1 compared to the basic classification. Instead, it lowered the F1 score by 0.02. Applying spellchecking to the data did

not result in any changes in the metrics. The removal of irrelevant sentences only increased F1 by 0.01 due to a 0.01 increase in precision. Recall remained identical.

**Answering RQ1.3:** Synthetic oversampling leads to a slight reduction in performance metrics while spellchecking and relevance classification do not cause any changes larger than 0.01 to the metrics.

### Word-based TORE Level Classification

***RQ2.1:*** *Which classifier performs best?*

Table 6.4.: Word-based TORE Level Classifier Results (RQ2.1 & RQ2.2)

| Experiment | Model | Mean Prec | Mean Recall | Mean F1 |
|---|---|---|---|---|
| Basic Classification | **BERT-Large** | **0.81** | **0.78** | **0.79** |
| | **RoBERTa** | 0.80 | 0.76 | 0.78 |
| | **BERT-base** | 0.78 | **0.78** | 0.78 |
| | **SNER** | 0.76 | 0.70 | 0.73 |
| | **Bi-LSTM** | 0.72 | 0.65 | 0.68 |
| | **Llama3-70B** | 0.39 | 0.22 | 0.30 |
| Synthetic Oversampling | **BERT-Large** | **0.83** | **0.81** | **0.82** |
| Spellchecking | **BERT-Large** | 0.81 | 0.78 | 0.79 |
| Relevance Classification | **BERT-Large** | 0.82 | 0.78 | 0.80 |

While RQ1 used the SmartAge dataset for training and testing, RQ2 & RQ3 used the Prolific, Forum and App Review datasets. Table 6.4 shows the results of the six classifiers performing the basic word-based TORE Level classification. BERT-Large again shows the best F1 score, with 0.79 being slightly higher than the 0.78 of BERT-Base and RoBERTa. Both BERT-Large and BERT-Base are equal in recall at 0.78, but BERT-Large achieves a slightly higher precision with 0.81. All three non-BERT models perform worse, with the lowest metrics for Llama3-70B.

Table 6.5.: Class Performance for Word-based TORE Level (RQ2.1 & RQ2.2)

| Experiment | TORE Level | Mean Prec | Mean Recall | Mean F1 |
|---|---|---|---|---|
| BERT-Large Basic Classification | Domain Level | 0.76 | 0.77 | 0.77 |
| | Interaction Level | 0.69 | 0.71 | 0.70 |
| | System Level | 0.84 | **0.81** | 0.82 |
| | No Level | **0.95** | **0.81** | **0.88** |

The class-specific performance of the basic classification using BERT-Large is shown in Table 6.5. Of the three TORE Levels, the *System Level* has the highest F1 with 0.82. This is followed by the *Domain Level* and the *Interaction Level*. The *No Level* class has the highest performance of all four classes with an F1 of 0.88. Note that in word-based annotation, this class makes up the vast majority of the dataset, as most words will not be assigned to any category.

**Answering RQ2.1:** The best word-based TORE Level classification is performed by BERT-Large with an F1 of 0.79. The *No Level* class performs best, followed by the *System Level*.

***RQ2.2:*** *Does preprocessing of the data improve classification results?*

The results of the preprocessing experiments are shown in the last three rows of Table Table 6.4. Using SMOTE for synthetic oversampling improves the results slightly for all metrics (F1 +0,03). Again, as in RQ1, spellchecking does not cause changes to the performance metrics. Relevance classification improves the precision by 0.01, leading to a 0.01 increased F1.

**Answering RQ2.2:** Synthetic oversampling helps for the basic classification while spellchecking and relevance classification cause no changes larger than 0.01.

***RQ2.3:*** *Are classification results transferable across feedback sources?*

Table 6.6.: BERT-Large Word-Based TORE Level Transferability Results (RQ2.3)

| Training Dataset | Test Dataset | Precision | Recall | F1 |
|---|---|---|---|---|
| Prolific | Forum | 0.55 | 0.58 | 0.57 |
| Prolific | App Review | 0.71 | 0.59 | 0.65 |
| App Review | Prolific | 0.71 | 0.69 | 0.70 |
| App Review | Forum | 0.52 | 0.54 | 0.53 |
| Forum | Prolific | 0.61 | 0.57 | 0.59 |
| Forum | App Review | 0.61 | 0.51 | 0.56 |
| Prolific & App Review | Forum | 0.50 | 0.66 | 0.58 |
| Prolific & Forum | App Review | **0.72** | 0.68 | 0.70 |
| App Review & Forum | Prolific | 0.70 | **0.78** | **0.74** |
| Average | | 0.63 | 0.62 | 0.62 |

Table 6.6 shows the precision, recall and F1 values for all transferability experiments of the best-performing BERT-Large classifier. It also lists the average across all experiments in the last row. The experiments shown represent all possible combinations of training and testing data for the three datasets available with word-based coding. Because training and testing data are separate, there is no cross-validation for these as the entire datasets were used for training and testing respectively.

Results are consistently lower than those of the BERT-Large classifier (F1 score 0.79) trained and evaluated on the combination of all datasets (see Table 6.4). The best-performing combination of training and testing data by the recall and F1 value is training on App Review and Forum and testing on Prolific data. When only training and testing on one dataset, all combinations, including the Forum dataset, perform worse than those without Forum data. When training on two datasets, only testing on Forum shows worse performance. Training with Forum and another dataset results in increased performance compared to only training on the forum dataset.

**Answering RQ2.3:** Word-based TORE Level classification does not transfer well across different feedback sources. The datasets transfer better if Forum data is not involved.

### Word-based TORE Category Classification

***RQ3.1:*** *Which classifier performs best?*

Table 6.7 shows the mean precision, recall and F1 values for all six tested classifiers for the word-based TORE Category basic classification. BERT-Large and RoBERTa perform best. Both have almost identical metrics, differing only by 0.01 in precision and recall. They are followed by BERT-Base, with SNER and Bi-LSTM behind. The worst performance by far is achieved by Llama3-70B, with an F1 score of only 0.16. Since BERT-Large and RoBERTa are almost

Table 6.7.: Word-based TORE Category Classifier Results (RQ3.1 & RQ3.3)

| **Experiment** | **Model** | **Mean Prec** | **Mean Recall** | **Mean F1** |
|---|---|---|---|---|
| Basic Classification | **BERT-Large** | 0.67 | **0.67** | **0.67** |
| | **RoBERTa** | **0.68** | 0.66 | **0.67** |
| | **BERT-base** | 0.66 | 0.64 | 0.65 |
| | **SNER** | 0.63 | 0.52 | 0.57 |
| | **Bi-LSTM** | 0.58 | 0.51 | 0.54 |
| | **Llama3-70B** | 0.21 | 0.12 | 0.16 |
| Synthetic Oversampling | **BERT-Large** | 0.68 | 0.65 | 0.66 |
| Spellchecking | **BERT-Large** | 0.67 | 0.67 | 0.67 |
| Relevance Classification | **BERT-Large** | 0.68 | 0.66 | 0.67 |
| Multi-Stage | **BERT-BERT-Large** | 0.67 | 0.66 | 0.67 |
| | **SNER-BERT-Large** | 0.64 | 0.58 | 0.61 |
| | **Bi-LSTM-BERT-Large** | 0.56 | 0.67 | 0.61 |
| | **Perfect-BERT-Large** | 0.88 | 0.85 | 0.86 |
| Class Specificity | **BERT-Large Comb. Class** | 0.74 | 0.74 | 0.74 |
| | **BERT-BERT-Large Comb. Class** | 0.74 | 0.73 | 0.74 |

identical in performance, we performed further experiments with BERT-Large as it had slightly higher performance in the previous two classification tasks.

Table 6.8.: Class Performance for Word-based TORE Category (RQ3.1, RQ3.2, RQ3.3, RQ3.5)

| Experiment | Metric | Stake-holder | Task | Activity | Domain Data | Inter-action | System Function | Interaction Data | Work-space | System | No Category |
|---|---|---|---|---|---|---|---|---|---|---|---|
| BERT-Large Basic Classification | Mean Prec | **0.85** | 0.47 | 0.48 | 0.53 | 0.67 | 0.60 | 0.69 | 0.67 | 0.79 | 0.95 |
| | Mean Recall | **0.94** | 0.43 | 0.37 | 0.60 | 0.77 | 0.45 | 0.66 | 0.69 | 0.79 | 0.96 |
| | Mean F1 | **0.90** | 0.45 | 0.43 | 0.56 | 0.72 | 0.53 | 0.68 | 0.68 | 0.79 | 0.95 |

Table 6.8 shows the individual class performance of the BERT-Large classifier for the basic classification. We can see that results for TORE Categories are very different. *Stakeholder*, for example, is the most accurately classified TORE Category with an F1 of 0.90. *Task* has the lowest precision of 0.47 and *Activity* the lowest recall of 0.37. The *No Category* class has the highest performance with 0.95 F1. Note that in word-based annotation, this class makes up the vast majority of the dataset, as most words will not be assigned to any category.

**Answering RQ3.1:** BERT-Large and RoBERTa perform the word-based TORE Category classification almost identically well with an F1 of 0.67. There is a large disparity between the performance of individual classes.

***RQ3.2:*** *Does multi-stage classification improve results compared to single-stage classification?*
The third section of Table 6.7 shows the results of the multi-stage classification. All classifiers use a BERT-Large second stage. Between the three classifier combinations SNER-BERT, Bi-LSTM-BERT and BERT-BERT, we see that the classifier with BERT-Large as the first-stage

TORE Level classifier outperforms the SNER and Bi-LSTM first-stage in F1. Only the mean precision is almost identical between Bi-LSTM-BERT and BERT-BERT. This is interesting, as the Bi-LSTM word-based Level classifier does not outperform the BERT word-based Level classifiers.

We do not see any remarkable differences between the multi- and single-stage performance of the classifiers. All metrics between the best-performing multi-stage BERT-BERT and best-performing single-stage BERT-Large are almost identical.

Table 6.7 also reports the metrics for the Perfect-BERT classifier, which simulates a perfect first-stage classifier as input for the second-stage BERT-Large classifier. We see remarkably higher values when compared to the other multi-stage as well as the single-stage classifiers, with a mean F1 value of 0.86.

**Answering RQ3.2:** Multi-stage classification does not result in improvement over single-stage classification. Perfect-BERT, however, shows the potential of multi-stage given further improvement in first-stage classifiers.

***RQ3.3:*** *Does preprocessing of the data improve classification results?*

Table 6.7 shows the results of the preprocessing experiments in the second section. Comparing the results of the synthetic oversampling experiment for BERT-Large with the basic configuration, we see no real difference between the two, with F1 only differing by 0.01. Again, spellchecking does not result in any changes to the metrics. Relevance classification also does not affect the F1 metrics, raising precision by 0.01 and lowering recall by 0.01.

**Answering RQ3.3:** Overall synthetic oversampling for the word-based TORE Category classification only shows marginal differences to the classification without oversampling. Spellchecking and relevance classification also do not affect the classifier's performance.

***RQ3.4:*** *Are classification results transferable across feedback sources?*

Table 6.9.: BERT-Large Word-Based TORE Category Transferability Results (RQ3.4)

| Training Dataset | Test Dataset | Precision | Recall | F1 |
|---|---|---|---|---|
| Prolific | Forum | 0.42 | 0.44 | 0.43 |
| Prolific | App Review | 0.59 | 0.50 | 0.55 |
| App Review | Prolific | 0.56 | 0.56 | 0.56 |
| App Review | Forum | 0.39 | 0.43 | 0.41 |
| Forum | Prolific | 0.46 | 0.44 | 0.45 |
| Forum | App Review | 0.47 | 0.39 | 0.43 |
| Prolific & App Review | Forum | 0.41 | 0.44 | 0.43 |
| Prolific & Forum | App Review | **0.61** | 0.46 | 0.54 |
| App Review & Forum | Prolific | 0.60 | **0.57** | **0.59** |
| Average | | 0.50 | 0.46 | 0.49 |

Table 6.9 shows the precision, recall and F1 values for all TORE category transferability experiments of the best-performing BERT-Large classifier. It also lists the average across all experiments in the last row. The experiments shown represent all possible combinations of training and testing data for the three datasets available with word-based coding. Because training and testing data are separate, there is no cross-validation for these as the entire datasets were used for training and testing respectively.

Results are consistent with those reported for the TORE Level transferability in RQ2.3. Again, the best-performing combination of training and testing data by the recall and F1 value is training on App Review and Forum and testing on Prolific data. Also, all combinations, including the Forum dataset, again perform worse than their counterparts.

**Answering RQ3.4:** Word-based TORE Category classification does not transfer well across different feedback sources. The datasets transfer better if Forum is not involved.

***RQ3.5:*** *Does the specificity of the TORE Categories influence classification results?*

Figure 6.3 shows the normalized confusion matrix for the best performing BERT-Large word-based TORE Category single-stage classifier on the left. Each row represents the percentages of predicted labels for every true label. *Activity* for example was correctly labeled 37% of the time and mislabeled as *Task* 15% of the time. The brighter the colour in the off-diagonal elements, the higher the confusion.

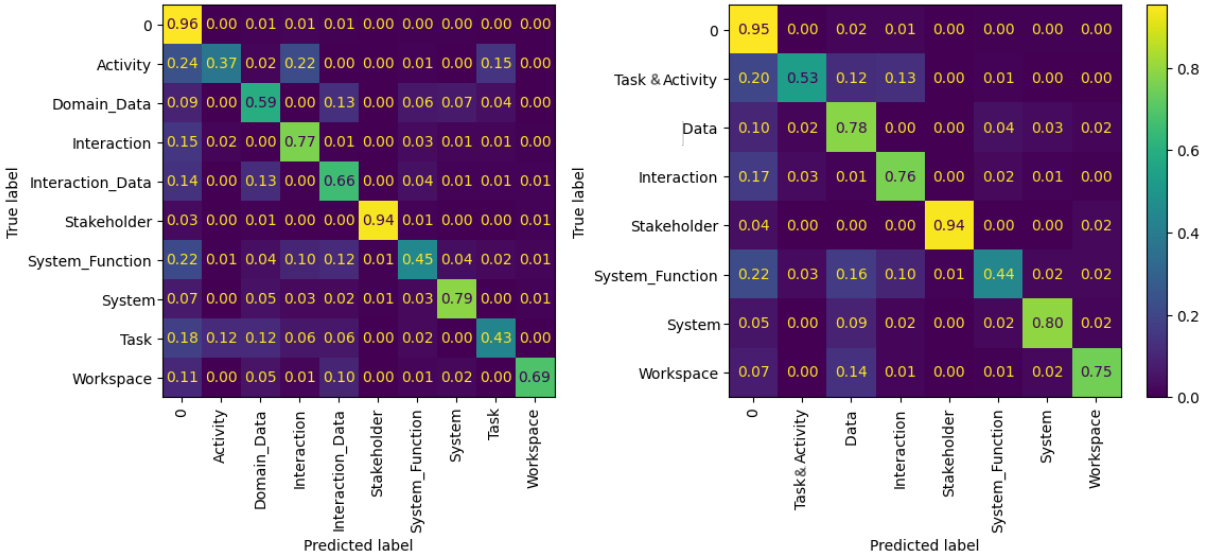



Figure 6.3.: Normalized Confusion Matrix BERT Single-Stage original (left) and combined classes (right)

Ignoring the default label 0, where no class is assigned to a word, the *Stakeholder* is the least confused class in the datasets. We see higher confusion between the *Activity, Interaction* and *Task* classes as well as the *Interaction Data* and *Domain Data* classes. The matrix indicates that the classifier is not able to tell these classes apart consistently. Therefore, we combined the *Activity* and *Task* categories as well as the *Interaction Data* and *Domain Data* categories and re-run the 5-fold cross-validation experiments for BERT-Large single-stage and BERT-BERT multi-stage. We did not combine the *Interaction* category with *Activity* despite the higher confusion because we see this distinction as essential when extracting usage information in such specificity.

The right side of Figure 6.3 shows the resulting confusion matrix of the BERT-Large single-stage, where the classes have been combined. The "Data" category represents the combined *Interaction Data & Domain Data* categories. The combined categories show an increase in the number of correctly assigned labels. The *Data* class is not misclassified as any other class by more than 4%. The *Task & Activity* class has a 0.16 improvement compared to *Activity* and *Task* as separate classes.

The metrics achieved by the BERT-Large single-stage and multi-stage classifiers with the combined classes can be seen at the bottom of Table 6.7. The combined class classifiers outperform BERT-Large basic classification in all metrics. When comparing combined class BERT-Large single-stage with BERT-BERT multi-stage, we see almost identical precision, recall and F1.

**Answering RQ3.5:** The results are affected by the specificity of the classes. Less fine-grained data and domain categories improve classification results overall and reduce confusion between classes.

### 6.2.5. Discussion

This section discusses the results of the non-generative and generative LLMs and the effects that different granularities have on the models' performance.

#### Non-Generative LLM Performance

For the word-based TORE Category classification BERT-Large as well as RoBERTa performed best (RQ3.1). Looking at the category-specific performance, there are large differences between individual categories. Interestingly, both the highest and lowest performing categories are on the *Domain Level*, while the *Interaction Level* is closer to the mean F1 values. The categories on the *Domain Level*, with the exception of *Stakeholder*, appear to be the hardest to accurately classify. The *System* category performs second best.

**Multi-stage classification** does not improve performance compared to single-stage classification (RQ3.2). A possible reason for this could be the dependencies between the two stages created by the multi-stage approach. The results of the second stage are directly dependent on how the first stage is classified. This means that misclassifications in the first stage, such as an *Interaction Level* being erroneously classified as a *Domain Level*, are compounded in the second stage as the categories can now not be correctly assigned. Data imbalance may also contribute to the lack of multi-stage improvement. As Table 2.7 shows, the datasets are not only imbalanced in the categories but also on the levels. This means that the first-stage classifier may already be affected by this imbalance, which is then compounded further during the second-stage classification. Nonetheless, Perfect-BERT demonstrates the potential for improvement compared to single-stage classification. To improve results, one could manually annotate the first-stage classification and then automatically extract detailed usage information in the second stage.

The investigation into the **specificity of the TORE Categories** (RQ3.5) showed that the classifier struggles to correctly classify semantically related categories like *Domain Data* and *Interaction Data*. For both categories, often the same words are used (e.g. "to plan a *hiking route*[Interaction Data]" and "to go on a *hiking route*[Domain Data]"). The distinction between these categories is often dependent on the context of the complete sentence they are used in. Combining these categories reduced confusion and improved overall results. Fine-grained classification, such as distinguishing between "Tasks" and "Activities" or *Domain Data* and *Interaction Data*, presents significant challenges due to linguistic ambiguities inherent in user feedback and data sparsity for certain categories. Linguistic ambiguities arise when users use imprecise language, making it difficult for the classifier to assign feedback to a single, distinct category. Users often use the same words to describe different aspects of the software, often obfuscating the lines between domain and interaction (Anders et al., 2022). Additionally, data sparsity exacerbates the problem, as certain categories may be underrepresented in the training data, limiting the model's ability to learn nuanced distinctions.

The improvement with less specific classes can also be observed when using TORE Levels instead of TORE Categories (RQ2.1). The less fine-grained TORE Level classification improves

performance by 0.12 in F1 compared to TORE Categories. Also, class-specific performance was closer than that of word-based TORE Category classification. TORE Level performance only differs by an F1 value of 0.12, the lowest difference of the three granularities (sentence-based TORE Level, word-based TORE Level and word-based TORE Category).

**Transferability** across feedback sources is low (RQ2.3, RQ3.4). This is in line with the findings of other works (Novielli et al., 2020) (Devine and al., 2023). A primary reason for these challenges may be differences in language style and tone between datasets discussed in Section 2.1.3. These variations in language style and structural organization could explain why transferability remains a challenge for our classifier. Our assumptions that including datasets from multiple different sources improves the robustness of the classifier are confirmed by the transferability experiments. F1 is consistently higher in experiments where two datasets are used for training and one for testing, compared to experiments where the same dataset is used for testing but only one is used for training. Including datasets from different sources helps the classifier's transferability on unseen data.

**Decreasing the granularity** further from word- to sentence-based TORE Level classification (RQ1.1) did not yield improved results. Our highest mean F1 value achieved is 0.77 by including the questions posed to users as a classification feature. Without these questions, the performance drops to a mean F1 of 0.73. However, we used different datasets for word-based and sentence-based classifications, which made the comparison difficult.

For **preprocessing**, the attempts to improve classification results by introducing synthetic oversampling (RQ1.3, RQ2.2, RQ3.3) only led to very slight improvements for the word-based Tore Level classification (+0.03 F1). The other two granularities saw small decreases in performance with synthetic oversampling. For the word-based classifications, this seems to indicate that the poor performance of some categories is not correlated with the number of occurrences in the dataset.

Spellchecking did not cause any changes in the performance metrics, likely due to the much lower amount of incorrect spelling than was expected. As mentioned in Section 6.2.2, only around 1% of the words in our datasets were incorrectly spelled. Given these low numbers, we also did not see any differences between the amount of incorrectly spelled across the different datasets. No dataset source, be it forum, app store, questionnaire or feedback app, had considerably more misspellings than another.

Relevance classification also did not yield any improvement for any of the granularities, even though around 24% of the datasets' sentences were removed as irrelevant. The likely reason for this is, that the classifiers on every granularity are already well capable of identifying whether a sentence or word contains no usage information. This can be seen in the *No Category* and *No Level* classes of the class-specific performance tables 6.3, 6.5 and 6.8. Irrelevant sentences tend to contain less usage information. Consequently, classifiers are already capable of "ignoring" these sentences even without their removal.

**Comparing related work** to our results, our highest mean F1 value achieved for the basic classifications is 0.79 for the word-based TORE Level classification using BERT-Large. As shown in Figure 4.3 in Section 4.2.2, other related work reports F1 values between 0.54 and 0.98, with a median of 0.82. This indicates that our classifiers' performance is on par with those of other existing fine-grained automatic classifiers while only being slightly lower than the median. This comparison, though, is difficult due to the very inconsistent reporting of performance metrics across the related work.

#### Generative LLM Performance

As discussed in Section 6.2.3, working with Llama posed considerable challenges due to its unreliability in following commands, especially for the word-based classifications. The classifica-

tion results are also considerably lower than that of the other tested classifiers (RQ1.1, RQ2.1, RQ3.1). Of the three tasks, Llama performed best in the sentence-based classification task while still being considerably lower than the alternative classifiers. A qualitative analysis of the classifications done by Llama showed that it struggled to understand the TORE Levels and categories correctly. Llama seemed to even misclassify categories that human coders did not have a problem distinguishing. We tried to make these categories more clear by including up to 2% of the entire dataset as an example for Llama. We also ensured that the examples given were representative of the data. While Llama was able to classify short sentences more accurately, especially those containing no usage information, longer sentences were mostly assigned labels seemingly randomly. Asking Llama for explanations on why a certain label was assigned also did not provide meaningful insights, as the model would either start discussing the sentiment of a sentence, which does not factor into usage information classification, or it would apologize for its mistake and assign a new label to the sentence without being prompted to do so. Reproducibility of classification was also nonexistent, as the same set of sentences or words would be classified differently almost every time they were input into Llama.

### 6.2.6. Threats To Validity

This section discusses the threats to the validity of the usage information classification experiments.

#### Reliability

The reliability of the results is threatened because of the marginal differences observed between classifiers across many experiments, often only ranging from 0.01 to 0.05. Such small variations make it challenging to draw definitive conclusions about which classifier performs best under specific conditions. This also affects successive experiments where the best-performing classifier (BERT-Large) was used instead of alternative classifiers. To mitigate the risk of our results being influenced by chance, we employed 5-fold cross-validation whenever feasible. The absence of statistical testing, however, introduces a limitation. Without formal statistical analysis, we cannot confirm whether the observed differences in performance are genuinely significant or merely the result of random variation.

#### Construct Validity

A threat to the construct validity lies in the evaluation of the classifiers' performance using the selected metrics: precision, recall, and F1. As detailed in Section 2.3.8, these metrics were chosen for their widespread use in assessing machine learning classifiers. The use of different metrics might have led to other observations which we did not make.

Another threat to the construct validity is the fact that all results reported for LLama were achieved with the prompts reported in Section 6.2.3. While we carefully revised our prompts, due to the non-discriminative nature of generative LLMs, the use of different prompts can achieve different results from those reported in this dissertation.

#### Internal Validity

One threat to internal validity is the implementation of the classifiers. To alleviate this, we utilize widely used libraries such as Huggingface and performed our testing with an established machine learning experiment management platform in the form of MLFlow. The code is also publicly available so results can be replicated.

We alleviated the internal threat due to the manual coding by employing multiple coders for every dataset and ensuring that every document was independently coded by two people, after which an inter-rater agreement was established.

Because we used different datasets for word-based and sentence-based granularities, another threat to the internal validity is introduced. This makes drawing conclusions on whether differences in performance are due to the change in granularity or due to dataset-specific characteristics difficult. This lack of consistency in datasets introduces a confounding factor, making it difficult to attribute performance changes solely to class granularity.

#### External Validity

Threats to the external validity are mainly due to the datasets. Almost all data in the word-based datasets concerns one software product, Komoot. For unrelated reasons, Komoot was chosen during our initial design of the questionnaire for the Prolific dataset (Anders et al., 2023). While this helped us to better analyze the transferability across feedback sources in RQ2.3 and RQ3.4, it poses a threat towards the generalizability of our classifiers on unseen data of different software products. Given the domain dependence of the classification, especially on the Domain Level, we don't expect similar performance of our classifier in different domains. This would require retraining on data from unseen domains. The generalizability of the sentence-based classifier is also not tested, as we only have a single dataset available for this task.

The Forum dataset partially contains feedback from other software products. This, however, introduces another threat, especially for the transferability investigation in RQ2.3 and RQ3.4, as it is the only one that is not solely about Komoot.

The Prolific dataset presents another inherent threat to online survey platforms. Users are anonymous participants who are paid for their participation. This could result in them feeling the need to give feedback that is not representative of their true opinions. To alleviate this to some degree, answers were checked for originality. Answers found to be copied from online sources (even those lightly changed from these sources) were excluded from the dataset.

The study context in which the SmartAge dataset is gathered presents a similar threat. The feedback is gathered by participants of the study, who were selected based on their age and the location in which they lived. While these participants were not paid, they did receive tablets on which they used the apps. This could influence the feedback they give.

A potential threat to the validity of our findings also stems from the diverse nature of the input data sources. These data types differ in terms of context, participant engagement, response format, and potential biases. The heterogeneity in data sources introduces variability that could impact the generalizability of our models, as they may perform differently depending on the source of the input.

## 6.3. UIC Conclusions

Based on the results reported in this Chapter, automatic word-based TORE Level classification works better than the other two granularities investigated (RQ2.1). It achieves the highest F1 value and the lowest difference in individual class performance. As discussed in Section 6.1.4, looking at word-based usage information can allow developers to see whether their software covers the users' needs in their daily life and their needs when using the software. Daily life needs are expressed on the *Domain Level* and can show whether the software provides the support users need to accomplish their actions. Software needs are expressed on the *Interaction and System Level* and cover whether the software behaves in the way users expect it to behave. When finer usage information is required, using the word-based TORE Category classifier with combined classes (RQ3.5) is recommended. The classifier's performance values are 0.05 lower

than the word-based TORE Levels, but a more fine-grained insight is provided. A possible use case for finer usage information being needed is if developers want to identify the different types of stakeholders that use their software, which the *Stakeholder* TORE Category offers. The effectiveness of the classifier is evaluated in the treatment validation in Chapter 9.

Chapter 7

# Feed.UVL

This chapter presents the Feed.UVL tool and its related Jira plugin developed in this dissertation. Feed.UVL supports developers by providing a web tool which supports feedback collection, management, analysis, and visualisation through its functionalities and views. Feed.UVL also serves as a platform to implement feedback analysis approaches, such as the feedback requirements relation (Chapter 5) and usage information classification (Chapter 6) presented in this dissertation. The tool was developed as part of the design of the FeReRe and UIC approaches in *design goal 1* and *design goal 2* (see Figure 1.2). As such, the functionalities provided help tackle P1 and P2 as specified in Figure 1.1 by providing a platform for developers to manage and execute the approaches.

Many of Feed.UVL's functionalities were developed as part of practicals, bachelor and master theses supervised by the author of this dissertation. A list of contributing theses and reports can be found in Table 1.

Section 7.1 describes Feed.UVL's architecture and gives a rough overview of its functionalities. Section 7.2 describes the functionalities in detail, including their requirements and implementation. A domain data diagram and a UI structure diagram for Feed.UVL can be found in Appendix C.2

Feed.UVL is an open-source tool whose code is available in the Appendix A. The Appendix also contains a complete list of all of Feed.UVL's requirements, which were originially documented in Jira. This chapter only contains a select number of requirements for each functionality because the total number of 731 Jira issues, which represent Feed.UVL's complete documentation (including test cases, bug reports and mockups) would exceed the scope of this chapter.

## 7.1. Feed.UVL Architecture

In this section, the foundational tool on which Feed.UVL was developed is introduced. Then the microservice and deployment architecture on which Feed.UVL is built, is explained and a short overview over Feed.UVL functionalities is provided.

### 7.1.1. Foundation

Feed.UVL's basic architecture is based on a tool called Feed.ai, which was developed during the OpenReq[1] project as part of a dissertation at the University of Hamburg (Stanik, 2020). This work laid the foundation for the general architecture of Feed.UVL.

Feed.ai was developed as a platform to gather and analyse Twitter feedback for individual software products. The main view for this was a dashboard, which can be seen in Figure

[1] https://www.openreq.eu/

7.1. The tool gathers tweets sent to predefined Twitter accounts, classifies them into three categories (problem report, inquiry and irrelevant), analysis their sentiment and then visualizes the frequency of tweets and the classification results. The tool provides further functionalities for manual correction of automatic classification and comparisons between multiple Twitter accounts. Further functionalities not relevant to this dissertation were developed and are explained in the foundational dissertation for Feed.ai (Stanik, 2020).

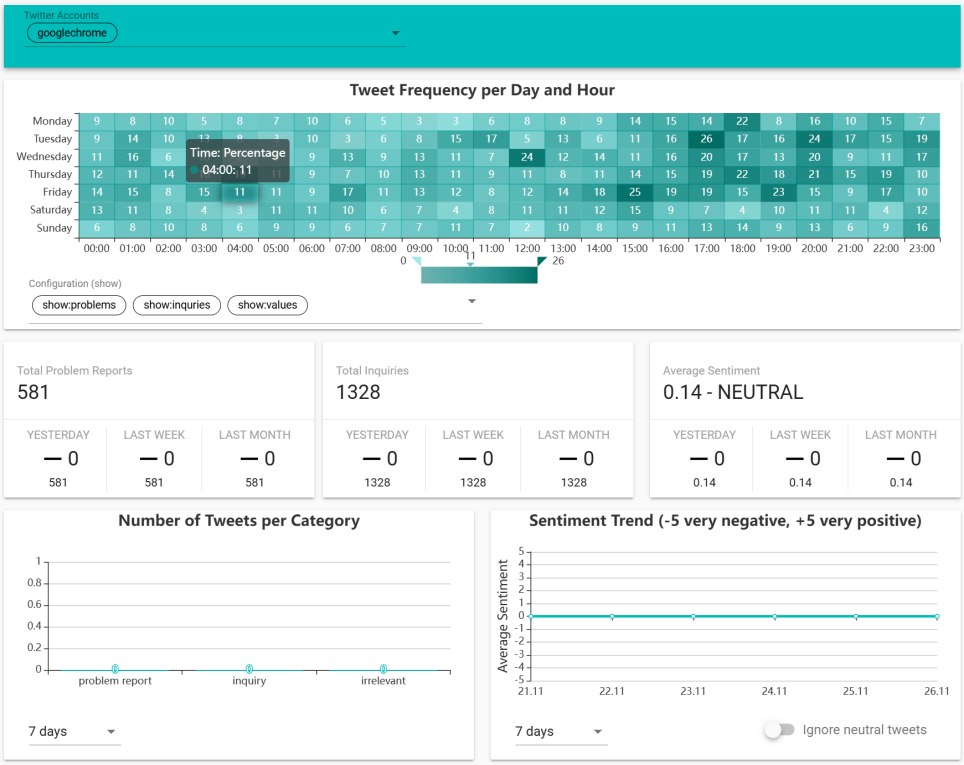



Figure 7.1.: Feed.ai Dashboard

The decision was made to utilize Feed.ai as a basis for the development of Feed.UVL in order to promote reuse and to benefit from the fact that Feed.ai had already been evaluated as part of the underlying dissertation. It should be noted that while Feed.UVL's core architecture is based on Feed.ai almost all of its functionalities are significant further developments from the initial scope for which Feed.ai was developed. Both tools can function completely independently from one another. After four years of development on Feed.UVL only its backend architecture, and the general frontend design matches that of Feed.ai.

### 7.1.2. Microservice Architecture

Feed.ai and, as a result, Feed.UVL is based on a microservice architecture. Microservice architectures are a design approach in software development where an application is structured as a collection of independent services that communicate with each other. Each service is designed to perform a specific function which operates independently. This allows it to be developed, deployed and maintained separately from other parts of the system. In the case of Feed.UVL, these independent services communicate over RestAPIs to exchange data.

This architecture contrasts with the traditional monolithic approach (Al-Debagy and Martinek, 2018) (Tapia et al., 2020), where an application is built as a single unit. In a monolithic system, all components are tightly coupled, making changes or scaling a particular functionality more challenging. Microservices, on the other hand, enable a more flexible approach to development.

It allows a focus on specific services without impacting the entire system. The main advantages for Feed.UVL stemming from the microservice architecture are a *flexibility in technology*, *faster deployment* and better *fault isolation*.

**Flexibility in technology:** Because each service operates independently, developers can use the technology (e.g. programming languages, frameworks) that suits them best. Feed.UVL's services, for example, are developed in five different programming languages. While the frontend is written in VueJs, the service orchestrating the communication between services is written in GO and its functionalities are written in either Python, C++ or Java. Theoretically, these functionalities could be written in almost any other programming language.

**Faster deployment:** Microservices enable parallel development, as different developers can work on separate services simultaneously. This allows the tool to be developed in parallel, with one developer not interfering with another. Additionally, changes or updates to one service can be deployed without affecting others, leading to quicker iterations and more reliable deployment pipelines. Developers can easily test any changes without having to rebuild the entire software.

**Fault isolation:** In a microservice architecture, the failure of one service is less likely to bring down the entire system. This isolation improves the overall resilience of Feed.UVL. The modular nature of microservices also simplifies debugging and maintenance. Smaller codebases are easier to understand, test, and modify, which enhances long-term maintainability.

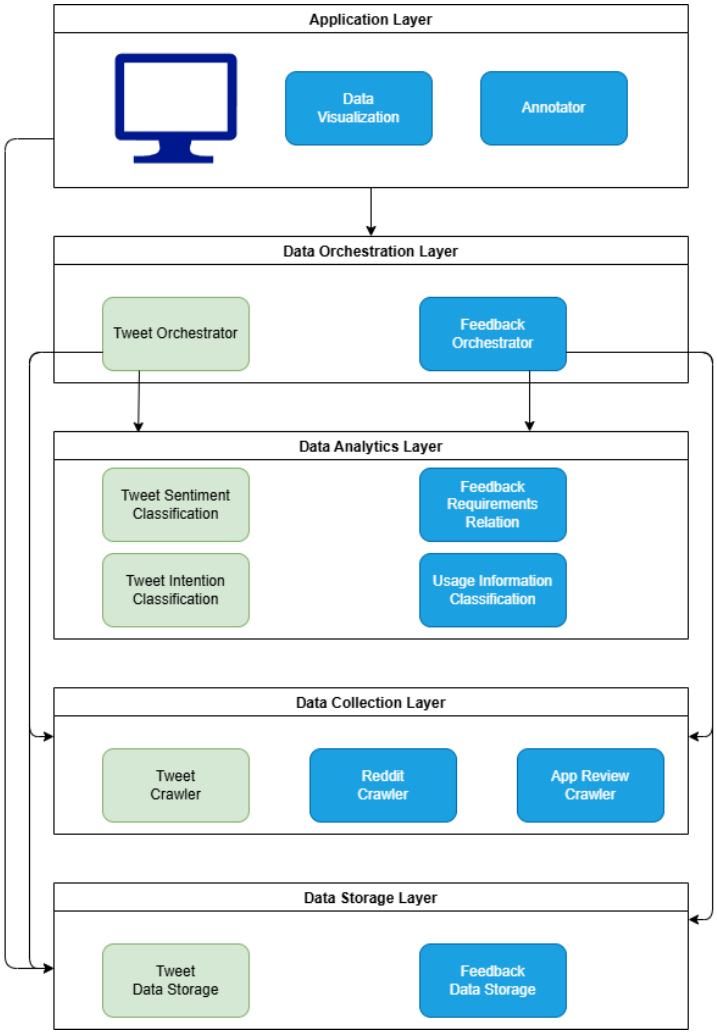



Figure 7.2.: Feed.UVL Architecture Overview

Feed.UVL microservices can be grouped into five different layers, as can be seen in Figure 7.2. The left side in green shows Feed.ai's core services. The right side in blue shows Feed.UVL's core services. As mentioned before, both Feed.ai and Feed.UVL are designed to function independently from one another, leading to multiple similar services on some of the layers.

On the **Application Layer**, the graphical interface is provided for users. As a web-based tool Feed.UVL is accessible through almost all web browsers even scaling appropriately for mobile devices. The GUI provides visualisations of the datasets and classifiers in Feed.UVL as well as the possibility to interact with the tool's manual annotation functionalities.

The **Data Orchestration Layer** handles communication between the backend services on the lower layers and the application layer. Through these orchestrators, all microservices are available through API requests. Along with the basic frontend service, which provides the overall UI on the application layer, the services on the data orchestration layer, together with the storage service on the *Data Storage Layer*, are the only non-modular parts of Feed.UVLs architecture because they are necessary to facilitate communication between the other services.

In the **Data Analytics Layer** microservices are provided which analyse user feedback. Each microservice here serves a different analysis purpose and each can be seen as a standalone machine learning classifier, independent of any existing or future classifier present in the software. Each service takes raw feedback data and performs its analysis, which then again can be displayed on the application layer. Splitting each classifier into its own service has the further advantage of allowing administrators to limit the hardware resources each classifier can access. This prevents any one classifier from taking up all available resources, affecting the availability of the tool or leading to crashes.

The **Data Collection Layer** contains so-called crawlers. These are services which collect explicit feedback from different software feedback sources. This feedback is then stored in the form of datasets in Feed.UVL and can be accessed, downloaded or analysed further in the tool.

Lastly, the **Data Storage Layer** is responsible for data persistence. All data within the tool, be it feedback datasets, analysis results or annotations, are stored in a database. To prevent any loss of information or possible racing conditions, all accesses to the storage service happen via the orchestrators on the data orchestration layer. This allows the orchestrator to manage the order in which the database is accessed.

### 7.1.3. Deployment Architecture

Feed.UVL trades the complexity of a difficult one-time setup with easy maintenance and the previously mentioned advantages of microservice architectures 7.1.2. Most of the tools mentioned in this section have already been introduced in Section 2.4.

Fundamentally Feed.UVL is deployed on a web-server running a Debian[2] Linux instance. Although theoretically, the application could be run on any operating system. Running a web-server provides continuous, parallel access to the application through any web browser, without requiring deployment every time the software needs to be used. Accesses to Feed.UVL's websites, as well as related tools like Jenkins and Portainer (Section 2.4), are handled through the reverse proxy Traefik, running on the server.

The microservice architecture of Feed.UVL is realized by implementing every functionality in Feed.UVL as its own Docker container. These are configured, built and deployed through Jenkins. Jenkins also handles the continuous deployment of these services by automatically rebuilding containers when changes are pushed to their respective Github Repositories[3]. The logging functionalities provided by Jenkins also help with debugging any build errors that the

[2] https://www.debian.org/
[3] https://www.github.com/feeduvl

containers might have during development. Jenkins is accessed through it's own website provided by the web-server Feed.UVL is running on.

Once the containers are built and deployed by Jenkins, they can be managed through an instance of Portainer. Just as Jenkins, Portainer can also be accessed through its own provided website. This allows for continuous, remote monitoring of Feed.UVL. Portainer additionally provides functionalities to schedule containers, perform cleanup operations on the server, log all containers and access them individually.

Feed.UVL's database is running MongoDB[4], a NoSQL database also running on Feed.UVLs web-server. Once the initial configuration of the server is done, all deployment tools (Jenkins, Portainer, MongoDB) can be started through a single Docker file, which is configured to also rebuild all necessary containers after a server restart. This, along with a Jenkins pipeline, which rebuilds all Feed.UVL services allows quick recovery after any server failure and reduces the need for any human maintenance.

### 7.1.4. Functionality Overview

Feed.UVL provides functionalities for feedback collection, management, analysis and visualization. The requirements and implementation, along with detailed screenshots for each functionality, are documented in the sections listed below. These functionalities can be grouped into the following categories:

- Feedback Collection (Section 7.2.1)
- Feedback Management (Section 7.2.2)
- Manual Feedback Analysis (Section 7.2.3)
- Automatic Feedback Analysis (Section 7.2.4)
- Feed.UVL Dashboard (Section 7.2.5)
- Jira Plugin (Section 7.2.6)

The **Feedback Collection** category contains multiple web crawlers which can be used through Feed.UVL to gather feedback from the web. This includes a crawler for the Google Play Store[5] where users specify for which app they wish to gather feedback. A crawler for the online forum Reddit[6] that allows users to specify for which specific subreddit (i.e. which specific forum on Reddit) they wish to gather posts and comments. Additionally, the Twitter crawler from Feed.ai is also retained. It should be noted that at the time of writing, changes to both Twitter's and Reddit's API have made crawling these sources more complicated and require a paid membership at the respective website.

The **Feedback Management** category allows users to inspect, delete, change or update all datasets currently handled in Feed.UVL. Next to the above-mentioned crawlers Feed.UVL also allows users to manually upload any datasets.

**Manual Feedback Analysis** is provided through Feed.UVL's annotator which allows users to create annotations for any dataset stored in Feed.UVL. These annotations can be configured to either allow annotation of individual words or entire text segments. Additionally, two annotations of the same dataset can be compared automatically through Feed.UVL's agreement functionality which automatically highlights all disagreements between annotators.

[4] https://www.mongodb.com/
[5] https://www.play.google.com/store
[6] https://www.reddit.com/

**Automatic Feedback Analysis** is provided through the many machine learning classifiers implemented in Feed.UVL. Currently, Feed.UVL provides classifiers for concept identification, data preprocessing, feedback requirements relation, usage information classification, relevance classification and (through a research project not related to this dissertation) acceptance criteria classification. The results of automatic analysis by the classifiers are also visualized through the creation of custom widgets for different types of analysis to best highlight results.

The **Feed.UVL Dashboard** combines multiple functionalities of Feed.UVL into one single dashboard with the purpose of allowing both feedback requirements relation and usage information classification manually and automatically for software projects.

Lastly, the **Jira plugin** connects to the Feed.UVL database in order to provide the results of feedback requirements relation and usage information classification to Jira projects. This allows developers to see feedback related to specific requirements and highlight the contained usage information directly in Jira, where the requirements of the software are documented.

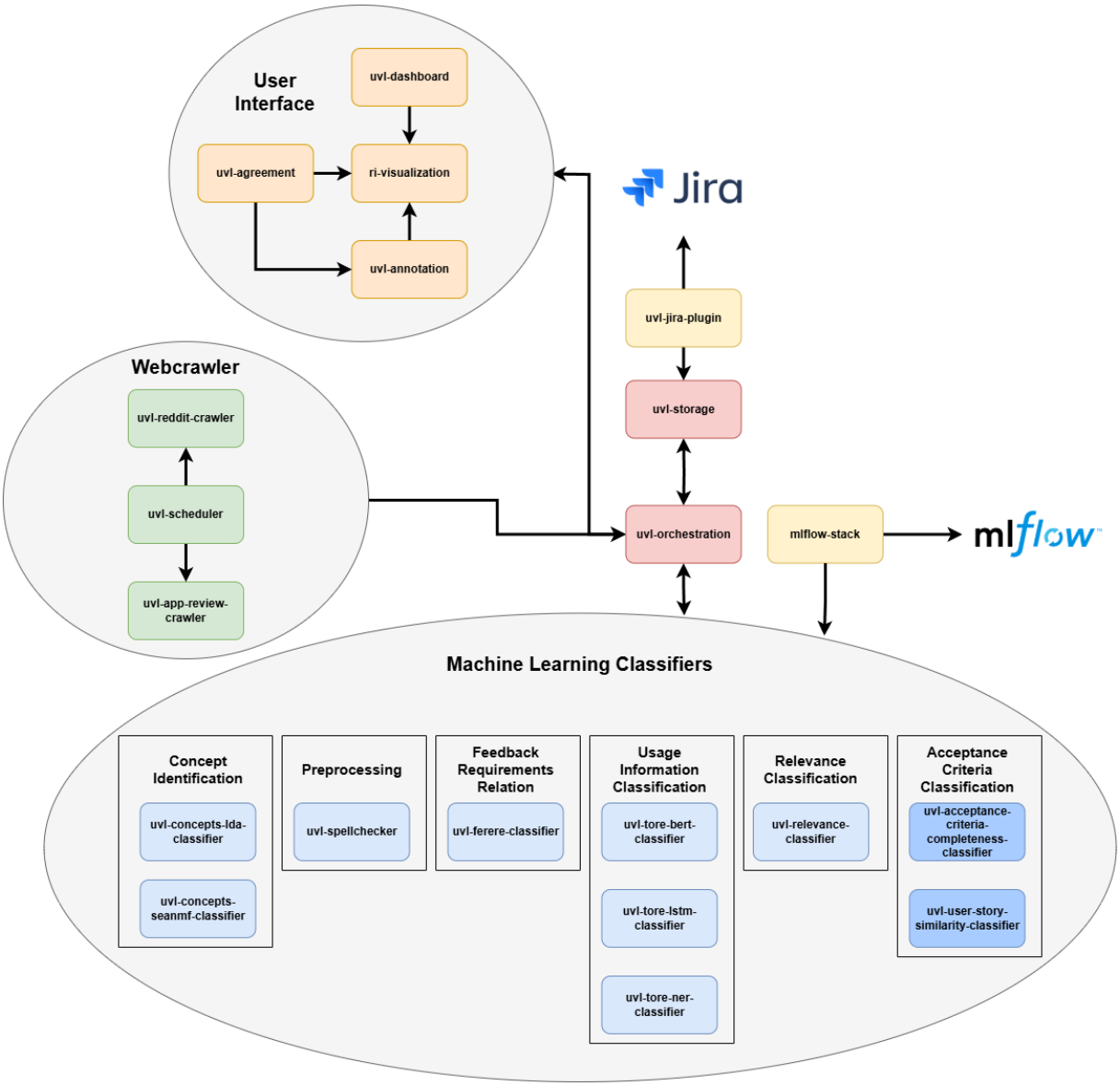



Figure 7.3.: Feed.UVL Services

The individual, colour-coded microservices that make up Feed.UVL can be seen in Figure 7.3. Services coloured in red are core backend services, namely the *uvl-storage* service, which handles the database and the *uvl-orchestration* service, which facilitates communication between all containers.

Services coloured orange are those related to the *UI* of Feed.UVL. The main service here is *ri-visualization*, which provides all views available to users through their browser. Furthermore, *uvl-dashboard* provides the functionalities for the Feed.UVL dashboard, *uvl-annotation* provides the manual annotation functionalities, and *uvl-agreement* the interrater agreement functionalities.

The blue-coloured services are the machine learning classifiers implemented in Feed.UVL. Services are further grouped by the type of analysis they perform. Note that *Acceptance Criteria Classification* services stem from an unrelated research project which also uses Feed.UVL's functionalities and are thus not connected to this dissertation.

Services in green are related to the feedback collection functionalities. These include *uvl-reddit-crawler* and *uvl-app-review-crawler* as well as the *uvl-scheduler*, which allows the other two services to automatically re-crawl their respective websites and update existing datasets in Feed.UVL with new feedback that might have been submitted since the last crawling.

Lastly, yellow-coloured services are external services. The *uvl-jira-plugin* connects to the *uvl-storage* service to access feedback requirements relation and usage information classification results and displays them in Jira. The *mlflow-stack* uses an instance of the machine learning experiment tool MLFlow (see Section 2.4) in order to automatically deploy larger machine learning models to Feed.UVL. This reduces the storage requirements of the classifier containers because models are stored separately.

## 7.2. Feed.UVL Functionalities

This section will explain the requirements and implementation of the individual functionalities of Feed.UVL in more detail. Requirements for Feed.UVL were documented using the TORE Framework (Section 2.2). Due to the sheer number of requirements, not every requirement is fully specified in this dissertation. We omit detailed System Function descriptions such as pre- and post-conditions, as well as individual sub-task steps. These were, however, fully specified in Jira during development and can be found in the digital Appendix A.

Navigation between individual views shown in the following sections is possible through Feed.UVL's navigation bar shown in Figure 7.4. The bar can be opened from any other view by clicking on the top left and will blend into the screen from the left. As can be seen, the bar also contains the initial views provided by Feed.ai on the bottom under *"Twitter-Feed.ai"*. These are not further specified here, and descriptions can be accessed in (Stanik, 2020).

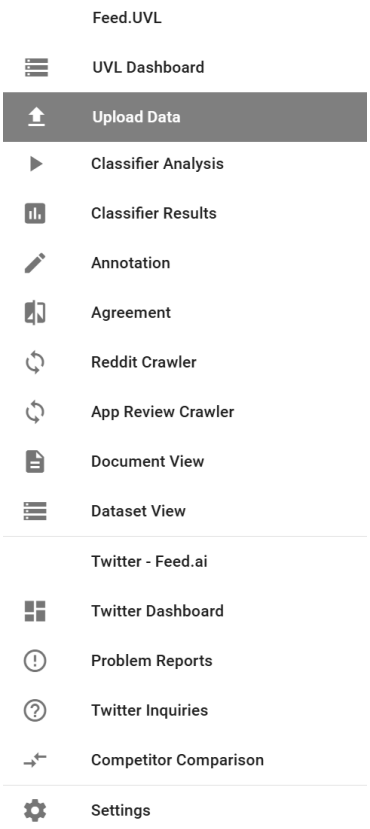



Figure 7.4.: Feed.UVL Navigation Bar

### 7.2.1. Feedback Collection

#### Requirements

This section introduces the requirements of Feed.UVL's functionalities related to the collection of feedback from online sources. These functionalities include two web crawlers for the online forum Reddit and the Google Play store, as well as a scheduling function for automated usage of the web crawlers.

Requirements for the feedback collection functionalities were derived from a need to gather large amounts of feedback, specifically from very common feedback sources as found in literature (see Chapters 3 & 3). We also investigated common filters used by publications using online feedback and derived filter requirements from these.

Table 7.1 lists the user task and related sub-tasks for the feedback collection. The user task captures the user's desire to collect and handle natural language datasets and their creation. The user task is further divided into three sub-tasks. **UT1S1** deals with the creation of new datasets due to predefined criteria by the user. As previously mentioned, the criteria were derived from common filters applied in machine learning publications, such as date ranges in which the feedback was submitted and minimum feedback lengths. The second sub-task **UT1S2** handles the scheduling of reoccurring crawler runs. This stems from a need to automatically gather up-to-date feedback and is can help developers see how feedback changes over time when the software is updated. The third sub-task **UT1S3** manages previous and ongoing dataset aggregations by, for example, changing the schedule to gather data more or less frequently or stopping the gathering process altogether.

Table 7.1.: Feedback Collection User Task & Sub-Task Requirements

| **UT1** | **Collect Natural Language Datasets** |
|---|---|
| The developer performs tasks related to the collection of natural-language feedback datasets. The task consists of the creation of new datasets, the scheduling of the continuous updates of a dataset, and the management of existing dataset collections. This includes an overview of the created datasets and the associated schedule. | |
| **UT1S1** | **Create New Dataset** |
| The developer first specifies details for the creation of the new feedback dataset, such as the feedback source they want to gather feedback from and any filter criteria. The developer then performs the feedback collection, resulting in a new feedback dataset. | |
| **UT1S2** | **Perform a Reoccurring Data Aggregation** |
| The developer repeats the previous gathering of feedback data with the same criteria as before on a fixed schedule in order to capture feedback which has been submitted since the last gathering. | |
| **UT1S3** | **Manage Excecuted Dataset Creations** |
| The developer obtains an overview of all previously created feedback datasets. They see the gathering criteria and schedule of each dataset. They make changes to the schedule of individual datasets. | |

Table 7.2.: Feedback Collection System Function Requirements

| **SF1: Configure Crawler Run** | |
|---|---|
| Description | The user configures all necessary information to run the crawler. This includes the name of the app or subreddit, the data range in which feedback was submitted, the number of feedback posts to be crawled, the minimum length of feedback, as well as comment depth (for Reddit crawler). Further additional filters can be configured: Language of the feedback, removal of URLs and emojis, and list of blacklisted words that exclude feedback. |
| Implements | UT1S1 |
| **SF2: Execute Crawler Run** | |
| Description | The user starts the crawler with the previously configured settings. This option is only enabled if all necessary configurations have been made. |
| Implements | UT1S1 |
| **SF3: Schedule Crawler Run** | |
| Description | The user selects the option for the crawler to update the dataset continuously with newly posted feedback. The schedule continues until SF4 is triggered or the date range specified in SF1 is reached. |
| Implements | UT1S2 |
| **SF4: Stop Scheduling Crawler Run** | |
| Description | The user selects the option to stop the schedule of the crawler run manually. No further updates to the created dataset are made. |
| Implements | UT1S3 |
| **SF5: List Crawler Runs** | |
| Description | The system lists all previous and ongoing crawler runs in an overview. It displays their name, the date of creation and how often they have been rerun. |
| Implements | UT1S3 |
| **SF6: Remove Entry from Overview** | |
| Description | The user removes a dataset from the list of all created datasets. The dataset is not deleted from the database but is not shown in the list anymore. |
| Implements | UT1S3 |

Table 7.2 lists the system functions which implement the previously introduced sub-tasks for the feedback collection in Feed.UVL. Six system functions were derived from the sub-tasks. **SF1** implements the configuration of the crawler, including all filter criteria. Filters are mostly equal between app review and Reddit crawler, but differ slightly because of differences between the app

store and an online forum. The Reddit crawler includes an option to select to which comment depth users wish to collect comments. This did not make sense for the app review crawler, as app stores are not conversational platforms. **SF2** starts the crawler run. This is restricted until all necessary configurations are completed in SF1. **SF3** allows users to schedule a repeated crawler run as described in UT1S2, and **SF4** allows for the cancellation of the crawler run. **SF5** lists every crawler run that has been created in the pas,t along with their names, creation date and how often they have been rerun according to their schedule. **SF6** removes entries from the overview of all crawler runs in case they are no longer relevant to the user.

Table 7.3.: Feedback Collection Workspace Requirements

| **WS1: Reddit Crawler View** | |
|---|---|
| Description | This is the main view of the Reddit crawler. It consists of two smaller workspaces: the configuration view, in which crawler runs are configured, and the overview view, which lists all previous and ongoing crawler runs. |
| Contains | WS1.1, WS1,2 |
| **WS1.1: Reddit Crawler Configuration View** | |
| Description | In this view, all parameters and filters for the Reddit crawler are configured. |
| Contains | SF1, SF2, SF3 |
| **WS1.2: Reddit Crawler Overview View** | |
| Description | In this view, all previous and ongoing Reddit crawler runs are listed. |
| Contains | SF4, SF5 |
| **WS2: App Review Crawler View** | |
| Description | This is the main view of the App review crawler. It consists of two smaller workspaces: the configuration view, in which crawler runs are configured, and the overview view, which lists all previous and ongoing crawler runs. |
| Contains | WS2.1, WS2.2 |
| **WS2.1: App Review Crawler Configuration View** | |
| Description | In this view, all parameters and filters for the app review crawler are configured. |
| Contains | SF1, SF2, SF3 |
| **WS2.2: App Review Crawler Configuration View** | |
| Description | In this view, all previous and ongoing app review crawler runs are listed. |
| Contains | SF4, SF5 |

Table 7.3 lists the workspaces which group the feedback collection system functions. Even though the Reddit and app review crawler are similar in their views, the decision was made to split them into two seperate views. This was mainly done to maintain the microservice architecture of Feed.UVL by allowing complete separation of the two crawlers. They can run independently from one another, and changes to one do not affect the other. As a result the the main views **WS1** and **WS2** are both split into two further sub-workspaces **WS1.1**, **WS1.2** and **WS2.1** and **WS2.2** respectively. The configuration view of both crawlers contains all filters and input fields to configure the crawlers. The overview view contains a list of all previous and ongoing crawler runs created for each crawler.

### Implementation

The app review crawler was implemented using Python and utilizes the google-play-scraper[7] library. This library provides APIs which allow easier data gathering from the Google Play Store. It was mainly chosen because it returns crawled data in the form of JSON objects which matches well with Feed.UVLs MongoDB database. Additionally, API requests using the library

[7] https://www.pypi.org/project/google-play-scraper/

are structured fairly minimalistically while still providing all necessary functionalities, simplifying implementation. Depending on the filter criteria set up by the user, the crawler sends an API request to the Google Play Store, which returns a list of feedback as a response. The library already provides some filters, such as, for example, the feedback language. Other filters, however, had to be manually implemented, such as the filtering of reviews by date.

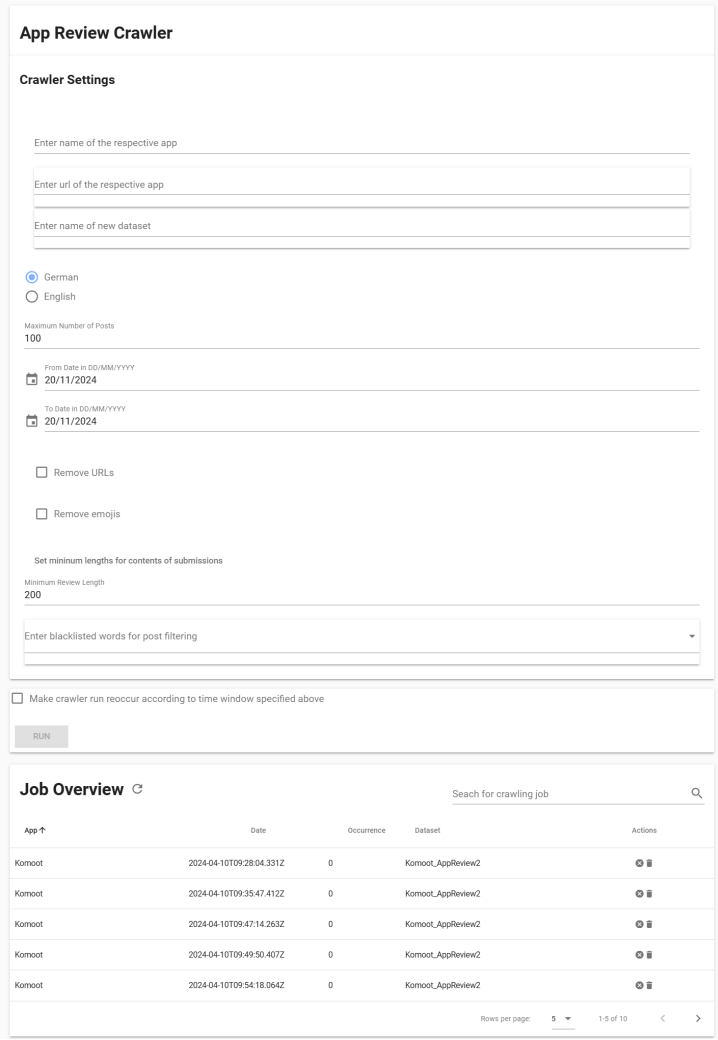



Figure 7.5.: Feed.UVL App Review Crawler

Figure 7.5 shows the implemented app review crawler running in Feed.UVL. The view consists of the "Crawler Settings" section which implements **WS2.1** and "Job Overview" which implements **WS2.2**.

In the crawler settings, the user enters the name of the app, the URL to the Google Play Store page and the name that the created dataset should have. Then, users can configure whether they wish to crawl for German or English feedback. In theory, this could easily be expanded to more languages should the need arise. Following this, users configure the maximum number of feedback that should be gathered, the date range in which the feedback should have been submitted and whether they wish for URLs and emojis in the feedback to be removed automatically. They can also configure the minimum character length that a feedback statement should have to be included. The last configuration is a blacklist which automatically excludes any feedback that contains certain words. This can help with the exclusion of spam bots which utilize these

platforms. After setting all filters, users have the option to make a crawler run reoccurring, at which point it is automatically scheduled to re-crawl at certain intervals.

In the job overview at the bottom, all previous crawler runs are listed. This overview lists the relevant App, when the job was created, how often it has been re-crawled according to its schedule and the name of the created dataset. The buttons on the right of the view allow users to stop any reoccurring schedules and remove the run from the overview.

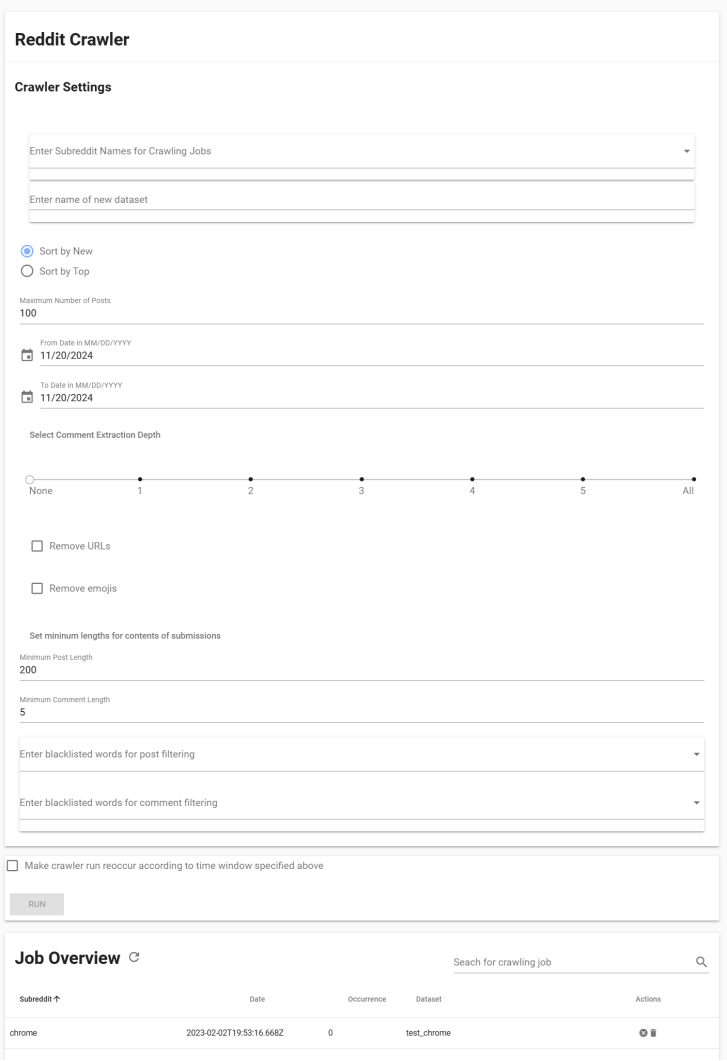



Figure 7.6.: Feed.UVL Reddit Crawler

Figure 7.6 shows the implemented Reddit crawler. As can be seen, the view is almost identical to the app review crawler, with small exceptions in the filters, such as the previously mentioned commend extraction depth. Additionally, a language filter did not make sense for this feedback source, as Reddit is (discounting country-specific forums) almost exclusively English. Instead, the option to *"Sort by New"* and *"Sort by Top"* were included. **Sort by New** gathers posts starting with the newest one in that forum first. **Sort by Top** gathers feedback with the one that has received the most "upvotes" (a form of "Like" showing approval of the post) first. Also, because both the initial forum post as well as its comments are crawled, the minimum feedback filter is extended to both filters by the minimum character length of the original post as well as the minimum character length of comments. If a post does not meet the minimum criteria, it is not included in the dataset. If the post meets the criteria but some of its comments do not, then

these comments are not included, but the post is. The scheduling and job overview sections of the view are identical to those of the app review crawler.

The Reddit crawler was also implemented in Python and uses the Python Reddit API Wrapper (PRAW[8]) to send API requests to Reddit, which then returns the posts and comments. PRAW was chosen because it offered a more complete feature set and more complete documentation compared to other competing libraries such as jReddit[9]. The posts and comments returned by PRAW are then filtered in the crawler's own code. Recent changes to Reddit's API have complicated this process, as Reddit no longer offers a free, unlimited API as it did when the crawler was originally implemented. Consequently, it now requires a registered, paid membership to allow API access, which has to be configured when setting up the Reddit crawlers microservice.

Both the App Review and Reddit Crawler use the same scheduling microservice. The scheduler uses a Flask[10] application and the Advanced Python Scheduler (APScheduler[11]) library to repeat the crawler runs on a fixed timeline. By default scheduled crawlers are run every 24 hours, but this can be modified to any desired time.

### 7.2.2. Feedback Management

#### Requirements

Requirements for the feedback management functionalities were derived as a consequence of the feedback collection requirements listed in the previous Section 7.2.1. These requirements allowed for the creation of new datasets from online feedback. These functionalities resulted in a need for dataset management functionalities to manage the created datasets. Additionally, further functionalities were needed to add datasets which could not be crawled through Reddit or the Google Play Store.

Table 7.4.: Feedback Management User Task & Sub-Task Requirements

| **UT2** | **Manage Natural Language Datasets** |
|---|---|
| The developer needs to manage the collection of all their available datasets. Each dataset contains a number of individual documents. The developer wants to add new datasets to the collection, remove old datasets from the collection, and look at the documents inside each dataset. | |
| **UT2S1** | **Add Dataset to Collection** |
| The developer adds one of their datasets to the collection of all their available datasets. | |
| **UT2S2** | **Remove Dataset from Collection** |
| The developer removes a dataset that is no longer needed from the collection of all their available datasets. | |
| **UT2S3** | **Investigate Datasets** |
| The developer looks at a list of all datasets in their collection and investigates the individual documents inside of a dataset to see the content. | |

Table 7.4 lists the user task and related sub-tasks for the feedback management functionalities. **UT2** deals with the developer's needs to inspect, add and remove datasets. Each dataset consists of individual documents. In an app review dataset for one specific app, for example, each document would be a single app review on that app's store page. This user task is further refined by 3 sub-tasks. **UT2S1** concerns adding datasets to the collection of available datasets. This is different from UT1S1 because instead of creating new datasets, the datasets added in UT2S1 already exist and are merely added to a collection of other datasets. **UT2S2** deals with the

[8]https://www.praw.readthedocs.io/en/stable/
[9]https://www.github.com/jReddit/jReddit
[10]https://www.flask.palletsprojects.com/en/stable/
[11]https://www.apscheduler.readthedocs.io/en/3.x/

removal of datasets that are already in the collection. This might be necessary because the datasets are no longer relevant to the developer, or they might be outdated and need to be replaced with newer data. **UT2S3** captures the need to investigate the content of datasets. As each dataset is made up of individual documents, developers want to see these documents.

Table 7.5.: Feedback Management System Function Requirements

| **SF6: List All Datasets** | |
|---|---|
| Description | The system shows a list of all datasets in the database. |
| Implements | UT2S3 |
| **SF7: Upload Dataset** | |
| Description | The user uploads a dataset to the database that they have stored locally on their machine. The dataset is formatted as a xlsx or csv file with the first column being the content of the dataset and the second column being a unique ID for each row. Each row represents a document in the dataset. |
| Implements | UT2S1 |
| **SF8: Delete Dataset** | |
| Description | The user removes a dataset from the database. The dataset is no longer shown in the list of all datasets. When deleting, the user is given the option to also delete all associated materials, such as Annotation and Classifier runs related to this dataset. |
| Implements | UT2S2 |
| **SF9: Show Dataset Content** | |
| Description | The system shows the list of documents contained in a specific dataset. Both the text and the ID of each document are displayed. |
| Implements | UT2S3 |

Table 7.5 shows the system functions which derive from the sub-task related to UT2. **SF6** displays a list of all datasets that are currently stored in the Feed.UVL database. The list contains all datasets created through *SF2: Execute Crawler Run* and **SF7**. SF7 allows for the manual upload of datasets that are stored on the user's device. These need to be formatted properly. The user is prompted how to format the files they wish to upload in the software. Entries can be deleted from the list and Feed.UVL's database through **SF8**. Lastly, **SF9** allows the user to see the documents within a specific dataset.

Table 7.6.: Feedback Management Workspace Requirements

| **WS3: Upload View** | |
|---|---|
| Description | In this view, the user can upload datasets from their machine to the system. A list of all currently available datasets is shown. Users can remove datasets here or navigate to WS4: Dataset View to see the content of a dataset. |
| Contains | SF6, SF7, SF8, SF29 |
| **WS4: Dataset View** | |
| Description | In this vie,w a list of the documents in a specific dataset is shown along with the ID of the document. Users can look at different datasets and delete datasets from the database. |
| Contains | SF8, SF9 |

Table 7.6 lists the workspaces which group the previously introduced system functions. **WS3** handles the upload of datasets, removal of datasets and listing of all currently available datasets in Feed.UVL. **WS4** displays the content of each dataset. Both the text as well as the ID of each document in a dataset is displayed. This view also allows users to delete the currently displayed dataset as well as select any other dataset for display.

### Implementation

As all parts of the frontend, both the upload view and dataset view are written in VueJS, while the backend services for these functionalities are written in the programming language GO. These two languages were dictated by the original setup of Feed.ai from which Feed.UVL was derived (Section 7.1.1). The feedback management functionalities represent the core functions of Feed.UVL as they are implemented in the *uvl-orchestration* and *ri-visualization* services (Section 7.1.2). Unlike most other functionalities of Feed.UVL these are not optional features because all other functionalities of the software rely on the ability to manage feedback datasets in Feed.UVL.

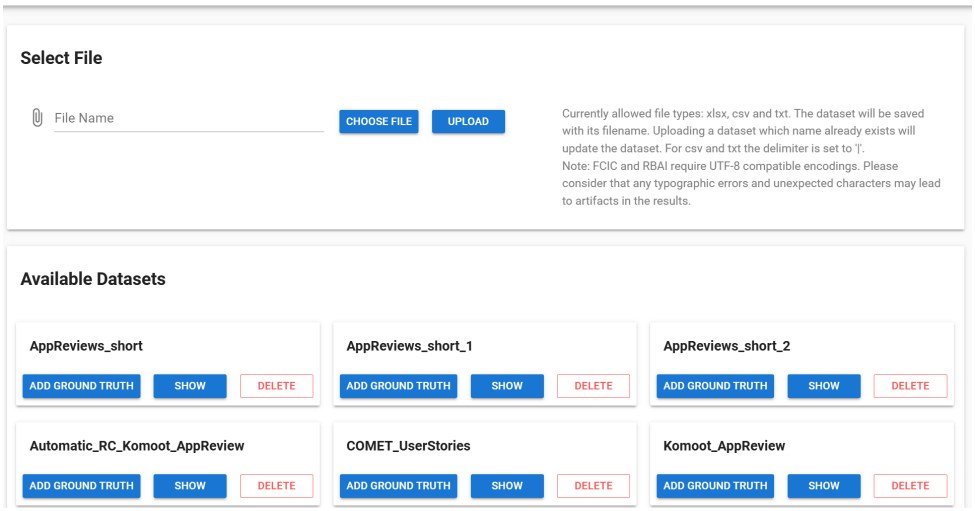



Figure 7.7.: Feed.UVL Upload View

Figure 7.7 shows the implemented upload view defined by WS3. At the top, the upload functionalities are shown. The user is given the option to choose a file from their device and then upload it to the software. To the right, a hint is shown that explains the required format of the upload. The system will display a warning message that the upload has failed, should an upload not meet these requirements. Below the upload the list of all available datasets is shown. Each dataset is displayed in a box with its name, the option to delete the dataset, the option to show the content of the dataset and the option to add a ground truth. Pressing delete will prompt the user to confirm the deletion and allows them to also select an option to delete all information related to that dataset, such as annotations and classifier runs. Pressing the "Show" button navigates the user to the dataset view. The "Add Ground Truth" button is related to the Automatic Feedback Analysis functionalities introduced in Section 7.2.4.

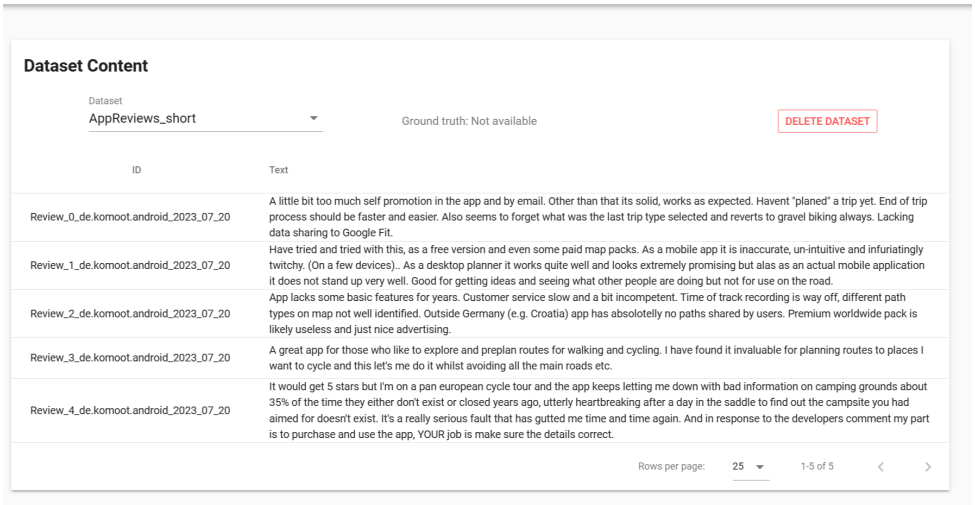



Figure 7.8.: Feed.UVL Dataset View

Figure 7.8 shows the implemented dataset view defined by WS4. Dataset documents are displayed as individual entries in a table, with the ID of a document on the left and the text content on the right. The top left shows a selection option which, when clicked, lists all available datasets and switches to the selected option. The top right shows a delete button for said dataset. The same confirmation dialogue is shown when clicking the delete button in the upload view. The bottom right of the screenshot shows that pagination is used to display datasets. This reduces load times for very large datasets and makes navigation and readability easier.

### 7.2.3. Manual Feedback Analysis

#### Requirements

The requirements for the manual feedback analysis were derived from the functionalities required to support manual structured analysis as introduced in Fundamentals Section 2.1. This mainly focused on the necessity to assign predefined categories to individual segments of the feedback. This motivated the creation of the Feed.UVL Annotator is an annotation tool specifically targeted for word-level and sentence-level coding of feedback texts. Specific requirements were created as part of a research project in which the author of this dissertation participated. As part of the project, usage information was manually classified using the TORE framework (Section 2.2). Existing annotation tools (namely MAXQDA and QDAMiner) that were used by the participants of the research project did not provide adequate functionalities to quickly perform the usage information classification. Consequently, requirements were derived based on the needs of the participants of the research project with the main focus of allowing easy-to-use usage information classification.

It should be noted that while usage information classification was the main use case targeted by the Annotator it is still applicable to any other annotation task, requirement word-level or sentence-level annotation.

Table 7.7 lists the user task and related sub-tasks for the manual feedback analysis. The main desire of the developer performing structured manual feedback analysis is to assign codes to a feedback dataset and then analysing which codes were assigned. An example use case of this could be a developer assigning sentiment codes (positive, negative, neutral) to feedback sentences and then seeing if the sentences were predominantly positive or negative to judge users'

Table 7.7.: Manual Feedback Analysis User Task & Sub-Task Requirements

| **UT3** | **Manage Manual Analyses** |
|---|---|
| The developer wants to manage the manual analysis of feedback. For this, they need to perform the analysis on feedback data by assigning codes to the feedback. Afterwards, they wish to see the results of the analysis by inspecting the codes they have assigned. In some cases, multiple developers may wish to see if their analyses of the feedback match and if they are in agreement with the analyses. Also, developers may wish to discard old analyses because they are no longer relevant. | |
| **UT3S1** | **Perform Manual Analysis** |
| The developer performs the analysis of feedback by choosing a dataset they wish to analyse, defining the codes they wish to assign to that feedback and then annotating the feedback with the chosen codes. | |
| **UT3S2** | **Inspect Manual Analysis** |
| The developer inspects the analysis once it is complete to see which codes they assigned and calculate statistics about these codes, such as the number of occurrences of each code. | |
| **UT3S3** | **Create Agreement Between Analyses** |
| Two developers want to find out if they are in agreement about their analysis of the feedback. For this they see if they have assigned the same codes to the same segments of feedback. They discuss any differences and resolve them so they are in agreement. | |
| **UT3S4** | **Discard Previous Manual Analyses** |
| The developer wants to discard old analyses that are no longer relevant. | |

sentiment towards their software. The user task is divided into four sub-tasks. **UT3S1** handles the manual coding itself. For this, developers choose a dataset and the codes they wish to use for the analysis. They then assign these codes to segments inside the dataset. **UT3S2** handles the inspection of the coding results. Here developers investigate which codes were assigned to which segments in which frequency. **UT3S3** stems from a need for developers to create an agreement between them and their colleagues. As analysis can be a subjective task, developers might have to come to an agreement about what the results of the analysis are. To create the agreement, they need to find out where developers disagree in their analysis and discuss these differences. **UT3S4** deals with the deletion of old analyses. Developers may wish to discard them because they are out-of-date or no longer relevant.

Table 7.8 lists the system functions that derive from the sub-tasks for the manual feedback analysis. In total, 18 system functions were created to provide adequate tool support for the manual feedback analysis. These 18 system functions can be separated into annotation and agreement functionalities. Annotation functionalities handle the UT3S1, UT3S2 and UT3S4, while agreement functionalities mainly implement UT3S3.

**SF10** - **SF16** and **SF20** provide functionalities for the manual coding itself. Users may create, configure, load, save and delete annotations of natural language datasets stored in Feed.UVL. The system automatically tokenizes the datasets according to whether individual words or entire sentences in the feedback should be assigned codes. **SF15** handles the code assignment. Feed.UVL annotations support three different types of code. The first type are "word codes", which are a free-text field in which users can enter any text they wish, which is then assigned to the segment. The second type are "category codes" which users can define in SF11. These are a predefined list of categories which can be assigned to segments. TORE categories (Section 2.2), for example, can be used as "category codes". The advantage of category codes is that they are simpler to select from a given list and thus easier to assign than "word codes". The third type of code are "relationship codes". These codes define a relationship between two segments and their category codes. For example, these codes allow users to capture that one segment of the feedback "affects" another by creating an "affects" relationship code between the segments. Which relationships exist can also be defined by users using SF11. **SF20** allows users to highlight different segments in the annotation. They may highlight classification results of classifiers in

Feed.UVL (see Section 7.2.4), or part-of-speech tags such as nouns, verbs and adjectives. They may also highlight a custom selection of category codes in different colours to quickly see which segments have been assigned which codes.

**SF17** and **SF18** deal with the creation, display and export of statistics of an annotation. Tables are provided that list the occurrences of each type of code. Here, developers can see how often which code was assigned to which text segment. This allows them to inspect the results of the analysis and draw conclusions from it.

**SF19** defines the recommendation function of the Feed.UVL annotator. This requirement came from a research project performing usage information classification on a word-level. Researchers noticed that the same words were often assigned the same usage information categories. To speed up the annotation process, the recommendation function was created. The function analyses all annotations which exist in Feed.UVL and stores the most common category code assigned to a word. If the word is then annotated in a new annotation, the recommendation function automatically pre-selects the most used category code. This pre-selection does not influence the amount of clicks users need to make to assign a different code to the word, but it speeds up the selection process if the pre-selected code is correct.

**SF21** - **SF27** handle the creation and deletion of agreements between annotations. Users may create an agreement between any two annotations of the same dataset. The system then automatically calculates the disagreements, i.e. where codes between the two annotations do not match, as well as Kappa values about the agreement between the annotations. These requirements again came from the research project performing usage information classification when it became necessary for researchers to create an interrater agreement for their dataset to guarantee a certain quality of data.

Table 7.9 lists the workspaces which group the previously introduced system functions for the manual feedback analysis. These views can be separated into those related to annotations and those related to agreements. Annotation and agreement views were intentionally kept seperate despite heavy reuse of annotation functionalities for the agreements in order to maintain Feed.UVL's microservice architecture. Annotation functionalities may be used completely independently from agreement functionalities as users may not have a need for the creation of agreements.

**WS5** contains the main view in which users create, delete and load annotations. This includes a list of all existing annotations from which previous annotations may be loaded. **WS6** contains the statistics of individual annotations. Here, tables are provided that list the occurrences of each code and allow navigation to specific segments of the annotation. **WS7** is the main annotation view. Here, users can assign new codes to segments by clicking on them, which opens the sub-workspace **7.1** in which the different types of codes can be assigned. **WS8** is the configuration view for annotations, here the category and relationship codes that should be available in an anotation can be configured. The view also allows for the enabling and disabling of the recommendation functionality (SF19). **WS9** - **WS11.1** concern the agreement functionalities. These workspaces are structured the same way as the annotation workspaces.

Table 7.8.: Manual Feedback Analysis System Function Requirements

| **SF10: Create Annotation** | |
|---|---|
| Description | The user creates a new annotation by selecting a dataset for which to create the annotation. The user can select whether the annotation should assign codes to individual words or entire sentences. |
| Implements | UT3S1 |
| **SF11: Configure Annotation** | |
| Description | The user decides which predefined codes should be assignable inside of an annotation. They can add new codes or delete existing ones. |
| Implements | UT3S1 |
| **SF12: Load Annotation** | |
| Description | The user loads a previously created annotation in order to continue working on it. |
| Implements | UT3S1 |
| **SF13: Save Annotation** | |
| Description | The user saves the annotation they are currently working on. The system also automatically saves an annotation every 5 minutes while it is being worked on to prevent information loss. |
| Implements | UT3S1 |
| **SF14: Delete Annotation** | |
| Description | The user deletes an annotation from the system. |
| Implements | UT3S4 |
| **SF15: Assign Code** | |
| Description | The user assigns a code to a segment. This segment is either an individual word or an entire sentence, depending on the type of Annotation created in SF10. There are three types of codes: (1) Word codes, which allow users to type any text they wish and assign it as a code, (2) category codes, which are predefined codes as configured in SF11, and (3) relationship codes which describe a relationship between two category coded segments. Users can freely choose which types of codes they wish to use. |
| Implements | UT3S1 |
| **SF16: Update Code** | |
| Description | The user changes a code that was previously assigned to a segment. The segment now shows as being coded with the new code and not the old. |
| Implements | UT3S1 |
| **SF17: List Annotation Statistics** | |
| Description | The system lists statistics about the current annotation. This includes occurences of all three types of codes as well as how often they appear in combination. Category code occurrences can also be displayed in the form of a heat map. |
| Implements | UT3S2 |
| **SF18: Export Annotation Statistics** | |
| Description | The user downloads the statistics tables to use them outside of the system. |
| Implements | UT3S2 |
| **SF19: Show Annotation Recommendations** | |
| Description | The system displays recommended category codes to the user when assigning codes to a word. The recommendations are based on the frequency of a code being assigned to the relevant word in all other existing Annotations. |
| Implements | UT3S1 |
| **SF20: Highlight Annotation Segments** | |
| Description | The user can highlight individual segments of the annotation. There are three types of highlights: (1) Highlighting the classification of Machine Learning Classifiers for the Annotation, (2) Highlighting the Part of Speech Tag of individual words, and (3) Highlighting the different category codes already coded in the annotation. |
| Implements | UT3S2 |
| **SF21: Create Agreement** | |
| Description | The user creates a new agreement between two annotations of the same dataset. |
| Implements | UT3S3 |
| **SF22: Load Agreement** | |
| Description | The user loads a previously created agreement in order to continue working on it. |
| Implements | UT3S3 |
| **SF23: Resolve Disagreement** | |
| Description | The user resolves disagreements by clicking on a disagreement between the annotations and either selecting one of the two annotations' codes, rejecting both or creating a new code. |
| Implements | UT3S3 |
| **SF24: List Agreement Statistics** | |
| Description | The system lists all unresolved and resolved disagreements between the two annotations as well as Brennan&Prediger and Fleiss Kappa. |
| Implements | UT3S3 |
| **SF25: Export Agreement Statistics** | |
| Description | The user downloads the statistics tables to use them outside of the system. |
| Implements | UT3S3 |
| **SF26: Save Agreement** | |
| Description | The user saves the agreement they are currently working on. The system also automatically saves an agreement every 5 minutes while it is being worked on to prevent information loss. |
| Implements | UT3S3 |
| **SF27: Delete Agreement** | |
| Description | The user deletes an agreement from the system. |
| Implements | UT3S4 |

Table 7.9.: Manual Feedback Analysis Workspace Requirements

| **WS5: Annotation Master View** | |
|---|---|
| Description | In this view, users can create new annotations, as well as select whether they wish to create a word-level or sentence-level annotation. The view also lists all existing annotations. These can either be edited or deleted. |
| Contains | SF10, SF12, SF14 |
| **WS6: Annotation Statistics View** | |
| Description | In this view, different statistics about the currently loaded annotation are shown. Users can switch between different tables showing the occurrences of each type of code (defined in SF15). The statistics of category codes can also be shown as a heat map. Users can also navigate to a specific code in WS7. |
| Contains | SF13, SF17, SF18 |
| **WS7: Annotation Edit View** | |
| Description | In this view the content of the annotation is displayed. Users can switch between seeing all or individual documents of the dataset. This view also provides the highlighting functions laid out in SF20. Clicking on a word or sentence (depending on annotation type) navigates to WS7.1. |
| Contains | SF13, SF20 |
| **WS7.1: Annotator Code Input View** | |
| Description | In this view, users can assign new codes to segments they have previously clicked on. They may also update codes already assigned to the segment. |
| Contains | SF15, SF16, SF19 |
| **WS8: Annotation Configuration View** | |
| Description | In this view category and relationships code for the annotation can be defined. Users can delete existing code categories or add new ones. This view also allows users to enable or disable recommendations (SF19). |
| Contains | SF11 |
| **WS9: Agreement Master View** | |
| Description | In this view, users can create new agreements between two annotations of the same dataset. The view also lists all existing agreements. These can either be edited or deleted. |
| Contains | SF21, SF22, SF27 |
| **WS10: Agreement Statistics View** | |
| Description | In this view, all unresolved and resolved disagreements are listed. Users can navigate to the disagreements from here to WS11. The view also shows the Brennan&Prediger and Fleiss Kappa between the two annotations. |
| Contains | SF24, SF25, SF26 |
| **WS11: Agreement Edit View** | |
| Description | In this view, the content of the agreement is displayed. Users can switch between seeing all or individual documents of the dataset. Agreements and disagreements between annotations are highlighted. Clicking on an agreement or disagreement navigates to WS11.1. |
| Contains | SF26 |
| **WS11.1: Agreement Code Input View** | |
| Description | In this view, users can resolve disagreements by rejecting the codes of one or multiple annotations. They may also create new codes similar to WS7.1. |
| Contains | SF23 |

### Implementation

The backend part of the annotations functionalities was implemented in Python because of its excellent support for natural language processing libraries. The main library used by the annotation functionalities is NLTK[12] to perform the tokenization of the datasets into either sentences or individual words. These tokens are then represented on the UI as clickable elements, which allow for code assignments. Additionally, a WordNet[13] API is used to perform the lemmatization of each token. During lemmatization, a word is returned to its root. This improves the performance of the recommendation function as codes can be stored for lemmatized words.

The backend of the agreement functionalities was written in GO. This choice was made as most of the functionalities of the annotation service could be reused and thus did not require a new Python implementation. Consequently, GO was favourable as it allowed for easier communication with the uvl-orchestration microservice, which was also written in GO. Reusing the annotation functionalities was not a concern for the microservice architecture, as using the agreement functionalities without annotations would not be feasible anyway.

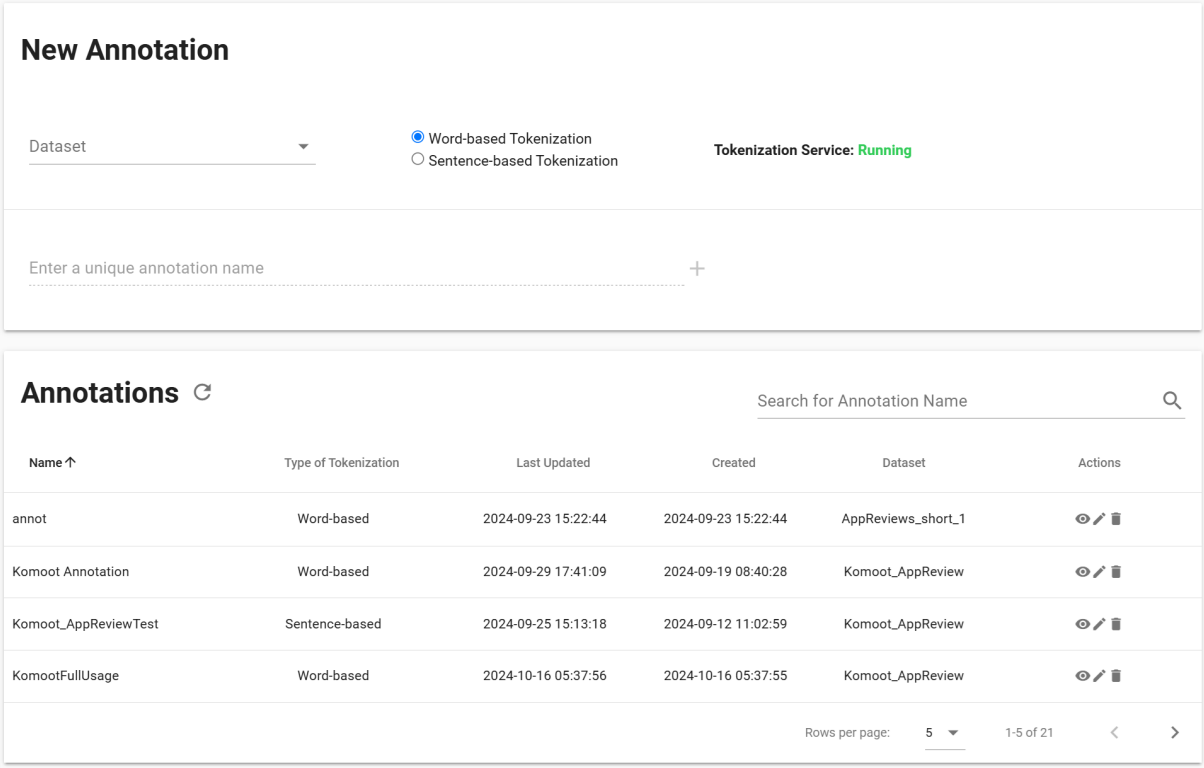



Figure 7.9.: Feed.UVL Annotation Master View

Figure 7.9 shows the implemented Annotation Master view. At the top, users may select the dataset they wish to annotate and whether they wish to perform word-level or sentence-level annotation. This choice is then reflected in the Annotation Edit View (Figure 7.13) by having either individual words, which can be clicked and assigned code to or entire sentences. Users may also give the annotation a custom name. Below the annotation creation, all available annotations are listed along with their names, type of tokenization (word or sentence), when they were last updated, when they were created and on which dataset the annotation was created. The list may also be sorted by any of these criteria. Additionally, a search filter is provided to search for

[12]https://www.nltk.org/
[13]https://www.wordnet.princeton.edu/

specific annotation names. To the right, users may either navigate to the Annotation Statistics view, the Annotation Edit View or delete the annotation from the database.

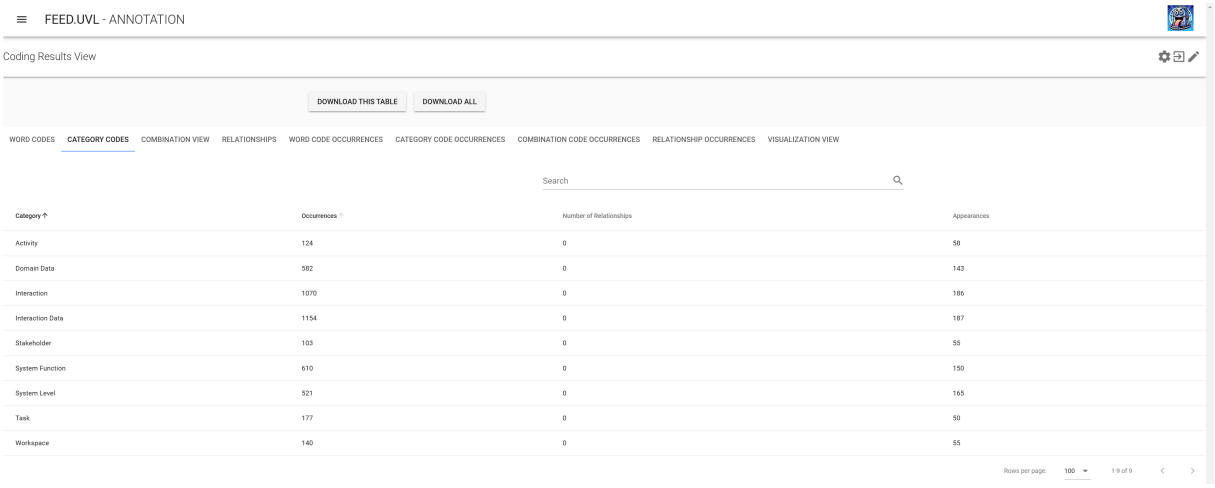



Figure 7.10.: Feed.UVL Annotation Statistics View

Figure 7.10 shows the Annotation Statistics View for an example annotation. A set of eight different tables is provided, which can be navigated by clicking on the respective tab. The "Category Codes" table shown here lists the category codes present in the annotation, how often they occur in total ("Occurences"), the number of relationship codes they are part of, i.e. how many segments these category codes are assigned to, are also part of a relationship code, and the number of appearances they have, i.e. in how many documents of the dataset the code appears at least once. Similar tables are available for word codes, relationship codes and the combination of all three ("Combination View"). Additionally, a list of all occurences of each code can be seen in the "Occurences" tables. These tables help users gain an overview of entire feedback datasets. They simplify analysing usage information on a meta-level, i.e. investigating which usage information users are mostly discussing in their feedback or searching for common themes in the dataset by filtering for specific usage information and related terms.

Lastly, as can be seen in Figure 7.11, the category codes and relationship codes may also be visualized as a heatmap by clicking on the "Visualization View" tab. This view uses BaklavaJS[14] to automatically create a graph representation of the codes. Codes are visualised as boxes and coloured according to their occurrences or appearances (i.e. how often they are assigned in total or in how many documents they are assigned at least once). Relationships between category codes are represented through connections between boxes. This functionality is intended to provide an easy visualisation of how often certain codes are used in the annotation. Elements can be moved around freely to reorganize the graph.

Figure 7.12 shows the configuration view of the annotator. Here, users can define the category and relationship codes that should be available in the annotation. This can be changed at any time. Users may change or delete existing codes or add new ones. Additionally, the category code recommendation function may also be disabled or enabled here.

Figure7.13 shows an example Annotation Edit View for a word-level annotation. At the top left, users may select whether they wish to see all documents or which individual documents below. When selecting "All Documents", as can be seen at the top, pagination is used to maintain a reasonable UI scale. To the right of the pagination options, highlighting for classifier results, part of speech tags of category codes are shown, along with buttons to open the configuration view, statistics view or save and exit the annotation. The main part of the view is the tokenized

[14]https://www.github.com/newcat/baklavajs

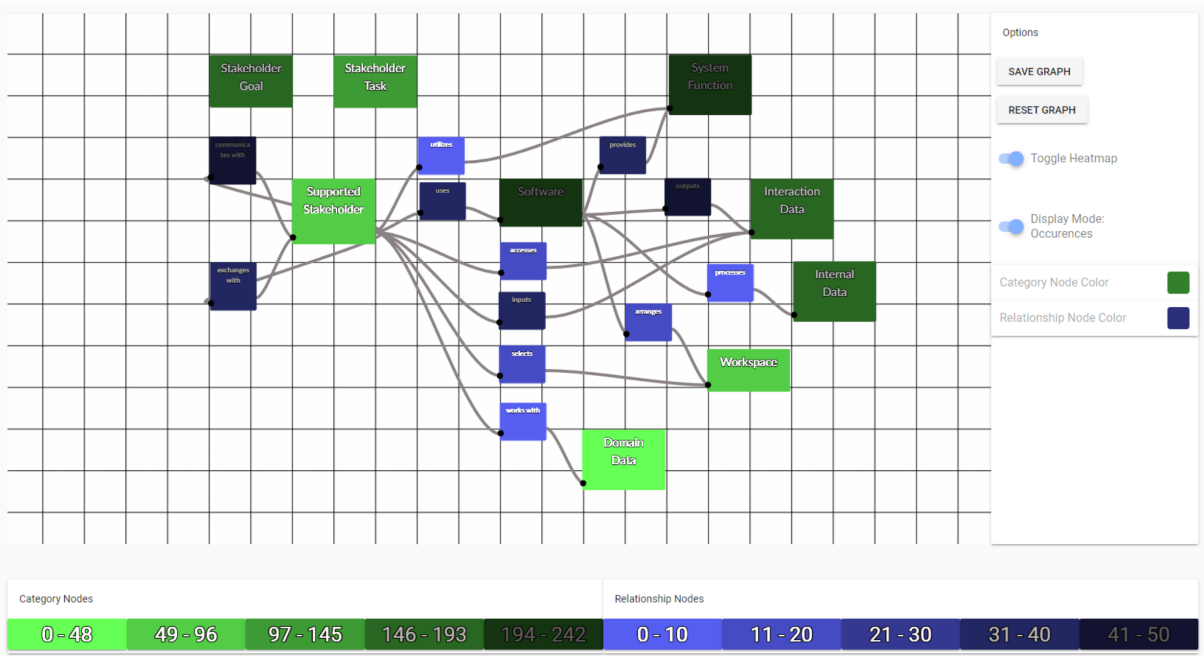



Figure 7.11.: Feed.UVL Annotation Statistics Visualization View

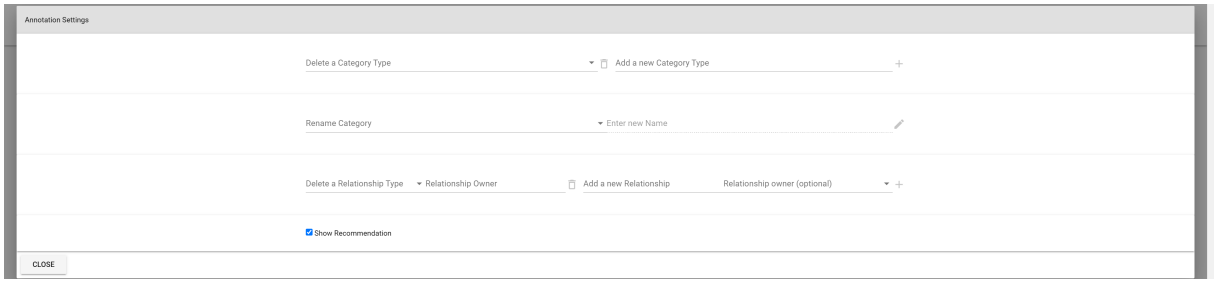



Figure 7.12.: Feed.UVL Annotation Configuration View

texts that can be seen below. By default, words that have been assigned codes are coloured according to the type of code that has been assigned to them. In the shown example, category codes have been assigned to words, resulting in a blue bounding box around the words.

Clicking on any word opens up the Code Input View at the clicked word. This can be seen at the bottom of the screenshot for the word "information". The input has several options, which are reported here from left to right. Clicking on the three bars on the left allows users to highlight all similar words in the same document and assign the same code to all. Clicking the trash can icon allows users to delete the assigned codes from that word. The "Name" field allows users to assign a word code. By default, the lemmatized word is written here, but this can be replaced with any text the user wishes. Clicking the category field lists all available category codes in that annotation. If the recommendation function is enabled, the category field is automatically pre-selected with the most common category code assigned to the clicked word. Clicking on the "chain" icon to the right of the category field allows users to assign relationship codes. An example of this can be seen in Figure 7.14. Here, users select the words they wish to relate to and select a relationship code before confirming their selection. Multiple words can be selected by holding this Shift key. The remaining two buttons on the right of the input view, move the view further down in case it is blocking relevant text (arrow icon) and bring up the configuration

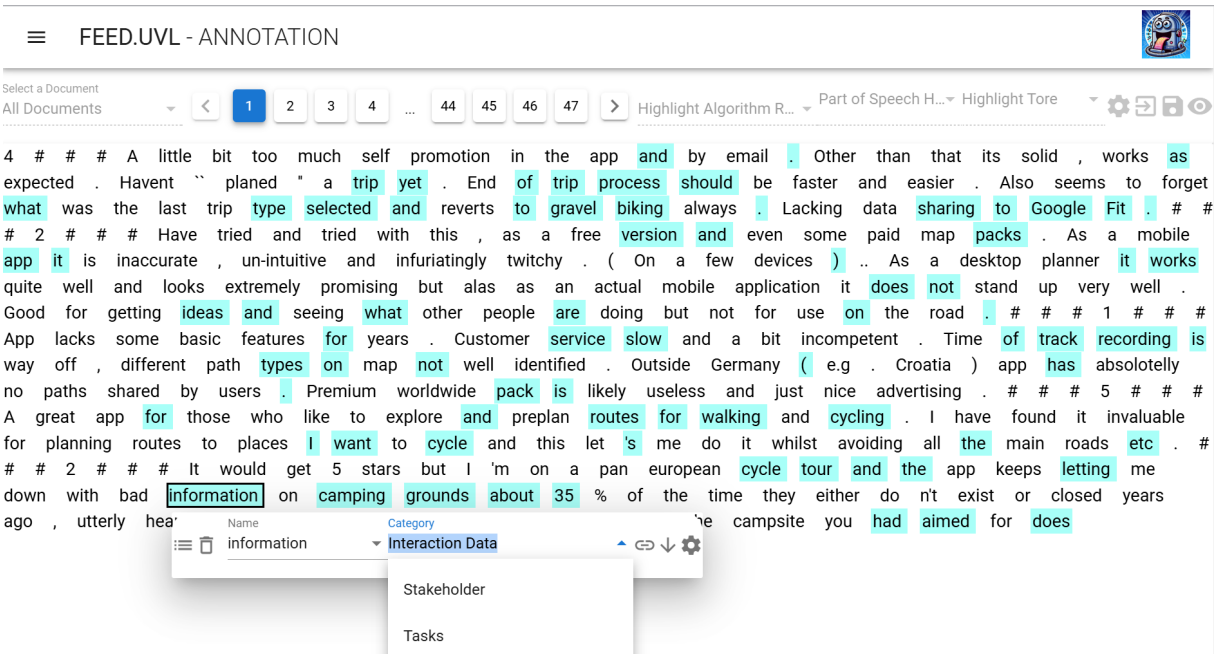



Figure 7.13.: Feed.UVL Annotation Edit & Code Input View

view (gear icon). All functionalities mentioned here are also explained with tooltips by hovering over the icon.

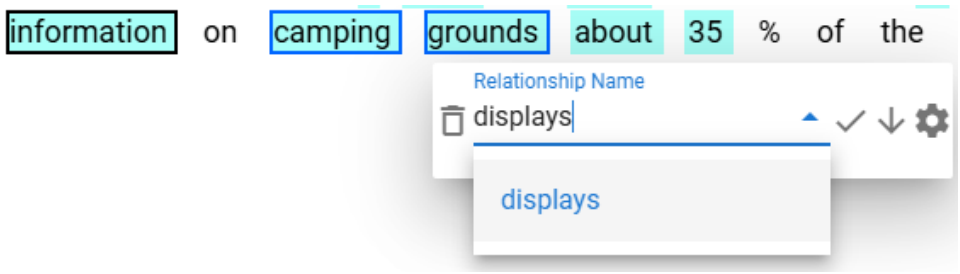



Figure 7.14.: Feed.UVL Code Input View Relationship

We have omitted further screenshots of the agreement views, as they reuse the same UI structure as the Annotation Master, Statistics and Edit Views. Screenshots are, however, available in the digital Appendix A.

### 7.2.4. Automatic Feedback Analysis

#### Requirements

The requirements for the automatic feedback analysis functionalities were derived to enable developers to efficiently analyze feedback using machine learning techniques by complementing the manual feedback analysis processes. The automation focuses on deploying classifiers, analyzing feedback datasets, and managing analysis results. These requirements emerged because of the time and labour-intensive process of manually analysing feedback.

Table 7.10 defines the primary user task, UT4, which is to manage automatic analyses. Developers aim to perform automated analysis of feedback data using machine learning classifiers, inspect the results, and discard outdated analyses when necessary. This task is divided into three sub-tasks. UT4S1 covers performing automatic analysis, where developers select a dataset, define

Table 7.10.: Automatic Feedback Analysis User Task & Sub-Task Requirements

| **UT4** | **Manage Automatic Analyses** |
|---|---|
| The developer wants to manage the automatic analysis of feedback. For this, they need to perform the analysis on feedback data by using a machine learning classifier to perform the analysis. Afterwards, they wish to see the results of the analysis by inspecting the classifier results. This may include inspecting the performance of the classifier. Also, developers may wish to discard old analyses because they are no longer relevant. | |
| **UT4S1** | **Perform Automatic Analysis** |
| The developer performs the automatic analysis of feedback by choosing a dataset they wish to analyse, defining the machine learning classifier that performs the analysis and setting its parameters. | |
| **UT4S2** | **Inspect Automatic Analysis** |
| The developer inspects the analysis once it is complete to see the classifier's classification. They may also want to analyse the classifier's performance by calculating performance metrics. They may also wish to make changes to the classifier's classification. | |
| **UT4S3** | **Discard Previous Automatic Analyses** |
| The developer wants to discard old analyses that are no longer relevant. | |

a classifier, and configure its parameters to execute the analysis. UT4S2 focuses on inspecting the analysis results. This involves examining the classifier's output, evaluating its performance through metrics such as precision, recall, and F1 scores, and making manual adjustments to the classification if needed. UT4S3 addresses the need to discard previous analyses that are no longer relevant, ensuring that the system remains organized and current. The user task and sub-task are based on the manual analysis tasks.

Table 7.11 elaborates on the system functions derived from the previously introduced sub-tasks. SF28 allows users to deploy machine learning classifiers either by directly uploading a trained model to the system or by using the MLFlow stack (Figure 7.3) to automatically load the model. This reduces the size of the classifiers' microservice containers considerably when working with larger models. SF29 allows users to upload a ground truth in the Upload View, which can then be used during SF32 to calculate performance metrics. SF30 and SF31 enable users to configure an analysis run by selecting a classifier, dataset, and associated parameters and to initiate the analysis process. Once the analysis is executed, SF32 generates the run results, including calculated performance metrics if a ground truth dataset is available.

For UT4S2, system functions focus on result inspection and modification. SF33 lists all stored run results, which can be sorted based on various attributes such as date, classifier, or performance score. SF34 enables filtering of these results by specific criteria like classifier type or dataset. Once a result is identified, SF35 allows users to view detailed outputs by selecting the "Show Result" option. Additionally, SF36 supports exporting run results into annotations, integrating the classification outcomes into manual annotation processes for further inspection, modification, or use in agreements.

UT4S3 is implemented through SF37, which allows users to delete run results that are no longer relevant. This ensures the system remains uncluttered and focused on current analyses, reflecting the developers' need to manage their workspace efficiently.

Table 7.12 outlines the workspaces that group the system functions. WS12, the Classifier Analysis View, contains functionalities for configuring and executing analysis runs (SF30, SF31) and for managing the results list (SF33, SF34). WS13, the Classifier Results View, focuses on displaying individual run results through SF35. Which widgets are displayed in this view to show the results depends on each classifier is defined when deploying the classifier to Feed.UVL.

Table 7.11.: Automatic Feedback Analysis System Function Requirements

| **SF28: Deploy Machine Learning Classifier** | |
|---|---|
| Description | The user deploys a machine learning model into Feed.UVL for later use in feedback analysis. The user may either provide the finished model directly in the classifiers microservice or they may upload the training run of the classifier to the MLFlow-Stack which then automatically loads the model into Feed.UVL once the microservice is running. |
| Implements | UT4S1 |
| **SF29: Upload Ground Truth** | |
| Description | The user uploads a ground truth for a dataset. The ground truth consists of an xlsx or csv file with the ID of the document in the first column and the assigned classes in the second column. This ground truth can then be seen in WS4: Dataset View and is used to calculate metrics in SF32. |
| Implements | UT4S2 |
| **SF30: Configure Analysis Run** | |
| Description | The user selects the classifier they wish to use and the dataset they wish to perform the classification on. Custom fields are displayed depending on the classifier, which allows the user to customize the classifier's parameters for the run. |
| Implements | UT4S1 |
| **SF31: Start Analysis Run** | |
| Description | The user starts the analysis run with the classifier, dataset and parameters defined in SF30. |
| Implements | UT4S1 |
| **SF32: Create Run Results** | |
| Description | The system creates the run results by running the selected classifier on the dataset. If a ground truth is present for the dataset, the system calculates precision, recall and F1 metrics. |
| Implements | UT4S1 |
| **SF33: List All Run Results** | |
| Description | The system lists all run results stored in its database. The list can be sorted by date, classifier, dataset, parameters, run name, status or score (optional). |
| Implements | UT4S2 |
| **SF34: Filter Run Results** | |
| Description | The user filters the list of all run results. They may filter by classifier, dataset or run name. |
| Implements | UT4S2 |
| **SF35: Display Run Results** | |
| Description | The system displays the results of a run when the user clicks on the "Show Result" Button next to a run. |
| Implements | UT4S2 |
| **SF36: Export Run Results** | |
| Description | The user exports the run results into an annotation. This creates an annotation in WS5: Annotation Master View, where the classification results are created as codes. The annotation can be inspected, changed and used for agreement creation as if it were a manual analysis annotation. |
| Implements | UT4S2 |
| **SF37: Delete Run Results** | |
| Description | The user deletes previous run results that are no longer relevant. |
| Implements | UT4S3 |

Table 7.12.: Automatic Feedback Analysis Workspace Requirements

| **WS12: Classifier Analysis View** | |
|---|---|
| Description | In this view, classifier runs can be configured and executed. The View also provides a list of all previous runs. Users can navigate to WS13: Classifier Results View for each run. |
| Contains | SF30, SF31, SF33, SF34, SF36, SF37 |
| **WS13: Classifier Results View** | |
| Description | In this view, users can inspect the results of a classifier run. Depending on the classifier, this view provides different widgets to show the results. |
| Contains | SF35 |

### Implementation

Because Feed.ai originally already provided some support for classifier integration in the form of a Twitter classifier, integration of the automatic classifier analysis functionalities followed Feed.ai's implementation. Consequently, functionalities were implemented directly into the uvl-orchestration microservice written in GO and do not require any additional services.

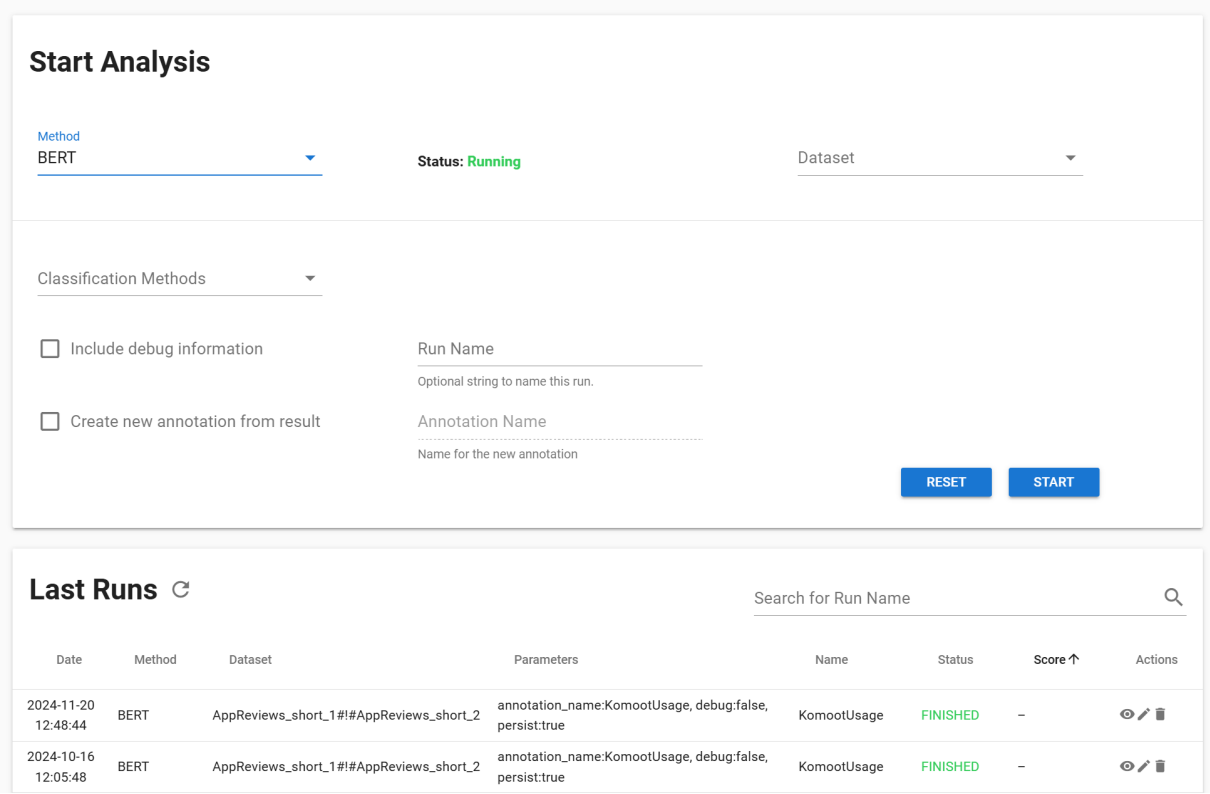



Figure 7.15.: Feed.UVL Classifier Analysis View

Figure 7.15 shows the Classifier Analysis View implemented in Feed.UVL. At the top users select a classifier and a dataset which they wish to analyse. Depending on the classifier selection, the customizable parameters below the selection change dynamically. In the case of the BERT classifier selected here, a more detailed classification method can be selected, debug information can be included in the run results, an annotation can be created from the classification (SF36), and names can be given to the run and the created annotation.

Once the classifier is started, it will appear in the list of "Last Runs" below with an "In Progress" status. Once classification is complete, the status will change to either "Finished" or "Failed" if any errors have occurred. From the list of last runs, users can navigate to the classifier results view, to the annotation (if one was created) or delete the run.

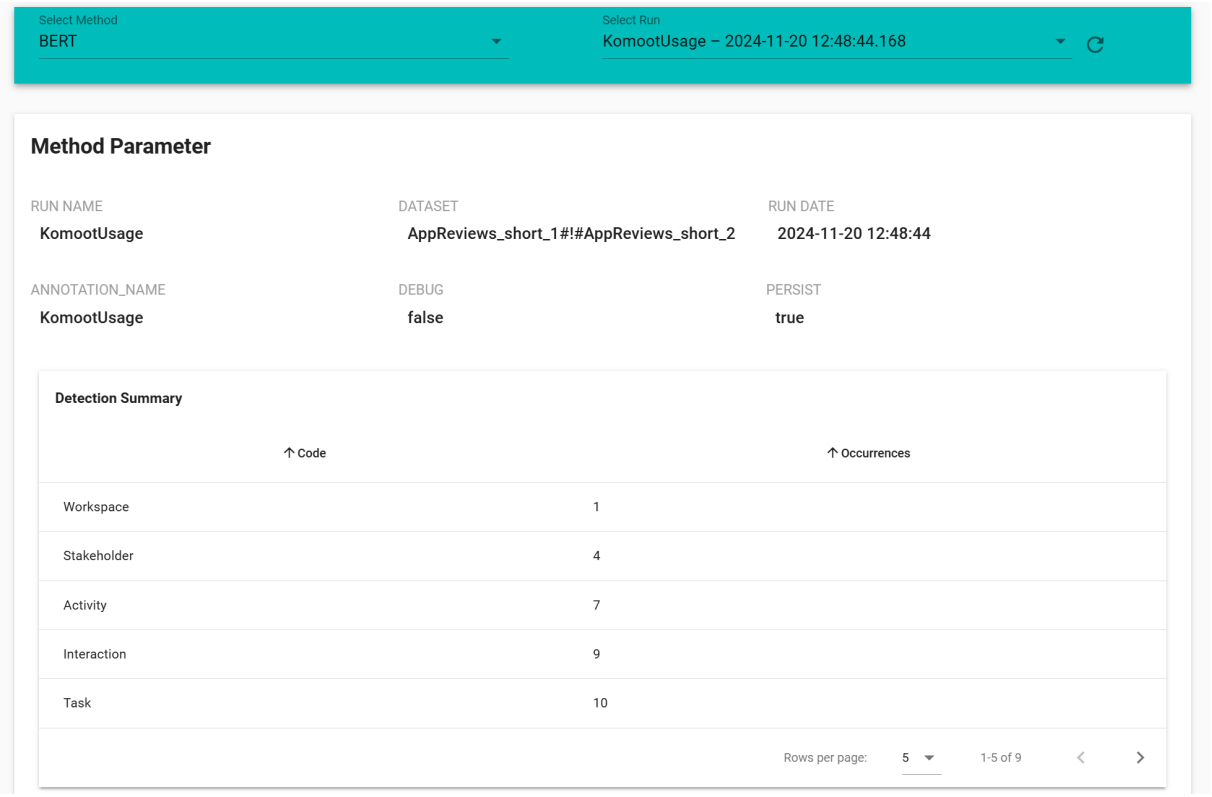



Figure 7.16.: Feed.UVL Classifier Results view (Usage Information)

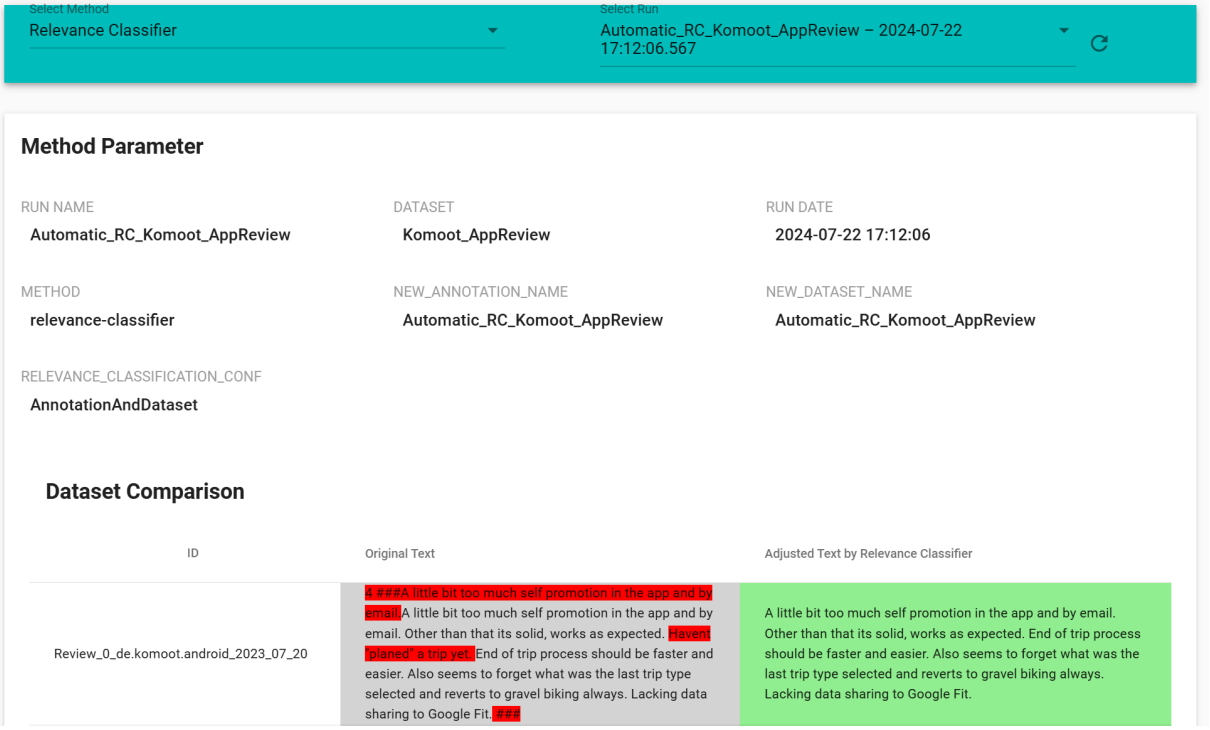



Figure 7.17.: Feed.UVL Classifier Results View (Relevance Classification)

Figures 7.16 and 7.17 show two alternative versions of the Classifier Results View. Because classification goals can differ, results views are customizable to the classification. Figure 7.16

shows the results view for a usage information classifier. The results show a summary of the classification as well as a detailed list of assigned codes (not shown). Figure 7.17 shows the results view for a relevance classifier, which determines if parts of feedback are relevant for developers. Below the general parameter information, the results are displayed as a table showing in red which text segments were removed by the classifier as irrelevant. The new document is then shown on the right.

### 7.2.5. Feed.UVL Dashboard

#### Requirements

Feed.UVL's Dashboard was designed to unify multiple usage information related functionalities spread throughout Feed.UVL while also implementing new functionalities for the relation of feedback and requirements. Consequently, the dashboard can be divided into two separate parts. The first part combines the manual and automatic feedback analysis functionalities described in Sections 7.2.3 and 7.2.4. The requirements for this part will not be reported here again as both the user tasks and system functions are unchanged. Only the workspaces have been slightly altered to match the dashboard layout.

The second part concerns feedback requirements relation. For this the dashboard provides both manual as well as automatic analysis functionalities. The requirements for the manual analysis were derived based on discussions with annotators performing feedback requirements relation. The goal was to provide tool support to simplify their work. Following this, the requirements for the automatic feedback requirements relation were derived based on seamless implementation into the now-established manual analysis functionalities.

Table 7.13.: Feed.UVL Dashboard User Task & Sub-Task Requirements

| **UT5** | **Manage Feedback Requirements Relation** |
|---|---|
| The developer manages multiple feedback requirements relation project simultaneously. For each, they relate user feedback to relevant requirements. | |
| **UT5S1** | **Manage Project Datasets** |
| The developer decides which feedback and which requirements datasets are part of a project. Each project represents a software product the developer is currently working on. | |
| **UT5S2** | **Relate Feedback to Requirements** |
| The developers relates feedback statements to documented requirements. | |
| **UT5S3** | **Discard Outdated Relations** |
| The developer discards outdated feedback requirements relations that are no longer relevant. | |
| **UT5S4** | **View existing Feedback Requirements Relation** |
| The developer views a previously created feedback requirement relation. | |

Table 7.13 describes the primary user task for the feedback requirements relation dashboard and its sub-tasks. UT5 concerns the management of the feedback requirements relation. The developer's goal is to perform the relation for multiple software projects and administrate the datasets used in each one. This includes the feedback and the requirements relevant to each project. This task is divided into four sub-tasks. UT5S1 concerns the general administration of the datasets. UT5S2 - UT5S4 are related to the actual relation of feedback to requirements. This includes the creation of new relations, discarding of old relations and viewing of existing relations.

Table 7.14.: Feed.UVL Dashboard System Function Requirements

| **SF38: Create Dashboard** | |
|---|---|
| Description | The user creates a new dashboard by selecting whether they wish to perform usage information classification or feedback requirements relation. They also set a unique name for the dashboard. |
| Implements | UT5S1 |
| **SF39: Load Dashboard** | |
| Description | The user loads a previously created dashboard by selecting the name of the dashboard they wish to load. |
| Implements | UT5S1 |
| **SF40: Add Feedback Dataset** | |
| Description | The user selects from the list of feedback datasets stored in Feed.UVL, which datasets should be related to requirements. The user may select one or multiple datasets. |
| Implements | UT5S1 |
| **SF41: Import Requirements Dataset** | |
| Description | The user imports the requirements for a software from Jira. For this, the user selects the name of the Jira project in which the requirements are stored. Then, they select which specific requirements they would like to import. These are then stored in Feed.UVL's database. |
| Implements | UT5S1 |
| **SF42: Remove Dataset** | |
| Description | The user removes either a feedback or requirements dataset from the dashboard. These are then deleted from the dashboard's storage. Feedback datasets removed from a dashboard are still retained in Feed.UVL's database. |
| Implements | UT5S1 |
| **SF43: Delete Dashboard** | |
| Description | The user deletes a dashboard that is no longer relevant. All information therein is removed. Feedback datasets are still retained in Feed.UVL's database |
| Implements | UT5S1 |
| **SF44: Update Dashboard** | |
| Description | The system automatically checks if a dataset used in a dashboard that is currently being loaded (SF39) has been updated since the last time the dashboard was used. The dashboard then warns the user of this update. |
| Implements | UT5S1 |
| **SF45: Manually Relate Feedback to Requirement** | |
| Description | The user manually relates feedback to a requirement by selecting every feedback from a list that they wish to add to a specific requirement. For this a button is provided under the list of requirements (SF48). |
| Implements | UT5S2 |
| **SF46: Automatically Relate Feedback to Requirement** | |
| Description | The system automatically performs the feedback requirements relation when prompted to by the user. The user prompts the automatic relation by pressing a button provided in the dashboard. A loading dialogue appears while the relation is being performed. Afterwards the dashboard reloades with the results of the relation. |
| Implements | UT5S2 |
| **SF47: Delete Releation** | |
| Description | The user deletes the relation of a specific feedback to a specific requirement. For this, a button is provided next to each feedback requirements relation. |
| Implements | UT5S3 |
| **SF48: List Feedback Related to Requirement** | |
| Description | The system lists every requirement imported from Jira (SF41). Under each requirement, the feedback related to that requirement is listed. |
| Implements | UT5S4 |
| **SF48: Filter List of Relations** | |
| Description | The user filters the list of requirements or of related feedback. This is done by entering a search string into the provided search bars. Only feedback or requirements containing the search string are displayed. Alternatively, the user may also only show requirements which do not have any feedback related to them. |
| Implements | UT5S4 |

Table 7.14 lists the system functions that derive from the previously introduced user tasks. SF38 allows users to create a new dashboard by selecting whether they wish to perform usage information classification or feedback-requirements relation and by assigning a unique name to the dashboard. SF39 enables users to load a previously created dashboard by selecting its name. SF40 provides the functionality for users to choose one or more feedback datasets stored in the system's database to relate them to requirements. SF41 allows users to import requirements datasets from Jira by selecting the Jira project and specific requirements, which are then stored in the system's database. SF42 lets users remove feedback or requirements datasets from a dashboard, ensuring that removed feedback datasets remain stored in the database. SF43 permits users to delete dashboards that are no longer needed, with feedback datasets still retained in the system's database. SF44 ensures the system checks for updates to datasets used in a dashboard being loaded and alerts users to any changes. This prevents users from relating outdated feedback by informing them of changes to the datasets. SF45 allows users to manually associate feedback with specific requirements by selecting feedback from a list and assigning it to a requirement. SF46 automates the process of relating feedback to requirements when triggered by the user, displaying the results after processing. SF47 lets users delete specific feedback-to-requirement relations via a button next to each relation. SF48 displays a list of requirements imported from Jira, with feedback related to each requirement listed below it. Additionally, SF49 enables users to filter the list by entering a search string, displaying only matching elements.

Table 7.15.: Feed.UVL Dashboard Workspace Requirements

| **WS14: Dashboard Selection View** | |
|---|---|
| Description | In this view, the user decides which dashboard they would like to load. They may also create a new dashboard from this view. |
| Contains | SF38, SF39, SF43 |
| **WS15: Feedback Requirements Relation Dashboard View** | |
| Description | This is the main view of the feedback requirements relation dashboard. It consists of the configuration view and the relation view. |
| Contains | SF44 |
| **WS15.1: Dashboard Configuration View** | |
| Description | In this view, the user selects their feedback and requirements datasets for the currently loaded dashboard. |
| Contains | SF40, SF41, SF42 |
| **WS15.2: Dashboard Relation View** | |
| Description | In this view, the user can see a list of all existing feedback requirements relations. Here, they can add new ones manually or automatically. They can also remove existing relations. |
| Contains | SF45, SF46, SF47, SF48, SF49 |

Table 7.15 outlines the workspaces that group the system functions. WS14 contains functionalities for dashboard creation, loading and deletion. WS15 is divided into two sub-workspaces. WS15.1 groups the functionalities for dataset management for the dashboard. WS15.2 groups functionalities for relating feedback and requirements both manually and automatically.

### Implementation

To maintain Feed.UVL's microservice architecture, the dashboard, was implemented in such a way that the usage information and feedback requirements relation sections can function independently. For the usage information, most of the functionalities were extracted from already existing containers in Feed.UVL. Some changes were made to these to function with multiple datasets simultaneously, which was not possible previously. The backend for the feedback requirements relation functionalities was created in a new container written in Python. This

was done to seamlessly integrate with automatic relation functionalities, which were written in Python due to Python's availability of machine learning libraries.

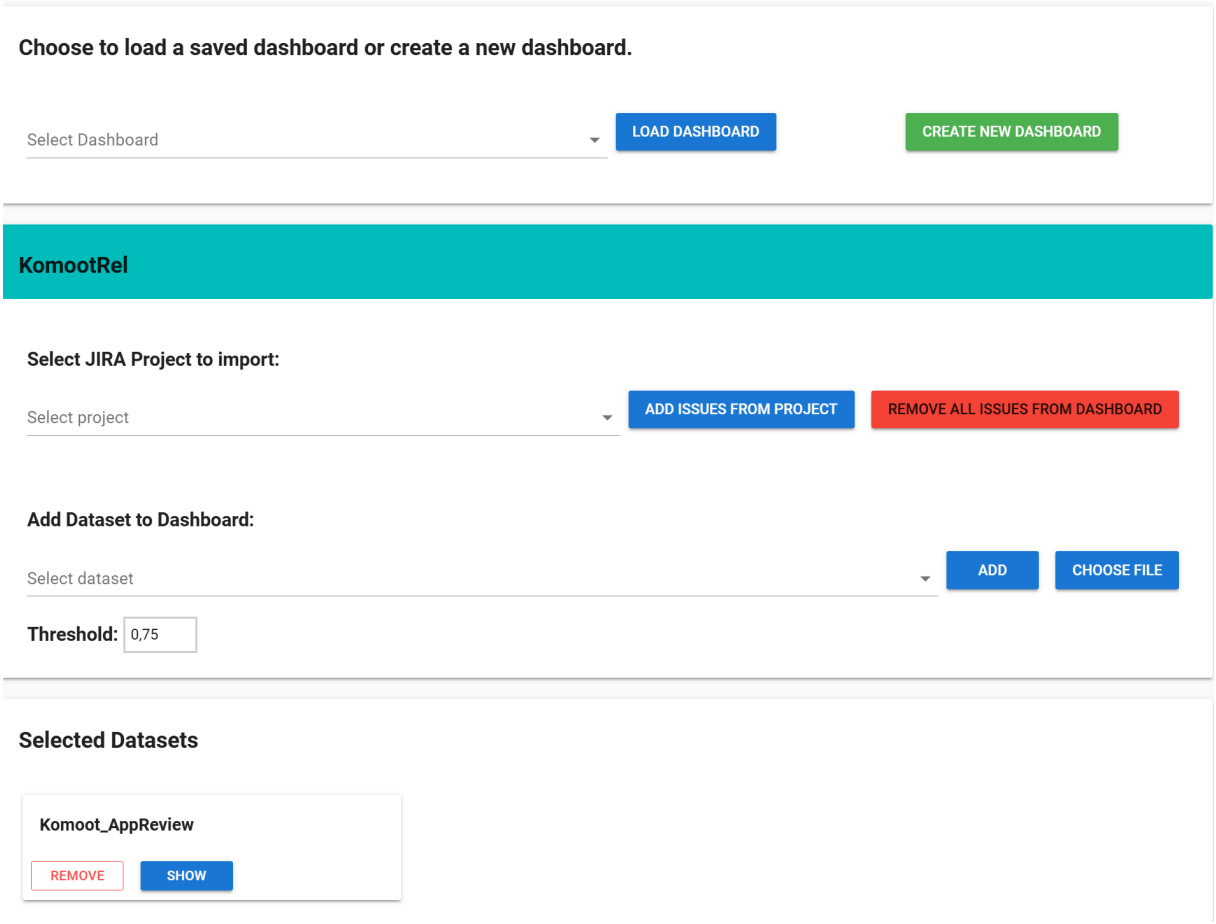



Figure 7.18.: Feed.UVL Dashboard Selection and Configuration View

Figure 7.18 shows the implementation for WS14 at the top and WS15.1 below. At the top, users can select the dashboard they wish to load from a dropdown. They may also create a new dashboard which will open a pop-up which allows them to set a name for the dashboard and select whether they wish to perform usage information classification or feedback requirements in the dashboard. Below are the selection options for the import of requirements from Jira. This is implemented by accessing Jira's API to import issues from within the selected Jira project. Users are shown a list of Jira issue types they wish to import in a pop-up. Afterwards, they can select all or specific issues from the project to import. Below the Jira import, users can select from the list of datasets stored in Feed.UVL. They may select one or multiple of these datasets and add them to the dashboard.

Figure 7.19 shows an excerpt from the implementation of WS15.2. In this view, which is placed below WS15.1, users are presented with a list of all imported requirements in a table. A user may click on a requirement, which unfolds the list of all feedback related to that requirement. In this view, users may delete all relations, relate new feedback to the requirement or delete individual relations. A button is also provided for each requirement to delete it from the dashboard. At the top of the view, a button allows users to start the automatic relation of feedback and requirements based on the datasets selected in WS15.1. Search filters are provided for requirements and feedback to filter for specific texts.

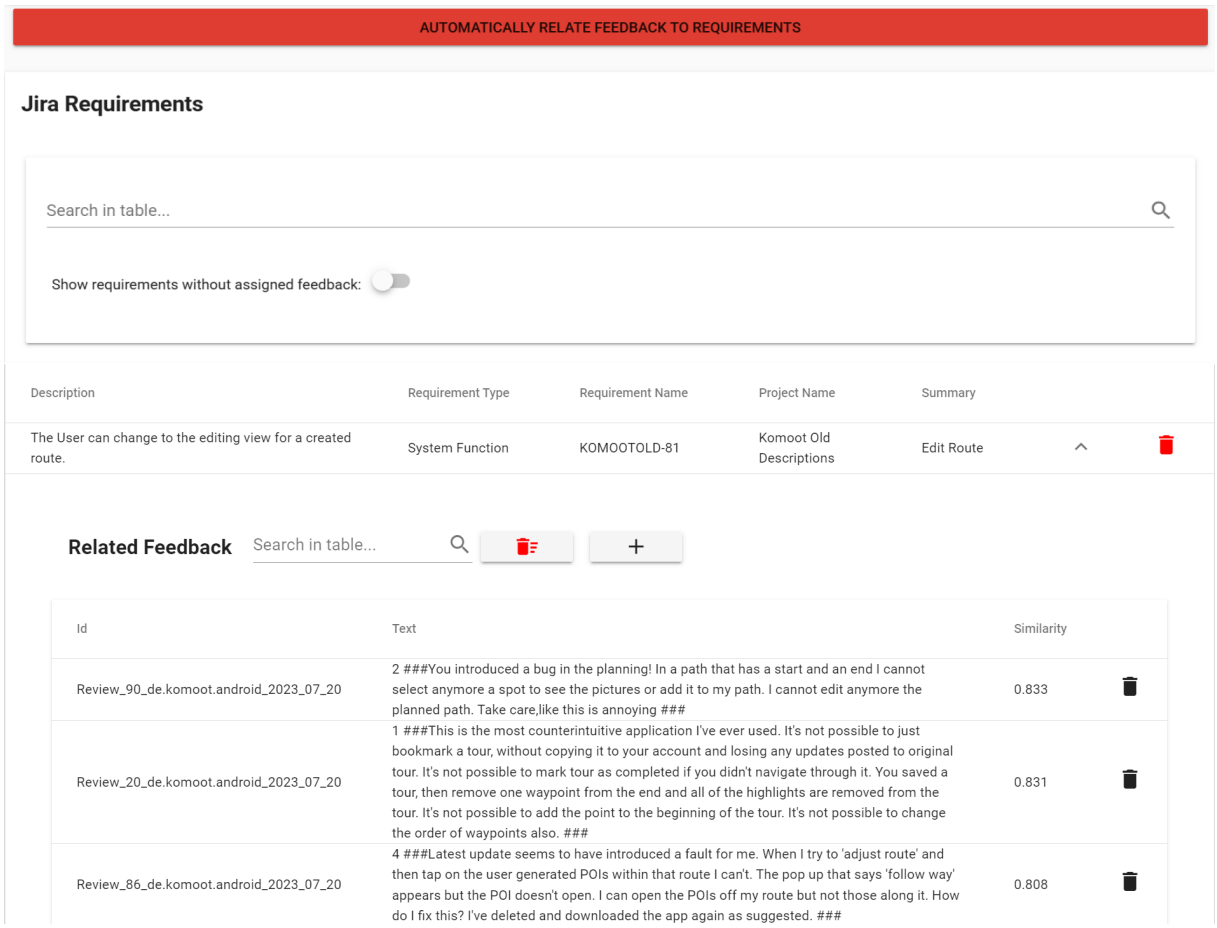



Figure 7.19.: Feed.UVL Dashboard Relation View

Figure 7.20 shows a screenshot of the usage information classification dashboard whose requirements are omitted in this section. As can be seen, the dashboard reuses views and functionalities from the manual and automatic classification functionalities of Feed.UVL (Sections 7.2.3 and 7.2.4). Users select datasets they wish to analyse, select the classifiers they wish to perform the analysis with and can see and change the results of the classification in the form of an automatically created annotation.

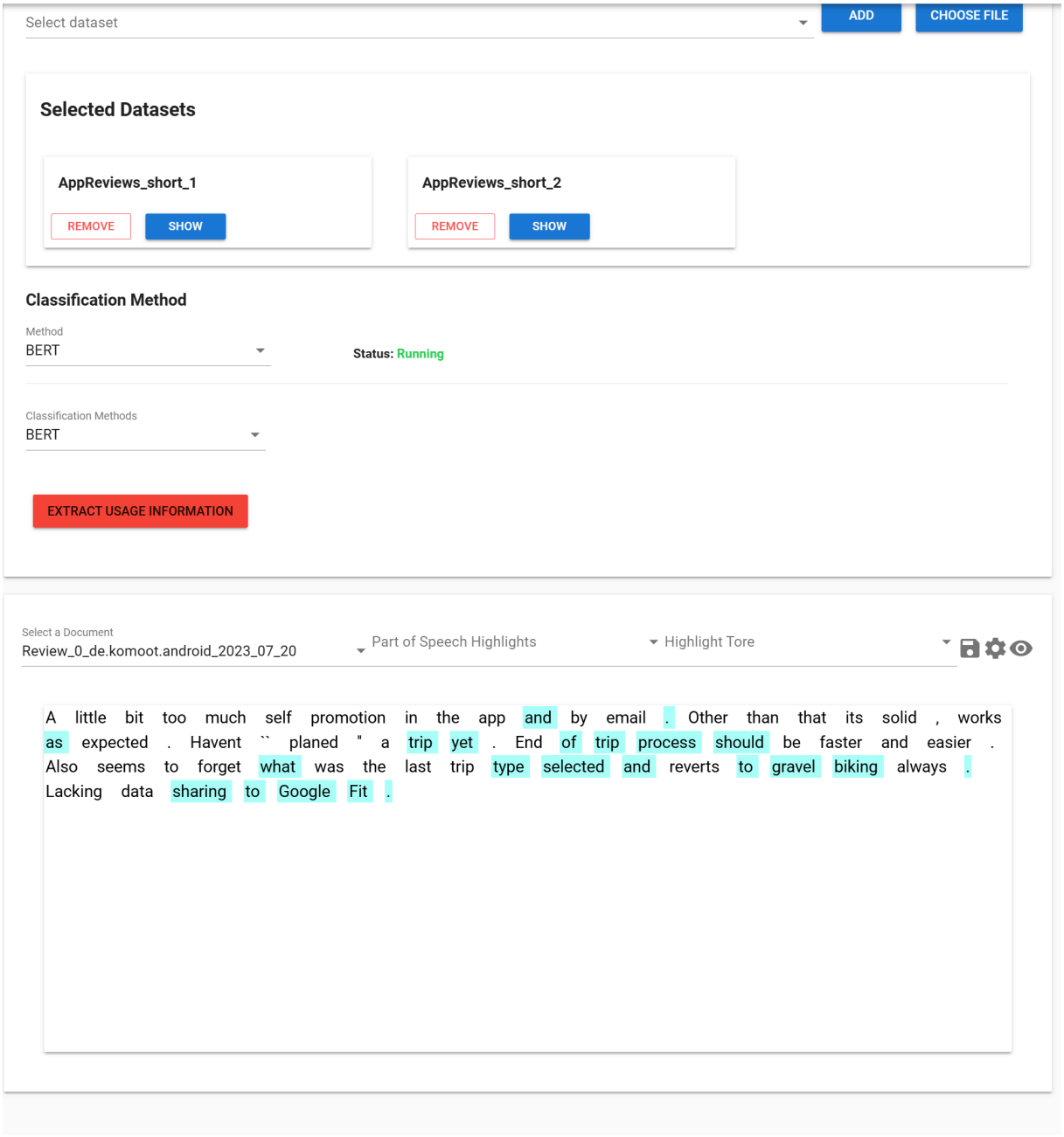



Figure 7.20.: Feed.UVL Dashboard Usage Information Classification

### 7.2.6. Jira Plugin

#### Requirements

Requirements for the Jira plugin were derived from a need to make the information created and stored in Feed.UVL more accessible to developers without requiring them to switch tools constantly. More specifically, the plugin is targeted towards allowing developers to see feedback related to a requirement directly next to that requirement in Jira. Additionally, it allows them to highlight the usage information contained in that feedback within the same view. Note that requirements are stored as issues in Jira. Each issue represents a different requirement.

We do not specify new user tasks for the plugin as its purpose is not to support fundamentally different tasks than those specified by UT5S4, UT4S2 and UT3S2. The purpose of the plugin is merely to make this information more easily accessible.

Table 7.16.: Jira Plugin System Function Requirements

| **SF49: Configure Jira Plugin** | |
|---|---|
| Description | The user specifies the names of the dashboards for the usage information classification and feedback requirements relation they wish to import into Jira. |
| **SF50: Import Dashboard Information** | |
| Description | The system imports the information of the dashboards specified in SF49 and stores it in Jira. For this, the API provided by Feed.UVL is used. |
| **SF51: Display Related Feedback** | |
| Description | The system displays a list of related feedback for every requirement. |
| **SF52: Highlight Usage Information** | |
| Description | The user selects the highlighting of individual types of usage information based on the TORE Model. The selected usage information is then highlighted in different colours in the feedback. |
| **SF53: Hide Feedback** | |
| Description | The user hides individual feedback statements for an issue. This feedback is then no longer displayed for that issue. |
| **SF54: Filter Feedback** | |
| Description | The user filters the feedback by entering a search string. Only feedback containing that string is displayed. |

Table 7.16 lists the system functions of the Jira plugin. SF49 allows users to configure the Jira plugin by specifying the names of dashboards for usage information classification and feedback-requirements relation that they wish to import into Jira. SF50 enables the system to import the specified dashboard information using the API provided by Feed.UVL and store it in Jira. SF51 displays a list of feedback related to each requirement within Jira. SF52 allows users to highlight specific types of usage information based on the TORE Model (Section 2.2), with different types highlighted in distinct colours within the feedback. SF53 enables users to hide individual feedback statements associated with an issue, ensuring the hidden feedback is no longer displayed for that issue. It should be noted that this does not affect the information stored in Feed.UVL. SF54 provides functionality to filter feedback by allowing users to enter a search string, displaying only feedback that contains the specified string.

Table 7.17.: Jira Plugin Workspace Requirements

| **WS16: Jira Plugin Settings View** | |
|---|---|
| Description | In this view, the user specifies the names of the dashboards they would like to import from Feed.UVL. The system then imports the feedback requirements relation and usage information classification. |
| Contains | SF49, SF50 |
| **WS17: Jira Plugin Issue View** | |
| Description | In this vie,w the feedback is displayed. The view is part of Jira's default Issue View, which displays all information related to a specific issue. |
| Contains | SF51, SF52, SF53, SF54 |

Table 7.17 shows the workspaces which group the system functions of the Jira plugin. WS16 provides an interface for users to enter the dashboards they would like to import into Jira from Feed.UVL. WS17 groups all functionalities related to the actual displaying of the imported information.

### Implementation

The Jira plugin was written in Java code due to Atlassian's (the developer of Jira) extensive support for plugin development using Java. Jira plugin development was done using Atlassian's provided SDK, which provides help with the API, development and running of test environments. To access information from Feed.UVL, the API of the dashboard container, was extended to extract the needed information via API requests and store it on the Jira server. The automation of this means that users only need to specify the names of the dashboards they wish to import without requiring any further setup of the plugin.

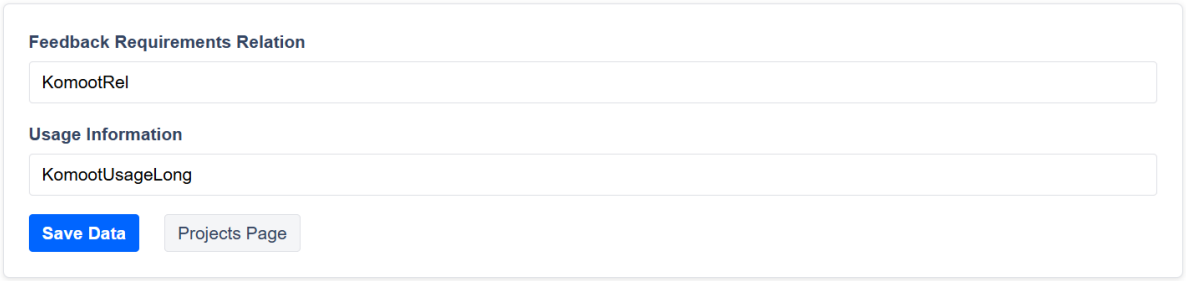



Figure 7.21.: Jira Plugin Settings View

Figure7.21 shows the implementation of WS16: Jira Plugin Settings View. The view is accessible via the projects setting in every Jira project. As a result, users can specify different dashboards to import for every project. The names of the two dashboards storing the feedback requirements relation and usage information classification have to be entered in the provided input fields. The *Save Data* button then imports the information. Should the dashboards change in Feed.UVL, a click on the same button will update the information in Jira. A button is also provided to navigate to the project's issue page.

Figure 7.22 shows the plugin inside of the Jira default issue page on the right. The plugin uses some default functionalities provided by Jira to resize depending on the screen it is being looked at with. For every issue (i.e. every requirement), the plugin will display the list of related feedback. By clicking on the eye next to a feedback statement, the feedback is hidden for that requirement and will no longer be displayed. The top right of the feedback panel offers three buttons. The button on the left opens the highlighting menu. Here, users can select which type of usage information they wish to highlight. Highlighting can be seen in the example feedback shown in the figure. Note that the highlighted information here does not necessarily represent actual usage information and is only highlighted for demonstration purposes. The middle button opens the search bar that allows users to only see feedback containing a specific string. The button on the right will hide the feedback panel completely in case users do not wish to see it.

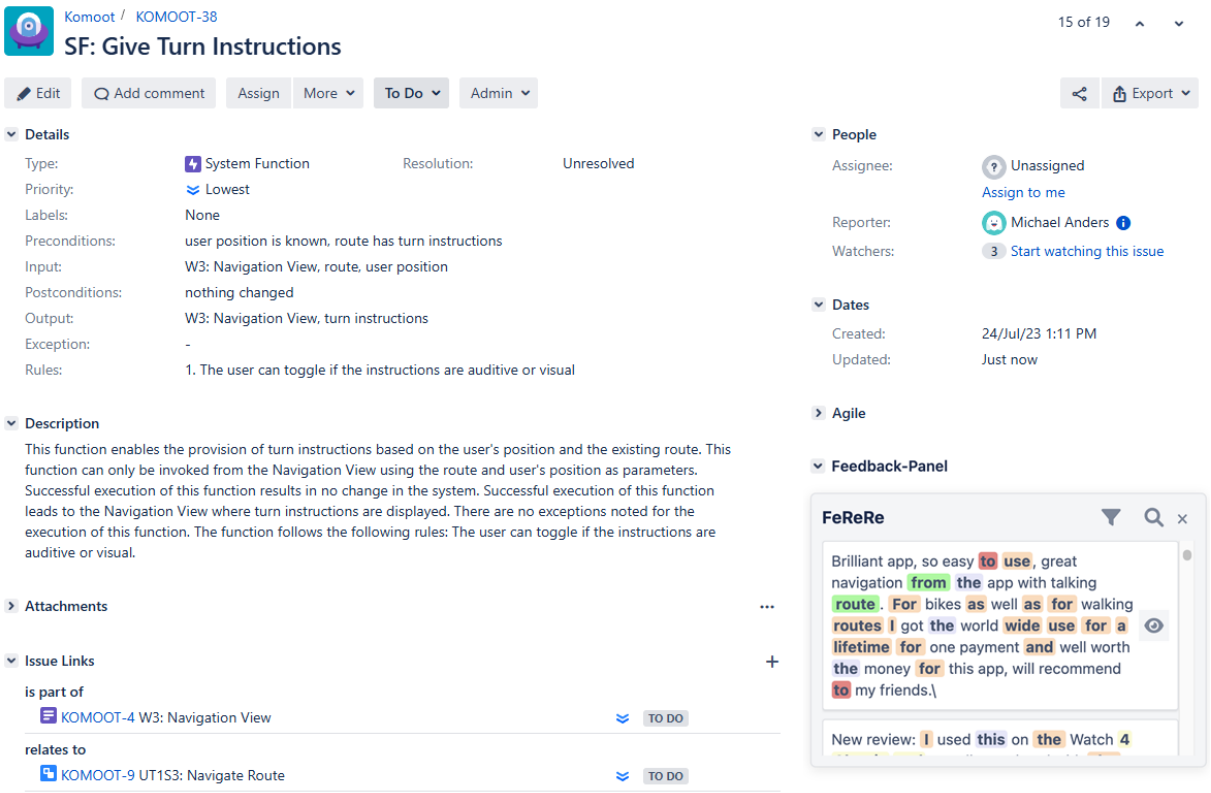



Figure 7.22.: Jira Plugin Issue View

## 7.3. Conclusion

During its development as part of this dissertation Feed.UVL has been applied in different research contexts, demonstrating its adaptability and effectiveness in classification tasks. In this dissertation, Feed.UVL was mainly utilized to create the usage information und feedback requirements relation datasets presented in Section 2.5. It was also utilized to validate the classifiers presented in the solution investigation of this disseration (Chapter II). Consequently, this dissertation benefited from its structured approach to data analysis and its integration of both manual and automated feedback classification methods. The ability to compare annotations and evaluate classifier performance provided a robust framework for ensuring reliable results.

Beyond the research in this dissertation, Feed.UVL's manual annotation functionalities have been successfully employed in a project focused on investigating user-developer communication that the author of this dissertation was part of (Anders et al., 2022). Other participants of the project reported that compared to commercial alternatives like MAXQDA, Feed.UVL offered a more efficient workflow, allowing participants to categorize and analyze feedback far more quickly. The combination of manual annotation and visualization functionalities facilitated a structured analysis process, improving the overall quality of use. Especially, the recommendation function of the manual annotator was praised.

Additionally, Feed.UVL has been utilized in an independent research project investigating user story similarity. The tool's microservice-based architecture and modular design enabled its application in a domain beyond its initial scope without extensive modifications. This demonstrates its potential for broader applicability in various classification and analysis tasks.

However, as Feed.UVL was only intended as a prototype to support the other focuses of this dissertation, namely feedback requirements relation and usage information classification, it lacks some functionalities to be more applicable to practical usage. Future work on the

tool should increase security by implementing a role-based access control system with limited permissions. Currently, the tool is locked behind a single password, giving everyone with that password complete access to the frontend of the application. While the server infrastructure is locked behind a permission system, the frontend is not. Implementing role-based security would strengthen Feed.UVL further.

Also, investigating the practical application of Feed.UVL would be an interesting endavour by implementing it into the workflow of a company. Unfortunately, this was out-of-scope for this dissertation but could provide an entry for future research.

# Part IV.

# Treatment Validation

Chapter 8

# Feedback Requirements Relation Evaluation

The FeReRe approach was not evaluated in a real-life practical scenario in this dissertation. Instead, this section discusses the effects of observations made during the evaluation of the classifier and comparison with human coders on a potential practical application. The goal is to discuss whether problem *P1: Understand which functionalities users are discussing in their feedback*, introduced in Section 1.1, is solved by the approach and whether the use cases introduced in Section 5.1.2 are adequately supported.

Section 8.1 introduces the different factors considered for the evaluation. Section 8.2 discusses these factors in detail for a company wishing to utilise FeReRe. Lastly, Section 8.3 concludes the chapter.

## 8.1. Evaluation Methodology

The evaluation presented in this chapter discusses the theoretical deployment in a software development company which wishes to utilize the FeReRe approach to create connections between feedback and requirements for requirements validation as proposed in the FeReRe use case in Section 5.1.2. Different considerations are necessary to assess whether the utilization of FeReRe would fit the company. These considerations are the time efficiency and risk of missed relations by the classifier as well as deployment requirements for software, hardware and the long-term maintainability of the approach.

To assess these factors, we conduct a comparative evaluation against manual human classification. For this comparison, we evaluate the balance between time efficiency and classification accuracy. For time efficiency, we measure the time required for both manual and automated feedback relation processes. The comparison highlights potential time savings when using FeReRe.

For the risk of missed relations, we analyze the impact of incorrect classifications, focusing on missed relations (false negatives) and incorrect relations (false positives) because these lead to missing information when utilizing a classifier instead of performing the classification manually. The implications of these inaccuracies on practical development workflows are discussed, particularly in the context of human validation in a semi-automatic approach.

For the deployment requirements, we discuss the necessary hardware to run the classifier, how it is best integrated into existing software workflows, and the requirements for its long-term maintenance. We provide recommendations on deployment strategies, taking into account the most important constraints: the need for initial training data, the potential for continuous learning, and the adaptability of the approach to varying development environments.

## 8.2. Evaluation Scenario

Section 8.2.1 introduces the company wishing to use FeReRe and discusses the manual aspects of feedback requirements relation without the utilization of FeReRe. Section 8.2.2 compares these manual aspects to the time savings when using the semi-automatic FeReRe approach. Section 8.2.3 calculates the effect of missed relations due to imperfect classification. Section 8.2.4 discusses hardware and software requirements for the deployment of the approach. Section 8.2.5 discusses the requirements for the long-term maintainability of the classifier.

### 8.2.1. Scenario

We orient the evaluation scenario towards a hypothetical deployment in the Komoot gmbh, which develops the Komoot hiking app. In this hypothetical scenario, it is the goal of the company to utilise the FeReRe approach to semi-automatically relate feedback to requirements in order to validate whether their requirements match with the user's needs. This is in line with the primary use case for FeReRe as introduced in Section 5.1.2. We chose this company because the Komoot App Store dataset (Section 2.5.2) we have created is the most representative of our datasets. It consists of 335 unprompted feedback crawled from the actual app store page of the Komoot app, 79 recreated requirements for the entire Application and a manually created relation between the feedback and requirements. In contrast to the SmartVernetzt and SmartFeedback apps, Komoot also represents a commercial product with over 40 million users[1]. The research project scenario in which SmartFeedback and SmartVernetzt were created is also less representative of practical application. The ReFeed dataset is not used because we lack the necessary information about the software and source of the gathered feedback.

The Komoot gmbh is a medium sized company currently employing more than 65 members. We do not have concrete numbers on the number of feedback the company receives daily. However, a 2013 study found that, on average, apps receive around 22 new feedback messages a day (Pagano and Maalej, 2013). This number is likely to have risen since 2013, given the rise in popularity of mobile phones over the years. However, it is also skewed due to the thousands of reviews that larger apps receive in a day. Because more recent or more detailed studies aren't available, we assume the number of 22 new feedback messages for the purpose of this evaluation.

During the creation of the Komoot dataset (Section 2.5.2), it took an average of around 2-3 seconds to relate a feedback statement to a requirement. The dataset contains 335 feedback statements and consists of 79 requirements. The creation of this dataset took around 15-20 hours, varying slightly between coders. This time does not include time spent on creating interrater agreements or correcting any relations. For the purpose of this evaluation, we assume that a practical application in the company would not include the creation of interrater agreements due to the time and effort spent on this.

Note that in order to create a complete relationship between feedback and requirements, each feedback statement needs to be compared to each requirement to determine if they are related. Comparing 335 feedback with 79 requirements results in a total of 26.465 comparisons. Divided over 15 hours, this creates an estimate of around 2 seconds per feedback requirements comparison.

Assuming that the developer receives 22 new feedback statements a day and relates these to their 79 requirements, a total of around one hour would need to be spent to create the complete relation each day. This number, of course, is only a rough estimate as it depends on multiple factors, namely the length of the feedback, the number of requirements that the software has, the efficiency of the coder and how familiar they are with the requirements of the software. In the final dataset, a feedback statement is related to 2.5 requirements on average. This means

[1] https://www.komoot.com/jobs

Table 8.1.: Manual Relation Scenario

| Software | Requirements | Daily Feedback | Possible Relations | Actual Relations | Time Spent |
|---|---|---|---|---|---|
| Komoot | 79 | 22 | 1738 | 55 | 58 min |
| SmartVernetzt | 31 | 22 | 682 | 23 | 23 min |
| SmartFeedback | 29 | 22 | 638 | 23 | 21 min |
| ReFeed | 14 | 22 | 308 | 67 | 10 min |

that 22 new feedback statements would result in approximately 55 relations between feedback and requirements.

Table 8.1 lists the above-mentioned number of requirements, daily feedback messages, possible and actual relations, as well as the time spent creating these relations. We also include these numbers for SmartVernetzt, SmartFeedback and ReFeed for completeness. Note that due to the more targeted feedback gathered for the SmartAge apps through the questions asked in SmartFeedback, the average number of requirements related to feedback is only 1.1 for both apps compared to 2.5 for Komoot. The average for the ReFeed dataset is 3.1, but as discussed in Section 2.5.2, this dataset does not contain all requirements of the discussed software. Consequently, the ReFeed dataset is not further considered in this evaluation.

### 8.2.2. Time Efficiency

When we disregard the time spent on training the classifier (as we also disregard time spent for humans to get familiar with the software and its requirements), the time it takes for the classifier to relate 22 feedback statements to 79 requirements becomes negligible even on a moderately powerful computer without any dedicated GPUs. The machine, using an AMD Ryzen 7 7800X3D processor and 32GB of DDR5 RAM, needs around 90 seconds to perform the classification, most of which is spent loading the model. The time is reduced to around 15-16 seconds on a computer where the classifier is already deployed, thus removing the time needed to load the model. If a dedicated GPU is used for classification, the classification takes only around 2.5 seconds using an NVIDIA GeForce RTX 3080 Ti. In any of these scenarios, classification is much more time-efficient than the one hour of manual labour required.

If the classifier were to be used fully automatically, then no further manual labour would be required. In a semi-automatic approach, such as the one proposed in this dissertation, some human effort would be required of the developer after the classification. For this semi-automatic approach, the classifier presents the list of all relations it has classified. The developer can then accept or decline these.

When we split the 335 feedback in the Komoot dataset into chunks of 22 and let the classifier relate these to the 79 requirements, it produces an average of 63 relation recommendations per 22 feedback. Using the previous 2-second estimate to decide if feedback and requirements are related, the developer would need to spend around 2 minutes approving or rejecting the proposed relations. This is a much shorter time needed than the fully manual approach, which would require 58 minutes.

### 8.2.3. Risk of Imperfect Classification

While the classifier is able to perform the relation much more quickly than a human, this likely comes at a cost to the accuracy of the classification. The high kappa values presented in Section 2.5.2 indicate that humans can perform the classification relatively accurately. While there is still a risk that multiple coders could make the same mistake, Kappa can be an indicator of the reliability of the coding process (Carletta, 1996). A high number of disagreements would result in a low kappa value and would indicate that humans struggle to perform the classification. Given

the high kappa values, we assume a perfect human relation for the purpose of the comparison to the classifier's performance.

As shown in Section 5.2.4, the Kommot classifier achieves a precision of 0.71 and a recall of 0.86. Using the example 22 feedback statements, assuming the average relation to 2.5 requirements we have in our datasets, we can calculate the impact of the imperfect classification as follows:

- Number of feedback: 22
- Number of requirements: 79
- Average relationships per feedback: 2.5
- Total relations: 55 = Number of feedback x Average relationships per feedback
- Precision: $P = 0.71$; Recall: $R = 0.86$

#### Missed Relationships Due to Recall (False Negatives)

$$Recall = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{8.1}$$

$$\text{FN} = \frac{\text{TP}}{Recall} - \text{TP} \tag{8.2}$$

$$\text{TP} = Recall \times 55 = 0.86 \times 55 = 47.3 \tag{8.3}$$

$$\text{FN} = 55 - 47.3 = 7,7 \tag{8.4}$$

By solving the formula used to calculate recall (Formula 8.1) for the false negatives (FN), we get Formula 8.2. By inserting the above-listed assumed values, we receive a total of 7.7 false negatives. Thus, approximately eight correct relations are missed by the classifier out of 55 that would otherwise be classified by a human. This means about 15% of the relations are missed. The company wishing to deploy FeReRe would have to assess whether this risk is worth the potential benefits of creating the relations. If they are not created at all in their workflow, then even an imperfect relation, which misses 15%, would be beneficial.

Precision is lower than recall at 0.71. The impact of the precision score can be calculated as follows:

#### Incorrect Relationships Due to Precision (False Positives)

$$Precision = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{8.5}$$

$$\text{FP} = \frac{\text{TP}}{Precision} - \text{TP} \tag{8.6}$$

$$\text{FP} = \frac{47.3}{0.71} - 47.3 \tag{8.7}$$

$$\text{FP} = 66.6 - 47.3 = 19.3 \tag{8.8}$$

As previously done for the recall formula, we solve the precision calculation Formula 8.5 for the false positives and receive Formula 8.6. Inserting the assumed values, the classifier incorrectly creates around 19 feedback requirements relations. However, in a semi-automatic approach, as proposed in this dissertation, the precision value becomes much less relevant as

the recommendations of the classifier are corrected by a human. Consequently, the 19 incorrect recommendations would be filtered out. The only relevance precision retains is in the amount of manual labour required to remove these false positives. As discussed in the previous section, however, only around 2 minutes are required for this task. Ultimately, these numbers are only representative of our available data and would have to be assessed by the company wishing to utilize FeReRe as they may be different.

### 8.2.4. FeReRe Deployment

Given the time efficiency and missing accuracy considerations discussed in Sections 8.2.2 and 8.2.3, we now discuss the feasibility of deploying the FeReRe approach in a real-world software development workflow.

#### Hardware & Software Requirements

A practical deployment of the classifier would involve integrating it into an existing requirements management system or a feedback analysis tool. This could be achieved in one of the following ways:

- Standalone Application: A separate tool where developers upload feedback and receive related requirements.
- Plugin for Issue Tracking Systems: Integration with platforms like Jira or Azure DevOps, where feedback can be automatically linked to requirements within the project.
- Automated Notification System: Developers receive daily or real-time notifications suggesting requirement relations based on incoming feedback.

Given the classifier's computational requirements (Section 8.2.2), it can operate on developers' workstations provided they are equipped with adequate processors or preferably a dedicated GPU. However, deployment in a cloud environment is recommended to minimize the load on individual workstations. This also guarantees that all developers access the same classification instead of performing it individually on each device. The time taken for classification - ranging from 2.5 to 90 seconds depending on the hardware configuration (as discussed in Section 8.2.2) - ensures that automated suggestions can be provided in a short time, minimizing workflow disruption.

Feed.UVL (Chapter 7) provides an example prototype for a feedback analysis tool which integrates FeReRe into the analysis workflow. The accompanying Jira plug-in demonstrates how the information generated by FeReRe can then be integrated more seamlessly into an existing system without requiring dedicated tools just to utilize the FeReRe information.

#### Classifier Training Requirements

As shown in Section 5.2.4, the generalizability of the classifier is poor. While this is a consistent observation for machine learning classifiers (Devine and al., 2023), it means that domain-specific training is necessary for every software that the approach is to be used on. Consequently, training data is necessary for the classifier, meaning that a manual ground truth has to be created once to train the classifier adequately. As our experiments showed, this training data has to contain a substantial amount of feedback. The ReFeed dataset, for example (Section 2.5.2), does not contain enough statements with only 60 feedback. A concrete number of feedback statements is difficult to estimate because it depends on factors such as the number of requirements in the project, the length of the feedback and how many requirements are covered by the feedback. The classifier would require training on a growing volume of feedback and continuous evaluation until

additional training data no longer leads to further improvements. The balance of the dataset, i.e. how much feedback is related to each requirement, could also affect classification, though we could not perform any experiments towards this due to a lack of data.

The creation of the ground truth would cost a substantial amount of manual labour, even though this process would likely only need to be done once per software. An alternative method would be to utilize continuous learning for the classifier. In continuous learning (also called active learning or human-in-the-loop in literature) (Mosqueira-Rey et al., 2023), the classifier trains based on the approval and rejection of its recommendations by humans. This means that the approach could be deployed with an untrained classifier, which is continuously updated based on the feedback it receives due to human intervention. This negates the need for a ground truth at the cost of poor performance at the beginning of the project. Developers would be required to continuously reject poor recommendations by the classifier until it has gathered enough data to improve performance. Whether ground truth creation or continuous learning is preferable is ultimately dependent on the project's resources. Continuous learning is recommended due to the long-term potential for improvement beyond the capabilities of a created ground truth. A mixture of both approaches is also possible, where a smaller ground truth is first used to train the initial classifier, which is then continuously trained through human approval or denial of the classifier recommendations (Settles, 2009).

### 8.2.5. Maintainability

A restriction on the long-term maintainability of the approach is the need for retraining based on changing requirements or feedback sources. As requirements or feedback sources change during the course of a software project, the classifier is required to retrain in order to not provide recommendations for relations based on outdated information. Continuous learning would provide a solution to this, as over time, the outdated requirements or feedback sources would be replaced by new information. This, again, however, comes at the cost of short-term performance. Retraining the classifier would update the information instantaneously, while continuous learning would take time for the classifier to update.

In summary, a classifier utilizing continuous learning is the preferred deployment strategy for the FeReRe approach in a practical setting in order to avoid the necessity for large amounts of training data up front and because of changing requirements.

## 8.3. Conclusion

In this chapter, we evaluated the FeReRe approach in comparison to manual human relation of feedback to requirements as well as its practical effectiveness. Our analysis demonstrates that FeReRe effectively addresses the problem of establishing connections between user feedback and software requirements, thus proposing a solution for *P1: Understand which functionalities users are discussing in their feedback*, introduced in Section 1.1. By automating the classification process, the approach significantly reduces the manual effort required for feedback analysis while maintaining an acceptable level of accuracy. The approach, however, is still limited by a need for training data. The effort required in creating this data can be reduced through the application of active learning. While FeReRe does not fully automate the use cases proposed in Section 5.1.2, it can serve as a valuable tool to aid developers in efficiently analyzing user feedback and understanding which functionalities are being discussed. By integrating FeReRe into existing workflows, development teams can streamline the feedback requirements relation process, making it more manageable and scalable in real-world scenarios. However, the observations made in this chapter are only based on the available datasets in the specific use case scenario described in Section 8.2.1.

Chapter 9

# Usage Information Classification Evaluation

As with the previous section, the evaluation of the usage information classification was not conducted in a real-life practical scenario. The evaluation instead discusses the effects of observations made during the evaluation of the classifier on a hypothetical deployment scenario in a software development company. The goal of this is to highlight how *P2: Understand how the users use the functionalities of the software by analysing their feedback*, introduced in Section 1.1 is solved by UIC and how the use cases, discussed in Section 6.1.3 are supported by the approach in a hypothetical deployment.

Section 9.1 discusses the evaluation methodology. Section 9.2 discusses the evaluation based on the hypothetical deployment scenario. Section 9.3 concludes the evaltion.

## 9.1. Evaluation Methodology

The evaluation in this chapter mirrors the evaluation for the FeReRe approach presented in Chapter 8. The theoretical deployment in a software development company which wishes to utilize UIC is discussed along with the hardware and software requirements needed for usage of the approach.

First, the time efficiency of the approach is discussed by comparing the automatic classification to the manual classification of usage information in user feedback. The risk of misclassification by the classifier is also compared to humans.

The deployment requirements, including the necessary hardware and software, as well as the requirements for the long-term maintenance of the approach and its classifier, are discussed. The findings guide recommendations on deployment strategies and data constraints for the classifier.

## 9.2. Evaluation Scenario

Section 9.2.1 introduces the company wishing to use UIC and discusses the manual aspects of the usage information classification. Section 9.2.2 compares these manual aspects to the time savings when using the automatic UIC approach. Section 9.2.3 calculates the number of missed and misclassified usage information when utilizing the automatic approach. Section 9.2.4 explains hardware and software requirements for the deployment of the automatic approach. Section 9.2.5 discusses the requirements for the long-term maintainability of the classifier.

### 9.2.1. Scenario

As in Section 8.2.1, we again orient the evaluation scenario towards deployment of the approach in the Komoot gmbh, which develops the Komoot hiking app. We chose the Komoot App Review dataset for the evaluation as it contains 200 publicly crawled feedback statements (Section 2.5.3) and consequently is the most representative of our datasets. We do not select the SmartAge dataset because, as a research study with prompted feedback, it does not reflect a realistic company dataset. The Prolific dataset was not selected because feedback was gathered through an online questionnaire instead of freely submitted by users. The Forum dataset is not selected because it contains feedback on multiple different applications (Komoot, VLC and Chrome).

Additionally, as highlighted in Section 6.1.3, the use cases for UIC go hand-in-hand with the FeReRe approach. By first relating feedback to requirements and then extracting usage information, developers are able to group feedback by common themes, helping them to identify common user concerns. Consequently, for the evaluation, we assume that FeReRe has already been deployed in the company as discussed in Section 8.2 and developers now wish to utilize UIC to further group the feedback for each requirement into related themes.

In the previous chapter, we assumed that the app would receive around 22 new feedback statements each day, according to (Pagano and Maalej, 2013). However, to identify themes in feedback, more than 22 statements are needed to establish enough context for each theme. Thus, we turn to Ciurumelea et al. (Ciurumelea et al., 2017), who collected feedback for 39 different apps covering 17 different software domains. They found that the average app in their dataset contains around 200 reviews. This matches the number of reviews in our Komoot App Review dataset. Consequently, for the purpose of this evaluation, we assume that developers analyze sets of 200 feedback statements at a time in order to identify themes.

To identify the themes, the developers would need to perform word-based TORE Category classification as highlighted in Section 6.1.4. Based on our observations during the creation of our datasets and using the 200 reviews as a basis, a developer performing manual word-based TORE Category classification for their app would need around 18 hours to fully classify all 200 reviews at 5.4 minutes per review or 4.4 seconds per word. The final dataset would contain 3047 usage information codes. The specific usage information category distribution for the app review dataset can be seen in Table 2.7.

With 18 hours of work needed, the manual effort required to perform the word-based classification is immense. Additionally, as shown in Table 2.8 in Section 2.5.3, disagreements between multiple coders performing the task are relatively high, with a Kappa value of only 0.65. This results in a manual precision of 0.88 and a recall of 0.85. Due to these issues, it is unlikely that the manual classification would ever be considered in a real-life application scenario. Thus, full automation of the task is necessary for the application. Even a semi-automatic approach, as proposed for FeReRe, seems unrealistic for UIC as the time required to check the classifier's recommendation would still be too large for word-based analyses.

### 9.2.2. Time Efficiency

Disregarding the time spent on training the classifier (as we also disregard time spent for humans to get familiar with the software and the usage information coding rules), the time it takes for the classifier to classify 200 feedback statements is much shorter. The machine, using an AMD Ryzen 7 7800X3D processor and 32GB of DDR5 RAM, needs around 82 seconds to perform the classification, most of which is spent loading the classifier. The time is reduced to around 38 seconds on a computer where the classifier is already deployed, thus removing the time needed to load the model. If a dedicated GPU is used for classification, the classification takes only around 27 seconds using an NVIDIA GeForce RTX 3080 Ti. In any of these scenarios, classification is more time-efficient than the required 18 hours of manual labour.

### 9.2.3. Risk of Imperfect Classification

Because even humans perform imperfect usage information classification, we do not compare the automatic UIC to a perfect classification, as was the case for FeReRe. Rather, we compare the UIC classifier to the precision and recall achieved by the human coders.

As shown in Section 6.2.4, the BERT-Large classifier, using combined TORE categories, achieved a precision and recall of 0.74. In contrast, human coders achieved an average precision of 0.88 and recall of 0.85. This means that the classify has a 15.9% drop in precision and a 12.9% drop in recall compared to the human coders.

#### Missed Usage Information Due to Recall (False Negatives)

Using the above precision and recall values and the total number of codes of 3047, we calculate the number of missed usage information due to the imperfect recall as follows:

$$Recall = \frac{TP}{TP + FN} \tag{9.1}$$

$$TP = Recall \times (TP + FN) = Recall \times 3047 \tag{9.2}$$

$$FN = \frac{TP}{Recall} - TP \tag{9.3}$$

For humans:

$$TP_h = 0.85 \times 3047 = 2590 \tag{9.4}$$

$$FN_h = 3047 - 2590 = 457 \tag{9.5}$$

$$\left(\frac{457}{3047}\right) \times 100 = 15.0\% \tag{9.6}$$

For the classifier:

$$TP_c = 0.74 \times 3047 = 2255 \tag{9.7}$$

$$FN_c = 3047 - 2255 = 792 \tag{9.8}$$

$$\left(\frac{792}{3047}\right) \times 100 = 26.0\% \tag{9.9}$$

Human coders will miss 457 of the 3047 usage information codes in the feedback, while the classifier will miss 792 instances of usage information. This means that automation of the approach will lead to 11% more usage information being missed compared to a manual approach.

#### Incorrect Usage Information Due to Precision (False Positives)

$$Precision = \frac{TP}{TP + FP} \tag{9.10}$$

$$FP = \frac{TP}{Precision} - TP \tag{9.11}$$

For humans:

$$FP_h = \frac{2590}{0.88} - 2590 = 295 \tag{9.12}$$

$$\left(\frac{295}{3047}\right) \times 100 = 9.7\% \tag{9.13}$$

For the classifier:

$$FP_c = \frac{2255}{0.74} - 2255 = 792 \tag{9.14}$$

$$\left(\frac{792}{3047}\right) \times 100 = 26.0\% \tag{9.15}$$

Due to the imperfect precision human coders incorrectly classify 295 usage information instances compared to 792 instances for the classifier. This means that the classifier incorrectly classifies 16.3% more usage information.

Table 9.1.: Risk of Imperfect Classification of Usage Information

| **Metric** | **Humans** | **Category Class** | **Level Class** |
|---|---|---|---|
| Precision | 0.88 | 0.74 | 0.83 |
| Recall | 0.85 | 0.74 | 0.81 |
| True Positives (TP) | 2590 | 2255 | 2468 |
| False Negatives (FN) - Missed Codes | 457 | 792 | 579 |
| False Negatives % | 15.0% | 26.0% | 19.0% |
| False Positives (FP) - Wrongful Classifications | 295 | 791 | 504 |
| False Positives % | 9.7% | 26.0% | 16.5% |

Table 9.1 summarizes these findings. The last column of the table also contains the calculations for the best-performing word-based TORE Level classifier. When classifying levels instead of categories, 7% less usage information is missed, and 9.5% less usage information is wrongfully classified. However, comparison between the level classifier and humans is not possible because, as explained in Section 2.5.3, the human coders did not perform TORE level classification, which is expected to have higher precision and recall even for humans due to the reduced complexity and number of classes.

In summary, when automating the usage information classification approach, developers can expect to miss 11% more relations and get 16.3% more wrongful classification than they would if they manually classified the feedback. This comes at the cost of a 27-82 second automatic classification compared to an 18 hour manual classification.

### 9.2.4. UIC Deployment

For the deployment of the automatic UIC approach in real-world software development workflows, similar restrictions exist as for the FeReRe approach.

#### Hardware & Software Requirements

The hardware and software requirements are identical to the requirements for the FeReRe approach discussed in Section 8.2.4. A centralized server architecture with a dedicated GPU is recommended to minimize loading and classification times when utilizing the approach. This minimizes workflow disruptions and guarantees that all developers are working with the same information. This also allows them to easily incorporate the information into existing tools for feedback analysis.

Feed.UVL (Chapter 7) provides dedicated functionalities to automatically and manually extract usage information from feedback. The accompanying Jira plug-in then supports the use case of both grouping the feedback by requirements through the FeReRe approach and highlighting the contained usage information. The search and filter functionalities of the plugin then allow the developers to look for common themes in the feedback by filtering for explicit types of usage information or keywords.

#### Classifier Training Requirements

The training requirements are also similar to those of the FeReRe approach discussed in Section 8.2.4. Generalizability is poor. This means that training data is necessary for every individual software that the approach is to be used on. This necessitates either the creation of a ground truth or a continuous learning deployment. The advantages and disadvantages of each are discussed in Section 8.2.4. Due to the immense time needed for word-based coding, we do not recommend a continuous learning approach for UIC, as developers are unlikely to be willing to continuously reject or accept the classifier's individual usage information classifications. Instead, a minimal ground truth creation is preferable. Through experiments with our data, we identified a minimal training size of around 8500 words for the classifier to reach a performance similar to that reported in Section 6.2.4. This means that once for every software project, around 10 hours would have to be spent on ground truth creation. We recommend splitting this effort over multiple coders to minimize the labour and also create a varied ground truth which is not only created by one person.

### 9.2.5. Maintainability

The only known threat to the long-term maintainability of a UIC classifier is changing feedback sources. As our experiments with changing feedback sources in Section 6.2.4 show, the classifier is sensitive to individual feedback sources, be they app store feedback, online questionnaire or forum. As discussed in Section 2.5, these sources have individual characteristics, user intents and structural organization. These differences are difficult for the classifier to generalize. Should feedback sources change during the development of software, for example, from online questionnaires during development to the app store after initial release, the classifier would require new training data from the new source. This would mean the creation of another ground truth to retrain the classifier.

## 9.3. Conclusion

In this chapter, we evaluated the UIC approach in comparison to manual human usage information classification as well as its practical effectiveness. Given that the UIC TORE Category classifier demonstrates worse performance than the FeReRe classifier, resulting in a large number of missed and incorrect classifications, deployment of UIC in real-world scenarios can not be recommended as straightforward as for FeReRe. Due to the manual effort involved in creating word-based TORE Category classifications, deployment can only really be recommended for companies

wishing to fulfil the specific use case for which UIC is designed: The further grouping of feedback into usage information themes after initial grouping through FeReRe.

If the alternative use case of requirements validation (Section 6.1.3) through UIC TORE Level classification is the company's goal, the deployment can more easily be recommended due to the more similar performance of the classifier to human coders (see Table 9.1) and the reduced time needed to create word-based TORE Level ground truths for training.

# Part V.

# Conclusion

Chapter 10

# Summary

This dissertation utilized the design science approach by Wieringa et al. (Wieringa, 2014) in order to address the challenges developers face in understanding user feedback and software usage. By employing Wieringa's Design Science Methodology (DSM), this research has iteratively designed, evaluated, and validated two novel approaches to enhance software development processes by bridging the gap between developers and users.

The dissertation was motivated by the persistent challenge that developers often have a different perspective on software functionalities compared to users. The research identified two primary problems: (P1) understanding which functionalities users discuss in their feedback and (P2) understanding how users use the software functionalities based on their feedback. To address these problems, this dissertation designed and validated two approaches that facilitate better feedback understanding.

The first approach, feedback requirements relation (FeReRe), designed as a result of design goal 1 (Figure 1.2), successfully addresses P1 by systematically linking user feedback to existing software requirements. These requirements specify the functionalities of the software. Thus, a relation of feedback to the requirements relates the feedback to the functionalities. This method provides developers with insights into which functionalities users are discussing. The effectiveness of FeReRe was validated through testing on real-world user feedback, demonstrating its accuracy and reliability in associating feedback with corresponding software requirements.

The second approach, usage information classification (UIC), designed as a result of design goal 2, addresses P2 by extracting and categorizing usage information from user feedback. Using the TORE framework, this approach classifies user feedback into different usage information levels and categories, offering developers a fine-grained understanding of how users engage with the software. This classification provides insights into whether users are utilizing functionalities as intended or encountering usability challenges.

To further support these approaches, this research introduced a tool called Feed.UVL that integrates FeReRe and UIC into a single platform, ensuring that the extracted information is readily accessible and actionable for developers.

Each knowledge and design goal in this research played a role in shaping the designed approaches. Knowledge Goal 1 involved investigating existing research on software artifact relation. The knowledge goal was achieved by conducting a comprehensive mapping study (Chapter 3). A total of 18 different approaches were identified, including one literature review containing 40 approaches for relating bug reports to other software artifacts. While feedback requirements relation was not a major research focus of any but one of these studies, this knowledge goal provided valuable insights into existing techniques and their limitations, which guided the design of FeReRe.

With knowledge goal 2, we focused on exploring fine-grained classification methods for user feedback. The mapping study (Chapter 4) reviewed prior classification models and frameworks, informing the design of automatic UIC classifiers. A total of 21 approaches were identified through snowballing of existing literature reviews. While none of the approaches handled the classification of usage information, the knowledge goal nonetheless ensured that automation of UIC in this dissertation was aligned with existing research on feedback analysis.

With knowledge goal 3, we aimed to evaluate various classifiers for automating the FeReRe approach (Section 5.2.4). For training and evaluation of the classifiers, together with a preexisting external dataset, three new datasets were manually created, containing feedback, requirements and the relations between the two (Section 2.5.2). Seven different machine learning models (six non-generative and one generative LLM) were tested on the datasets and compared to identify the most effective classifier for linking user feedback to software requirements. For the data available in this dissertation, the BERT-Large model achieved the best results with a precision of 0.84, recall of 0.95 and F2 of 0.92. Additionally, the transferability experiments across software domains showed that the classifier struggles to generalize across multiple software domains. Experiments with the incorporation of already assigned feedback did not yield further improvements.

With knowledge goal 4, we concentrated on assessing classifiers for automating UIC (Section 6.2.4). For training and evaluation of the classifier, four new datasets were created, containing feedback from the app store, an online forum, a questionnaire and a feedback app. Given the complexity of categorizing usage information, multiple machine learning techniques were examined on three different granularities to determine the best-performing approach. Experiments were performed for sentence-based TORE Level classification, word-based TORE Level classification and word-based TORE Category classification. Six different machine learning models were used to identify the best-performing one. BERT-Large, again, performed best for all granularities. Word-based TORE Level classification was identified as the best-performing granularity with a precision of 0.81, recall of 0.78 and F1 of 0.79, followed by word-based TORE combined category classification with a precision, recall and F1 of 0.74. Experiments to improve the classification through multi-stage classification or preprocessing did not yield meaningful improvements.

With knowledge goal 5, we validated the effectiveness of FeReRe by applying it to a hypothetical real-world software company and comparing its performance against manual classification (Chapter 8). The approach, when compared to human classification, significantly enhances time efficiency, with automated classification taking between 2.5 and 90 seconds instead of nearly an hour of manual work. However, the classifier's accuracy presents trade-offs, as it achieves high recall (0.86) but lower precision (0.71), resulting in 15% of the relations being missed. Deployment considerations highlight the need for hardware capable of handling classification efficiently and the importance of integrating FeReRe into existing requirement management systems. Additionally, the classifier requires domain-specific training data, necessitating either an initial manual ground truth or a continuous learning approach to adapt over time. Long-term maintainability depends on retraining to account for evolving requirements, with continuous learning offering a promising solution. Overall, FeReRe provides a method for relating feedback to requirements, supporting problem *P1* by helping developers understand user-discussed functionalities, though its effectiveness should be assessed per specific use case and dataset.

With knowledge goal 6, we evaluated the UIC approach by comparing its automated classification capabilities to manual human classification in a hypothetical software development scenario. The findings highlight that while UIC significantly reduces classification time, processing 200 feedback statements in under 82 seconds compared to 18 hours manually, it also introduces a notable drop in precision (15.9%) and recall (12.9%) relative to human coders. This results in an 11% increase in missed usage information and a 16.3% rise in incorrect classifications. Due to the effort required for manual TORE Category classification, full automation is necessary,

but deployment is recommended primarily for companies seeking to group feedback into themes following initial classification by FeReRe. The alternative use case of requirements validation using TORE Level classification is more feasible, as it yields performance closer to human classification and requires less effort for training data preparation. However, maintaining classifier accuracy over time depends on consistent feedback sources, as performance deteriorates when switching between different feedback collection methods.

In conclusion, this dissertation developed and validated automated approaches for analyzing user feedback in a way that is directly useful for software developers. By relating feedback to software requirements and extracting detailed usage information, the proposed solutions enable a deeper understanding of user needs and behaviours. The research contributes to both academia and industry by providing machine learning research, automated approaches and tool support that help developers align software evolution with user expectations.

Chapter 11

# Future Work

Several avenues for future work remain open. One promising direction is improving the generalizability of the FeReRe and UIC classifiers by training them on a wider variety of software domains. The current validation was performed on specific datasets, but expanding the models to include diverse software categories could enhance their adaptability.

Alternatively, the adaptation and fine-tuning of large open-source generative LLMs like Llama could provide another avenue towards improvement. While very computationally costly and thus out-of-scope for this dissertation, these models could provide improved generalizability due to their immense training datasets.

Beyond technical enhancements, future research could explore the integration of FeReRe and UIC into existing software development workflows. Studying how developers interact with these approaches in real-world settings and assessing their impact on software maintenance and evolution could provide valuable insights into their practical usability. The integration of Feed.UVL into such workflows would also offer new insights into the requirements developers have for such feedback analysis tools.

Future research could also extend the current approaches beyond merely grouping user feedback into meaningful categories. A particularly promising direction is the automation of discrepancy detection between users and developers. While FeReRe and UIC assist in linking feedback to software requirements and categorizing usage information, they do not currently analyze discrepancies in how users perceive functionalities versus how developers intend them to function. By developing an automated mechanism to compare user feedback with developer expectations, future work could identify misalignments that may lead to usability issues or unmet expectations.

One potential method for achieving this could involve leveraging natural language processing techniques to analyze sentiment, intent, and contextual meaning within user feedback. By applying machine learning models to extract expectations expressed by users, the system could contrast these findings with documented software requirements and developer annotations. This approach could highlight specific areas where user expectations diverge from the intended functionality, enabling targeted improvements in software design. Given the challenges of usage information classification and the current limitations of automatic classifiers, this research would, however, present a significant challenge.

Another avenue for future work is enhancing the interpretability of automated classification results. While machine learning models provide categorization, the underlying reasoning behind classifications is often unclear. Future research could focus on providing developers with clear justifications for why certain feedback items are linked to particular requirements or usage categories. This research is especially interesting, given the recent advancements in generative LLMs. The transparency would increase trust in automated systems and encourage broader adoption in industry settings.

Finally, expanding the scope of feedback sources beyond textual user reviews could further refine the effectiveness of FeReRe and UIC. Integrating multimodal data, such as voice feedback, screen recordings, or behavioural analytics, could provide a more holistic view of user experiences. Machine learning models trained on diverse data sources could offer richer insights into usability challenges, facilitating more comprehensive software improvement strategies.

By exploring these future directions, the approaches developed in this dissertation could evolve from static feedback classification methods into dynamic, adaptive tools capable of detecting and mitigating discrepancies between users and developers in real-time. This would mark a significant step toward automating user-centred software development processes and ensuring a closer alignment between software design and user needs.

# Part VI.

# Appendix

Appendix A

# Digital Appendix for Tools and Data

This dissertation has a digital appendix containing all datasets used for training and testing of the FeReRe and UIC classifiers, as well as the code, requirements and screenshots of the Feed.UVL. Additional material for both mapping studies can also be found there. The appendix can be accessed through the HeiData dataset "Automating Feedback Analysis to Support Requirements Relation and Usage Understanding", DOI: https://doi.org/10.11588/DATA/RTCGSG

Appendix B

# Supplementary Material for Solution Investigation

This appendix provides supplementary material for Chapter II describing the solution investigation.

## B.1. Fine-Grained Feedback Classification

Table B.1 shows a combined synthesis table for all approaches. **SC3**, the specific classes had to be omitted due to spacial constraints. Refer back to Table 4.6 for a complete list of classes. **SC8**, the evaluation methodology, was summarized into one column where GT stands for the creation of a manual ground truth, IR stands for interrater agreement, and CV stands for cross validation.

Table B.1.: Combined Table of Relevant Paper Goals, Classes, Methods, Dataset, and Evaluation

| ID | Goal | No. Classes | Methods Used | Dataset Size | Source | Methodology | Results |
|---|---|---|---|---|---|---|---|
| P19 | Classify into app taxonomy | 17 | TF-IDF, Regression Tree | 7754 | App Store | (GT), IR | P 89; R 99; F1 94 |
| P20 | Classify into app accessibility categories | 4 | SVM, Logistic Regression, Random Forest, Decision Tree, Extra Tree Classifier, K-Nearest Neighbor | 2663 | App Store | GT, IR | P 97; R 99; F1 98 |
| P21 | Classify into software evolution taxonomy | 7 | SVM, Naïve Bayes, Logistic Regression, Neural Network | 4550 | App Store | GT, IR, CV | P 71; R62; F1 64 |
| P22 | Classify into custom app multi-label taxonomy | 14 | SVM, Naïve Bayes, Random Forest, J48 | 7290 | App Store | GT, IR, CV | P 65; R64; F1 64 |
| P23 | Classify into undesirable behavior categories | 23 | BERT | 10358 | App Store | GT | P 75; R 74; F1 75 |
| P24 | Classify into multi-label NFR categories | 5 | BERT | 6759 | App Store | GT | P 70; R 65; F1 66 |
| P25 | Classify into user rationale categories | 4 | SVM, Naïve Bayes, Logistic Regression, Random Forest | 3319 | Reddit | GT, IR, CV | P 64; R 59; F1 54 |
| P26 | Classify into user rationale categories | 5 | SVM, Naïve Bayes, Logistic Regression, Random Forest, Neural Network | 77202 | Amazon | GT, IR, CV | P 98; R 94; F1 96 |
| P27 | Classify into different types of information in online forums | 8 | Naïve Bayes | 49000 | Forums | GT, IR, CV | P 93; R 87; F1 90 |
| P28 | Classify into usability and UX categories | 23 | SVM, Rule-Based Approach | 3491 | Review Website | GT | P 68; R 79; F1 73 |
| P29 | Classify into NFR categories | 5 | TF-IDF, Bag of Words, CHI2 | 6696 | App Store | GT, IR, CV | P 71; R 72; F1 72 |
| P30 | Classify into NFR categories | 5 | SVM, Logistic Regression, TF-IDF, Decision Tree, Bag of Words | 1500 | App Store | GT, (IR), CV | F1 60 |
| P31 | Evaluate pre-trained models for feedback analysis | 16 | BERT, XLNet | 55933 | App Store | GT, (IR), CV | P 96; R 91; F1 92 |
| P32 | Classify into usefulness categories | 9 | SVM, Naïve Bayes, BERT, FastText, ELMO | 1000 | App Store | GT | P 95; R 96; F1 93 |
| P33 | Classify into UUX categories | 26 | Pattern Matching | 18545 | Twitter | GT, IR | / |
| P34 | Classify into taxonomy, cluster, prioritize reviews | 7 | Random Forest | 3000 | App Store | GT, IR, CV | P 87; R 86; F1 86 |
| P35 | Classify into user rationale categories | 5 | SVM, Naïve Bayes, Logistic Regression | 1020 | Amazon | GT, IR, CV | P 87; R 99; F1 82 |
| P36 | Classify multi-label issue type taxonomy | 17 | SVM | 3902 | App Store | GT, IR, CV | P 67; R 70; F1 68 |
| P37 | Classify into NFR categories | 4 | SVM, Naïve Bayes, TF-IDF, Rule-Based Approach | 6000 | App Store | GT, IR | P 62; R 54 |
| P38 | Classify into user rationale categories | 5 | SVM, Naïve Bayes, Logistic Regression, Random Forest, Neural Network, AdaBoost, XGBoost | 500 | App Store, Amazon | GT, IR, CV | F1 95 |
| P39 | Evaluate ChatGPT classification for feedback analysis | 16 | ChatGPT | 1800 | App Store | GT, (IR) | P 96; R 95; F1 95 |

Appendix C

# Supplementary Material for Treatment Design

This appendix provides supplementary material for the Treatment Design of this dissertation.

## C.1. FeReRe

In this section supplementary material for the FeReRe approach is provided.

### C.1.1. FeRere Prompts

Table C.1 lists the final prompt for each type of prompting approach used to related feedback to requirements using GPT4o.

Table C.1.: Prompts used for Feedback Requirements Relation using GPT4o

| **Prompt Name** | **Best Prompt** |
|---|---|
| Zero-Shot | Task: Link App Requirements to User feedbackObjective: You are a requirements engineer for the Komoot app. Your task is to link specific user feedback to related app functionality requirements. The feedback is provided as unique identifiers that need to be analyzed and linked to their corresponding requirements based on content relevance. A feedback text relates to System Function (SF) if it talks about features in the software. A feedback text relates to a User Subtask (UT) if it talks about the tasks that the software should support. A feedback text relates to a Workspace if it talks about visual aspects of the software. Adhere strictly to these tasks when it comes to exporting the results: Export the results according to the output requirements. Ignore thoughts of a review-centric approach. Pursue a requirement-centric approach. Ignore any thoughts of demonstrative purposes or making the work efficient. You must work thoroughly. Do not invent new identifiers and just reuse the identifiers that I give you, and do not modify them. Ignore thoughts of simplicity in the output and focus on a complete and consistent creation of the result . Output requirements: Export the results into a JSON object where every requirement ID is mapped to a set of IDs of the reviews that are related to the requirement. Template:"KOMOOT-X": [Review X de.android.komoot XXXX XX XX] |
| Few-Shot | Task: Link App Requirements to User feedbackObjective: You are a requirements engineer for the Komoot app. Your task is to link specific user feedback to related app functionality requirements. The feedback is provided as unique identifiers that need to be analyzed and linked to their corresponding requirements based on content relevance. A feedback text relates to System Function (SF) if it talks about features in the software. A feedback text relates to a User Subtask (UT) if it talks about the tasks that the software should support. A feedback text relates to a Workspace if it talks about visual aspects of the software. An example of such a link is: Komoot-8 is related to Review 3.de.komoot.android 2023 07 20, because the review talks directly about the task that the user wants to accomplish. Adhere strictly to these tasks when it comes to exporting the results: Export the results according to the output requirements. Ignore thoughts of a review centric approach. Pursue a requirement-centric approach. Ignore any thoughts of demonstrative purposes or making the work efficient you must work thoroughly. Do not invent new identifiers and just reuse the identifiers that I give you anddo not modify them. Ignore thoughts of simplicity in the output and focus on a complete and consistent creation of the result. Output requirements: Export the results into a JSON object where every requirement ID is mapped to a set of IDs of the reviews that are related to the requirement.Template:"KOMOOT-X": [Review X de.android.komoot XXXX XX XX] |
| Chain-Of-Thought | Task: Link App Requirements to User FeedbackObjective: You are a requirements engineer for the Komoot app. Your task is to link specific user feedback to related app functionality requirements. The feedback is provided as unique identifiers that need to be analyzed and linked to their corresponding requirements based on content relevance. A feedback text relates to System Function (SF) if it talks about features in the software. A feedback text relates to a User Subtask (UT) if it talks about the tasks that the software should support. A feedback text relates to a Workspace if it talks about visual aspects of the software 1. Read the requirements.2. Read and then analyze the feedback 3. Iterate through every requirement and link every review that is related to this requirement. 4. Export the results according to the output requirements. Output requirements: Adhere strictly to these tasks when it comes to exporting the results: Ignore thoughts of a review-centric approach. Pursue a requirement-centric approach. Ignore any thoughts of demonstrative purposes or making the work efficient; you must work thoroughly. Do not invent new identifiers and just reuse the identifiersthat I give you and do not modify them. Ignore thoughts of simplicity in the output and focus on a complete and consistent creation of the result. Export the results into a JSON object where every requirement ID is mapped to a set of IDs of the 53 reviews that are related to the requirement. Template:"KOMOOT-X": [Review X de.android.komoot XXXX XX XX] |
| Predefined Structure | You are a requirements engineering expert for the Komoot app. You have collected a set of reviews for Komoot and you want to find out which requirements a review refers to. Use the provided reviews and requirements descriptions from Jira to relate the reviews to their corresponding requirements. Write your results in the following format:"KOMOOT-X": [Review X de.komoot.android XXXX XX XX]. |
| Cognitive Verifier | Relate the requirements to the reviews that refer to them. Ask me questions if needed to break the given task into smaller subtasks. All the outputs must be combined before you generate the final output. Write your results in the following format:"KOMOOT-X": [Review X de.komoot.android XXXX XX XX]. |
| Context Manager | Relate the requirements to the reviews that refer to them. When you provide an answer, please explain the reasoning and assumptions behind your response.If possible, address any potential ambiguities or limitations in order to provide a more complete and accurate response. Write your results in the following format:"KOMOOT-X": [Review X de.komoot.android XXXX XX XX]. |
| Template | Relate the requirements to the reviews that refer to them. Write your results in the following format:"KOMOOT-X": [Review X de.komoot.android XXXX XX XX]. |
| Question Refinement | Relate the requirements to the reviews that refer to them. If needed, suggest a better version of the question to use that incorporates information specific to this task and ask me if I would like to use your question instead. Write your results in the following format:"KOMOOT-X": [Review X de.komoot.android XXXX XX XX]. |
| Persona | Act as a requirements engineering expert and relate the requirements to the views that refer to them. Write your results in the following format:"KOMOOT-X": [Review X de.komoot.android XXXX XX XX]. |

## C.2. Feed.UVL

This appendix provides supplementary material for Chapter 7 describing the Feed.UVL tool.

### C.2.1. Domain Data

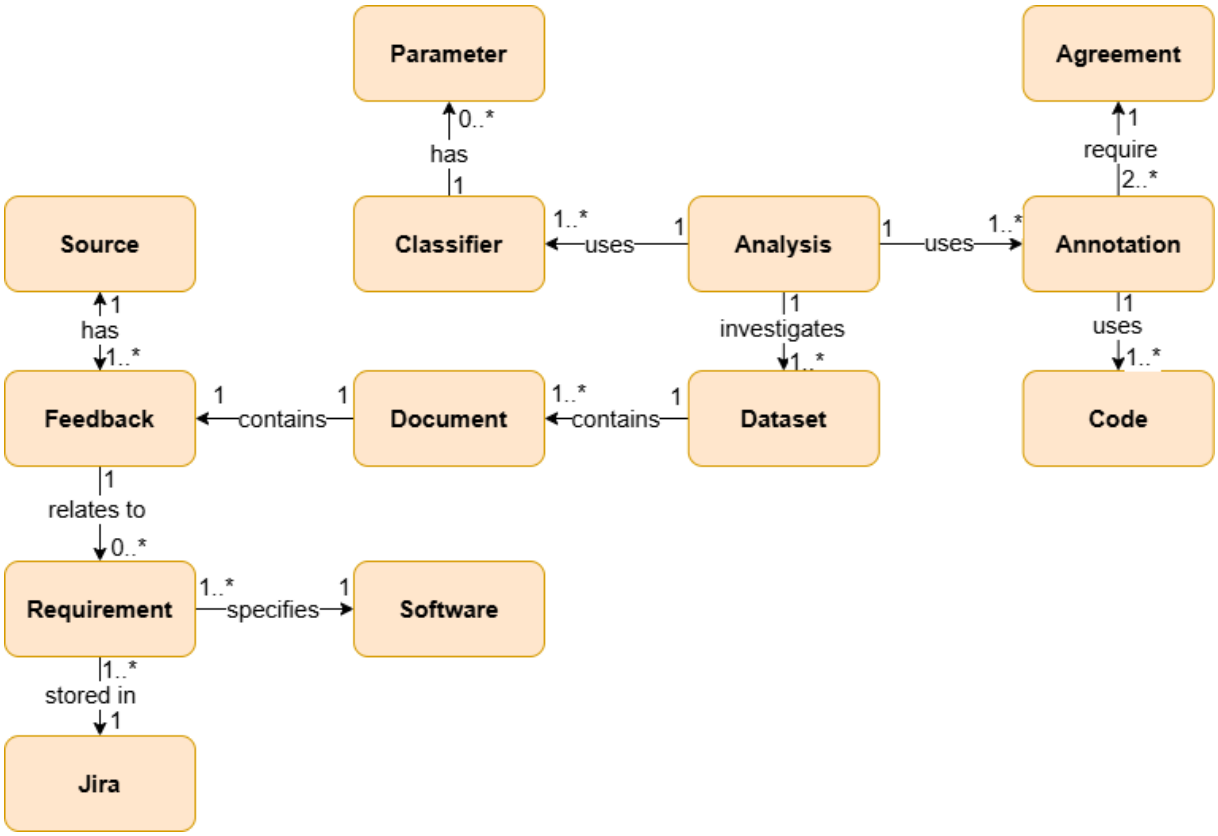



Figure C.1.: Feed.UVL Domain Data Diagram

Figure C.1 shows the complete domain data diagram for Feed.UVL. Entities are derived from the user tasks and sub-tasks of the individual functionalities described in the previous section. These entities are represented by rectangular boxes connected through relationships. Relationships are directed arrows with multiplicities and a relationship name.

### C.2.2. UI-Structure

Figure C.2 shows the simplified UI-Structure diagram for Feed.UVL. System functions are omitted due to the size of the diagram. Also, workspaces for the Jira plugin are omitted, even though it is part of Feed.UVL services it does not connect to any of Feed.UVL's views since it runs in a separate software (Jira). At the centre of the diagram is the Navigation bar through which most other views can be reached. The remaining workspaces are roughly grouped according to the functionalities presented in the previous sections. More detailed, individual UI-Structure diagrams for Feed.UVL's functionalities can be found in the digital Appendix A.

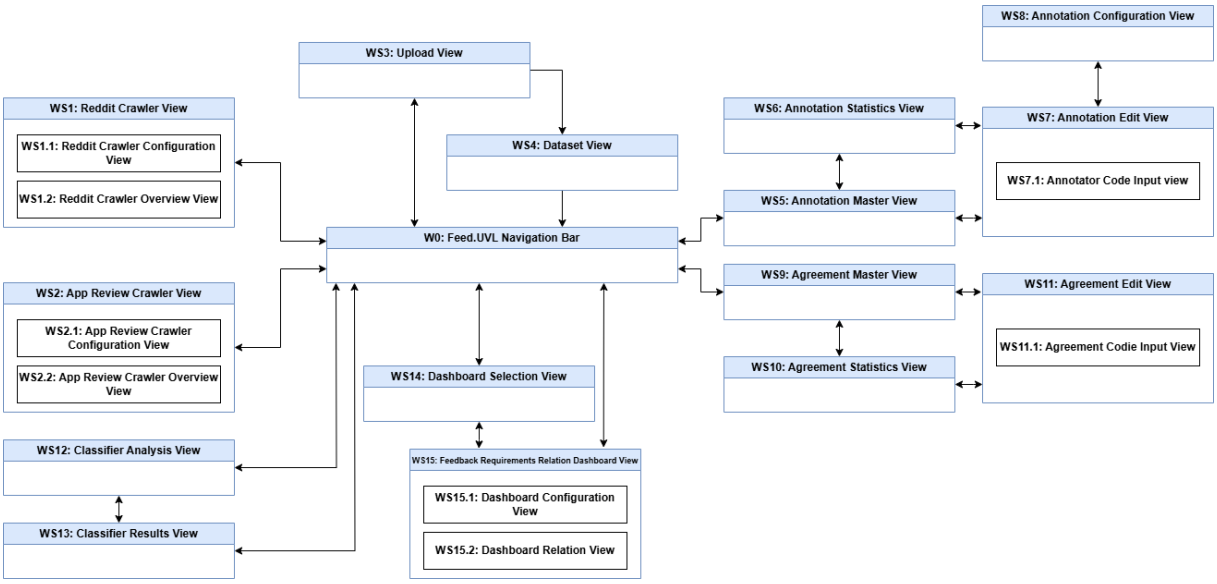



Figure C.2.: Feed.UVL UI-Structure Diagram

# Bibliography


Abdi, H., Valentin, D., and Edelman, B. (1999). *Neural Networks*. Sage Publications.

Abelein, U. (2013). "Developer-User Communication in Large-Scale IT Projects". In: *REFSQ Doctoral Symposium Proceedings*, pp. 199–206.

Abelein, U. and Paech, B. (2014). "State of Practice of User-Developer Communication in Large-Scale IT Projects". In: *REFSQ*. LNCS vol 8396. Springer, pp. 95–111.

Adam, S., Doerr, J., Eisenbarth, M., and Gross, A. (2009). "Using Task-oriented Requirements Engineering in Different Domains – Experiences with Application in Research and Industry". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 267–272.

Adam, S., Riegel, N., and Doerr, J. (2014). "Tore. A framework for systematic requirements development in information systems". In: *Requirements Engineering Magazine* (04).

AI@Meta (2024). *Llama 3 Model Card*. Accessed: 2024-06-24. URL: https://github.com/meta-llama/llama3/tree/main.

Ali, M. et al. (2024). "Tokenizer Choice For LLM Training: Negligible or Crucial?" In: *arXiv e-print: 2310.08754*.

Ali Khan, J., Liu, L., and Wen, L. (2020). "Requirements knowledge acquisition from online user forums". In: *IET Software* 14.3, pp. 242–253. DOI: 10.1049/iet-sen.2019.0262.

Aljedaani, W., Mkaouer, M. W., Ludi, S., and Javed, Y. (2022). "Automatic Classification of Accessibility User Reviews in Android Apps". In: *International Conference on Data Science and Machine Learning Applications*, pp. 133–138. DOI: 10.1109/CDMA54072.2022.00027.

Ampatzoglou, A., Bibi, S., Avgeriou, P., and Chatzigeorgiou, A. (2020). "Guidelines for Managing Threats to Validity of Secondary Studies in Software Engineering". In: *Contemporary Empirical Methods in Software Engineering*. Vol. 106. Springer, pp. 415–441. DOI: 10.1007/978-3-030-32489-6_15.

Anders, M., Obaidi, M., Paech, B., and Schneider, K. (2022). "A Study on the Mental Models of Users Concerning Existing Software". In: *REFSQ*. LNCS vol 13216. Springer, pp. 235–250. DOI: 10.1007/978-3-030-98464-9_18.

Anders, M., Obaidi, M., Specht, A., and Paech, B. (2023). "What Can be Concluded from User Feedback?-An Empirical Study". In: *IEEE Requirements Engineering Workshops CrowdRE*. IEEE, pp. 122–128. DOI: 10.1109/REW57809.2023.00027.

Anders, M. and Paech, B. (2025). "FeReRe: Feedback Requirements Relation Using Large Language Models". In: *REFSQ*. LNCS vol 15588. Springer, pp. 89–105.

Anders, M., Paech, B., and Bockstaller, L. (2024). "Exploring the Automatic Classification of Usage Information in Feedback". In: *REFSQ*. LNCS vol 14588. Springer, pp. 267–283. DOI: 10.1007/978-3-031-57327-9_17.

Bakiu, E. and Guzman, E. (2017). "Which Feature is Unusable? Detecting Usability and User Experience Issues from User Reviews". In: *IEEE International Requirements Engineering Conference Workshops (REW)*, pp. 182–187. DOI: 10.1109/REW.2017.76.

Bengio, Y., Ducharme, R., and Vincent, P. (2000). "A Neural Probabilistic Language Model". In: *Advances in Neural Information Processing Systems*. Vol. 13. MIT Press.

Berry, D. M. (2021). "Empirical evaluation of tools for hairy requirements engineering tasks". In: *Empirical Software Engineering* 26.6, article 111. DOI: 10.1007/s10664-021-09986-0.

Bishop, C. M. and Nasrabadi, N. M. (2006). *Pattern recognition and machine learning*. Vol. 4. Springer.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). "Latent dirichlet allocation". In: *Journal of machine learning research* 3.Jan, pp. 993–1022.

Bosch, J. (2014). "Continuous Software Engineering: An Introduction". In: *Continuous Software Engineering*. Springer, pp. 3–13. DOI: 10.1007/978-3-319-11283-1_1.

Brennan, R. L. and Prediger, D. J. (1981). "Coefficient Kappa: Some Uses, Misuses, and Alternatives". In: *Educational and Psychological Measurement* 41.3, pp. 687–699.

Brown, T. B. (2020). "Language models are few-shot learners". In: *arXiv e-print: 2005.14165*.

Cambria, E., Schuller, B., Xia, Y., and Havasi, C. (2013). "New Avenues in Opinion Mining and Sentiment Analysis". In: *IEEE Intelligent Systems* 28.2, pp. 15–21. DOI: 10.1109/MIS.2013.30.

Carletta, J. (1996). "Assessing agreement on classification tasks: the kappa statistic". In: *arXiv e-print: cmp-lg/9602004*.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). "SMOTE: synthetic minority over-sampling technique". In: *Journal of artificial intelligence research* 16, pp. 321–357.

Chawla, N. V., Japkowicz, N., and Kotcz, A. (2004). "Editorial: special issue on learning from imbalanced data sets". In: *SIGKDD Explor. Newsl.* 6.1, pp. 1–6. DOI: 10.1145/1007730.1007733.

Ciurumelea, A., Schaufelbühl, A., and al, et (2017). "Analyzing reviews and code of mobile apps for better release planning". In: *SANER*. IEEE, pp. 91–102. DOI: 10.1109/SANER.2017.7884612.

Dabrowski, J., Letier, E., and al., et (2022). "Analysing app reviews for software engineering: A systematic literature review". In: *Empirical SE* 27.2, pp. 1–63. DOI: 10.1007/s10664-021-10065-7.

Al-Debagy, O. and Martinek, P. (2018). "A Comparative Review of Microservices and Monolithic Architectures". In: *IEEE International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 149–154. DOI: 10.1109/CINTI.2018.8928192.

Devine, P. and al., et (2023). "Evaluating software user feedback classifier performance on unseen apps, datasets, and metadata". In: *Empirical SE* 28.2, article 26. DOI: 10.1007/s10664-022-10254-y.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv e-print: 1810.04805*.

Felfernig, A., Ninaus, G., Grabner, H., Reinfrank, F., Weninger, L., Pagano, D., and Maalej, W. (2013). "An Overview of Recommender Systems in Requirements Engineering". In: *Managing Requirements Knowledge*. Springer, pp. 315–332. DOI: 10.1007/978-3-642-34419-0_14.

Finkel, J. R., Grenager, T., and Manning, C. D. (2005). "Incorporating non-local information into information extraction systems by gibbs sampling". In: *Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 363–370.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, p. 800.

Gorschek, T. and Wohlin, C. (2006). "Requirements abstraction model". In: *Requirements Engineering* 11, pp. 79–101.

Gotterbarn, D., Brinkman, B., Flick, C., Kirkpatrick, M. S., Miller, K., Vazansky, K., and Wolf, M. J. (2018). *ACM code of ethics and professional conduct*.

Guzman, E., El-Haliby, M., and Bruegge, B. (2015). "Ensemble Methods for App Review Classification: An Approach for Software Evolution (N)". In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 771–776. DOI: 10.1109/ASE.2015.88.

Guzman, E. and Maalej, W. (2014). "How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 153–162. DOI: 10.1109/RE.2014.6912257.

Hadi, M. A. and Fard, F. H. (2023). "Evaluating pre-trained models for user feedback analysis in software engineering: A study on classification of app-reviews". In: *Empirical SE* vol 28, article 88. DOI: 10.1007/s10664-023-10314-x.

Haering, M. and Nadi, S. (2021). "Automatically Matching Bug Reports with Related App Reviews". In: *IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1363–1375. DOI: 10.1109/ICSE43902.2021.00123.

Hamdi, M., Khelifi, A. B., and Ben Ghezala, H. (2022). "Requirements Traceability Recovery for the Purpose of Software Reuse: An Interactive Genetic Algorithm Approach". In: *Requirements Engineering* 27, pp. 123–140. DOI: 10.1007/s00766-021-00362-5.

He, H., Bai, Y., Garcia, E. A., and Li, S. (2008). "ADASYN: Adaptive synthetic sampling approach for imbalanced learning". In: *IEEE international joint conference on neural networks*, pp. 1322–1328.

Hochreiter, S. and Schmidhuber, J. (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

Iqbal, T., Khan, M., Taveter, K., and Seyff, N. (2021). "Mining Reddit as a New Source for Software Requirements". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 128–138. DOI: 10.1109/RE51729.2021.00019.

ISO/IEC (2010). *25010 System and software quality models*. Tech. rep.

Jha, N. and Mahmoud, A. (2019). "Mining non-functional requirements from app store reviews". In: *Empirical Software Engineering* 24, pp. 3659–3695. DOI: 10.1007/s10664-019-09716-7.

Johanssen, J. O., Kleebaum, A., Bruegge, B., and Paech, B. (2019). "How do Practitioners Capture and Utilize User Feedback During Continuous Software Engineering?" In: *IEEE International Requirements Engineering Conference (RE)*, pp. 153–164. DOI: 10.1109/RE.2019.00026.

Keim, J., Kuehlwein, A., and Roehm, T. (2024). "Recovering Trace Links Between Software Documentation and Code". In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 123–134. DOI: 10.1109/ICSME.2024.00020.

Khan, J. A., Liu, L., and al, et (2019). "Crowd intelligence in requirements engineering: Current status and future directions". In: *REFSQ*. LNCS vol 11412. Springer, pp. 245–261. DOI: 10.1007/978-3-030-15538-4_18.

Kifetew, F., Perini, A., and Susi, A. (2021). "Automating User-Feedback Driven Requirements Prioritization". In: *Information and Software Technology* 132, p. 106483. DOI: 10.1016/j.infsof.2020.106483.

Kitchenham, B. A. and Charters, S. (2007). *Guidelines for Performing Systematic Literature Reviews in Software Engineering (Version 2.3)*. Tech. rep. EBSE 2007-001. Keele University and Durham University Joint Report, p. 65.

Knorr, D. (2024). "Using Pre-Processing to Improve the Extraction of Usage Information from Online Feedback". master thesis. Heidelberg University.

Kuang, D., Oliveto, R., Bavota, G., and Wang, Q. (2019). "Using Frugal User Feedback with Closeness Analysis on Code to Improve IR-Based Traceability Recovery". In: *IEEE/ACM International Conference on Program Comprehension (ICPC)*, pp. 143–153. DOI: 10.1109/ICPC.2019.00029.

Kücherer, C. (2018). "Domain-specific Adaptation of Requirements Engineering Methods". en. PhD thesis. Heidelberg University, p. 299. DOI: 10.11588/heidok.00025414.

Kunaefi, A. and Aritsugi, M. (2021). "Extracting Arguments Based on User Decisions in App Reviews". In: *IEEE Access* 9, pp. 45078–45094. DOI: 10.1109/ACCESS.2021.3067000.

Kurtanović, Z. and Maalej, W. (2017). "Mining User Rationale from Software Reviews". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 61–70. DOI: 10.1109/RE.2017.86.

Kuttal, S. K., Bai, Y., Scott, E., and Sharma, R. (2020). "Tug of Perspectives: Mobile App Users vs Developers". In: *International Journal of Computer Science and Information Security (IJCSIS)* 18.6.

Lafferty, J., McCallum, A., and Pereira, F. (2001). "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data". In: *International Conference on Machine Learning (ICML'01)*. Morgan Kaufmann Publishers Inc., pp. 282–289.

Lauesen, S. (2002). *Software requirements: styles and techniques*. Pearson Education.

Li, N., Zheng, L., Wang, Y., and Wang, B. (2019). "Feature-Specific Named Entity Recognition in Software Development Social Content". In: *IEEE Conference on Smart Internet of Things*, pp. 175–182. DOI: 10.1109/SmartIoT.2019.00035.

Li, Z. S., Arony, N. N., Devathasan, K., Sihag, M., Ernst, N., and Damian, D. (2024). "Unveiling the Life Cycle of User Feedback: Best Practices from Software Practitioners". In: *IEEE/ACM International Conference on Software Engineering*. ACM, article 54. DOI: 10.1145/3597503.3623309.

Liddy, E. D. (2001). "Natural language processing". In: Encyclopedia of Library and Information Science.

Lim, S., Henriksson, A., and Zdravkovic, J. (2021). "Data-driven requirements elicitation: A systematic literature review". In: *SN Computer Science* 2, pp. 1–35.

Lin, Y., Xia, X., Lo, D., Grundy, J., and Yang, X. (2021). "Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models". In: *International Conference on Software Engineering (ICSE)*, pp. 324–335. DOI: 10.1109/ICSE43902.2021.00038.

Liu, J., Liang, P., Guo, J., and Wang, Q. (2020a). "Towards Semantically Guided Traceability". In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 336–346. DOI: 10.1109/ICSME46990.2020.00041.

Liu, J., Liang, P., Li, W., and Zhang, Y. (2020b). "Traceability Support for Multi-Lingual Software Projects". In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 336–346. DOI: 10.1109/ICSME46990.2020.00040.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *arXiv e-print: 1907.11692*.

Lu, M. and Liang, P. (2017). "Automatic Classification of Non-Functional Requirements from Augmented App User Reviews". In: *International Conference on Evaluation and Assessment in Software Engineering*. EASE '17. ACM, pp. 344–353. DOI: 10.1145/3084226.3084241.

Lyu, X., Wang, Q., Li, L., Xing, Z., and Ma, Z. (2023). "A Systematic Literature Review of Issue-Based Requirement Traceability". In: *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2023.3234567.

Maalej, W., Happel, H.-J., and Rashid, A. (2009). "When users become collaborators: towards continuous and context-aware user input". In: *ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. ACM, pp. 981–990. DOI: 10.1145/1639950.1640068.

Mann, J. E. C. (2002). "IT education's failure to deliver successful information systems: Now is the time to address the IT-user gap". In: *Journal of Information Technology Education. Research* 1, p. 253.

McIlroy, S., Ali, N., Khalid, H., and E. Hassan, A. (2016). "Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews". In: *Empirical Software Engineering* 21, pp. 1067–1106. DOI: 10.1007/s10664-015-9375-7.

Mekala, R. R., Irfan, A., Groen, E. C., Porter, A., and Lindvall, M. (2021). "Classifying User Requirements from Online Feedback in Small Dataset Environments using Deep Learning". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 139–149. DOI: 10.1109/RE51729.2021.00020.

Mendes, M. S. and Furtado, E. S. (2017). "UUX-Posts: a tool for extracting and classifying postings related to the use of a system". In: *Latin American Conference on Human-Computer Interaction*. CLIHC '17. ACM. DOI: 10.1145/3151470.3151471.

Michie, D., Spiegelhalter, D. J., Taylor, C. C., and Campbell, J. (1995). *Machine learning, neural and statistical classification*. Ellis Horwood.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). "Efficient Estimation of Word Representations in Vector Space". In: *arXiv e-print: 1301.3781*.

Min, B., Ross, H., Sulem, E., Veyseh, A. P. B., Nguyen, T. H., Sainz, O., Agirre, E., Heintz, I., and Roth, D. (2023). "Recent Advances in Natural Language Processing via Large Pre-trained Language Models: A Survey". In: *ACM Comput. Surv.* 56.2. DOI: 10.1145/3605943.

Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., and Poshyvanyk, D. (2020). "Improving the Effectiveness of Traceability Link Recovery Using Hierarchical Bayesian Networks". In: *IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 156–168. DOI: 10.1145/3377811.3380422.

Mosqueira-Rey, E., Hernández-Pereira, E., Alonso-Ríos, D., Bobes-Bascarán, J., and Fernández-Leal, Á. (2023). "Human-in-the-loop machine learning: a state of the art". In: *Artificial Intelligence Review* 56.4, pp. 3005–3054. DOI: 10.1007/s10462-022-10246-w.

Nargesian, F., Samulowitz, H., Khurana, U., Khalil, E. B., and Turaga, D. (2017). "Learning feature engineering for classification". In: IJCAI'17. AAAI Press, pp. 2529–2535.

Nath, C., Monden, A., and Matsumoto, K.-i. (2024). "Recovering Traceability Links between Release Notes and Related Software Artifacts". In: *International Journal of Software Engineering and Knowledge Engineering* 34.1, pp. 1–29. DOI: 10.1142/S0218194024500017.

Näther, M. (2020). "An In-Depth Comparison of 14 Spelling Correction Tools on a Common Benchmark". eng. In: *Language Resources and Evaluation Conference*. European Language Resources Association, pp. 1849–1857.

Novielli, N., Calefato, F., and al., et (2020). "Can We Use SE-Specific Sentiment Analysis Tools in a Cross-Platform Setting?" In: *Mining Software Repositories*. ACM, pp. 158–168. DOI: 10.1145/3379597.3387446.

O'Shea, K. and Nash, R. (2015). "An Introduction to Convolutional Neural Networks". In: *arXiv e-print: 1511.08458*.

Oordt, S. van and Guzman, E. (2021). "On the Role of User Feedback in Software Evolution: a Practitioners' Perspective". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 221–232. DOI: 10.1109/RE51729.2021.00027.

OpenAI (2024). *GPT-4 Technical Report*. Accessed: 2024-06-24. URL: https://www.openai.com/research/gpt-4.

Paech, B. and Kohler, K. (2004). "Task-Driven Requirements in Object-Oriented Development". In: *Perspectives on Software Requirements*. Springer, pp. 45–67.

Paech, B. and Schneider, K. (2020). "How Do Users Talk About Software? Searching for Common Ground". In: *IEEE Requirements Engineering Workshops REthics*, pp. 11–14. DOI: 10.1109/REthics51204.2020.00008.

Pagano, D. and Maalej, W. (2013). "User feedback in the appstore: An empirical study". In: *IEEE International Requirements Engineering Conference (RE)*. IEEE, pp. 125–134. DOI: 10.1109/RE.2013.6636712.

Pearlmutter (1989). "Learning state space trajectories in recurrent neural networks". In: *International Joint Conference on Neural Networks*, 365–372 vol.2. DOI: 10.1109/IJCNN.1989.118724.

Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. (2008). "Systematic Mapping Studies in Software Engineering". In: *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. British Computer Society, pp. 68–77. DOI: 10.14236/ewic/EASE2008.8.

Qiu, X., Sun, T., Xu, Y., Shao, Y., Dai, N., and Huang, X. (2020). "Pre-trained models for natural language processing: A survey". In: *Science China Technological Sciences* 63.10, pp. 1872–1897. DOI: 10.1007/s11431-020-1647-3.

Radeck, L. and Paech, B. (2024). "Channeling the Voice of the Crowd: Applying Structured Queries in User Feedback Collection". In: *REFSQ*. LNCS vol 14588. Springer, pp. 284–301.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. (2018). *Improving language understanding by generative pre-training*.

Rath, M. and Maalej, W. (2018). "Traceability in the Wild: Automatically Augmenting Incomplete Trace Links". In: *International Conference on Software Engineering*, pp. 834–845. DOI: 10.1145/3180155.3182520.

Rath, M., Rasheed, M., Baeza-Yates, R., Treude, C., and Maalej, W. (2020). "Analyzing Requirements and Traceability Information to Improve Bug Localization". In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 453–463. DOI: 10.1109/ICSME46990.2020.00050.

Reimers, N. (2019). "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *arXiv e-print: 1908.10084*.

Rodriguez, A. D., Dearstyne, K. R., and Cleland-Huang, J. (2023). "Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability". In: *IEEE International Requirements Engineering Conference Workshops (REW)*, pp. 455–464. DOI: 10.1109/REW57809.2023.00087.

Ronanki, K., Cabrero-Daniel, B., Horkoff, J., and Berger, C. (2024). "Requirements Engineering Using Generative AI: Prompts and Prompting Patterns". In: *Generative AI for Effective Software Development*. Springer, pp. 109–127. DOI: 10.1007/978-3-031-55642-5_5.

Rönnqvist, S., Kanerva, J., Salakoski, T., and Ginter, F. (2019). "Is Multilingual BERT Fluent in Language Generation?" In: *NLPL Workshop on Deep Learning for Natural Language Processing*. Linköping University Electronic Press, pp. 29–36.

Saldana, J. (2021). *The Coding Manual for Qualitative Researchers*. Sage Publications.

Sang, E. F. T. K. and Meulder, F. D. (2003). "Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition". In: *arXiv e-print: cs/0306050*.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2020). "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter". In: *arXiv e-print: 1910.01108*.

Santos, R., Groen, E. C., and Villela, K. (2019). "A Taxonomy for User Feedback Classifications". In: *REFSQ*. LNCS vol 2376. CEUR-WS.

Santos, R. d., Villela, K., Avila, D. T., and Thom, L. H. (2021). "A practical user feedback classifier for software quality characteristics". In: *SEKE*. DOI: 10.18293/SEKE2021-055.

Scalabrino, S., Bavota, G., Russo, B., Penta, M. D., and Oliveto, R. (2019). "Listening to the Crowd for the Release Planning of Mobile Apps". In: *IEEE Transactions on Software Engineering* 45.1, pp. 68–86. DOI: 10.1109/TSE.2017.2759112.

Schuster, M. and Paliwal, K. (1997). "Bidirectional recurrent neural networks". In: *IEEE Transactions on Signal Processing* 45.11, pp. 2673–2681. DOI: 10.1109/78.650093.

Settles, B. (2009). *Active learning literature survey*. University of Wisconsin-Madison.

Sharp, H., Rogers, Y., and Preece, J. (2023). *Interaction design. beyond human-computer interaction*. eng. 6th edition. John Wiley and Sons, Inc., p. 720. DOI: 10.1007/s10209-004-0102-1.

Sokolova, M. and Lapalme, G. (2009). "A systematic analysis of performance measures for classification tasks". In: *Information Processing Management* 45.4, pp. 427–437. DOI: https://doi.org/10.1016/j.ipm.2009.03.002.

Stanik, C. (2020). "Requirements intelligence: On the analysis of user feedback". PhD thesis. Staats-und Universitätsbibliothek Hamburg Carl von Ossietzky.

Tapia, F., Mora, M. Á., Fuertes, W., Aules, H., Flores, E., and Toulkeridis, T. (2020). "From Monolithic Systems to Microservices: A Comparative Study of Performance". In: *Applied Sciences* 10.17. DOI: 10.3390/app10175797.

Thakur, V. (2025). "Improving the Automatic Relation of Feedback and Requirements". master thesis. Heidelberg University.

Tizard, J., Wang, H., Yohannes, L., and Blincoe, K. (2019). "Can a Conversation Paint a Picture? Mining Requirements In Software Forums". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 17–27. DOI: 10.1109/RE.2019.00014.

Tizard, J., Macdonald, C., Hosking, J., and Grundy, J. (2022). "A Software Requirements Ecosystem: Linking Forum, Issue Tracker, and FAQs for Requirements Management". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 161–171. DOI: 10.1109/RE54965.2022.00024.

Ullah, T., Khan, J. A., Khan, N. D., and Anjum, N. (2022). "Can end-user rationale improve the quality of low-rating software applications: A rationale mining approach". In: *Research Square*. DOI: 10.21203/rs.3.rs-1869525/v1.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., pp. 1–11.

Villarroel, L., Bavota, G., Russo, B., Oliveto, R., and Di Penta, M. (2016). "Release planning of mobile apps based on user reviews". In: *International Conference on Software Engineering*, pp. 14–24. DOI: 10.1145/2884781.2884818.

Vogelsang, A. and Fischbach, J. (2024). "Using Large Language Models for Natural Language Processing Tasks in Requirements Engineering: A Systematic Guideline". In: *arXiv e-print: 2402.13823*.

Wang, C., Daneva, M., and al, et (2019). "A systematic mapping study on crowdsourced requirements engineering using user feedback". In: *Journal of Software: Evolution and Process* 31.10, e2199. DOI: 10.1002/smr.2199.

Wieringa, R. J. (2014). *Design Science Methodology for Information Systems and Software Engineering*. Springer, p. 332. DOI: 10.1007/978-3-662-43839-8.

Williams, G. and Mahmoud, A. (2017). "Mining Twitter Feeds for Software User Requirements". In: *IEEE International Requirements Engineering Conference (RE)*, pp. 1–10. DOI: 10.1109/RE.2017.14.

Wohlin, C. (2016). "Second-generation systematic literature studies using snowballing". In: *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 15:1–15:16. DOI: 10.1145/2915970.2916006.

Wu, S. and Dredze, M. (2020). "Are all languages created equal in multilingual BERT?" In: *Workshop on Representation Learning for NLP*. Association for Computational Linguistics, pp. 120–130.

Yan, Y. (2022). "Machine Learning Fundamentals". In: *Machine Learning in Chemical Safety and Health*. John Wiley  Sons, Ltd. Chap. 2, pp. 19–46. DOI: https://doi.org/10.1002/9781119817512.ch2.

Yang, H. and Liang, P. (2015). "Identification and Classification of Requirements from App User Reviews." In: *SEKE*, pp. 7–12.

Yang, J., Dou, Y., Xu, X., Ma, Y., and Tan, Y. (2021). "A BERT and Topic Model Based Approach to reviews Requirements Analysis". In: *International Symposium on Computational Intelligence and Design (ISCID)*, pp. 387–392. DOI: 10.1109/ISCID52796.2021.00094.

Young, T., Hazarika, D., Poria, S., and Cambria, E. (2018). "Recent Trends in Deep Learning Based Natural Language Processing". In: *IEEE Computational Intelligence Magazine* 13.3, pp. 55–75. DOI: 10.1109/MCI.2018.2840738.

Yu, H., Wang, X., Zhang, C., and Zeng, J. (2023). "Towards Automatically Localizing Function Errors in Mobile Apps with User Reviews". In: *IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 397–409. DOI: 10.1109/ICSE48619.2023.00050.

Zhang, H., Gu, S., and Zhao, W. (2022). "IRRT: An Automated Software Requirements Traceability Tool Based on Information Retrieval Model". In: *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 348–359. DOI: 10.1109/SANER53432.2022.00045.

Zhang, J., Chen, Y., Liu, C., Niu, N., and Wang, Y. (2023). "Empirical Evaluation of ChatGPT on Requirements Information Retrieval Under Zero-Shot Setting". In: *International Conference on Intelligent Computing and Next Generation NetworksICNGN)*, pp. 1–6. DOI: 10.1109/ICNGN59831.2023.10396810.

Zhang, L., Wang, H., Li, W., and Liu, L. (2021). "Recovering Semantic Traceability Between Requirements and Source Code Using Feature Representation Techniques". In: *Journal of Systems and Software* 172, p. 110848. DOI: 10.1016/j.jss.2020.110848.

Zhang, L., Huang, X.-Y., Jiang, J., and Hu, Y.-K. (2017). "Cslabel: An approach for labelling mobile app reviews". In: *Journal of computer science and technology* 32, pp. 1076–1089. DOI: 10.1007/s11390-017-1784-1.

Zhang, T. and Ruan, L. (2020). "The challenge of data-driven requirements elicitation techniques". master thesis. Blekinge Institute of Technology.

Zhang, W., Wang, X., Lai, S., Ye, C., and Zhou, H. (2022). "Fine-Tuning Pre-Trained Model to Extract Undesired Behaviors from App Reviews". In: *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 1125–1134. DOI: 10.1109/QRS57517.2022.00115.

Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K. J., and al., et (2021). "Natural Language Processing for Requirements Engineering: A Systematic Mapping Study". In: *ACM Comput. Surv.* 54.3. DOI: 10.1145/3444689.

Zheng, S. and al, et (2024). "GPT-Fathom: Benchmarking Large Language Models to Decipher the Evolutionary Path towards GPT-4 and Beyond". In: *arXiv e-print: 2309.16583*.

Zhou, Y., Ma, X., Wang, Z., and Lo, D. (2020). "User Review-Based Change File Localization for Mobile Applications". In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 467–478. DOI: 10.1109/ICSME46990.2020.00054.

# List of Figures


1.1. Social Context Goals and Problems that Prevent Goal Success . . . . . 5
1.2. Design Science Goal Structure of this dissertation including Design Goals (DG) and Knowledge Goals (KG). Arrows indicate that the fulfilment of one goal or investigation contributes to the fulfilment of the other . . . . . 6
1.3. Design Cycle Template for this Dissertation. Covering Solution Investigation (SI), Treatment Design (TD), and Treatment Validation (TV) . . . . . 8

2.1. Why do practitioners collect user feedback (Oordt and Guzman, 2021) . . . . . 12
2.2. Sources of feedback for practitioners, e denotes explicit feedback and i implicit feedback (Oordt and Guzman, 2021) . . . . . 14
2.3. Original TORE levels and categories (Kücherer, 2018) . . . . . 15
2.4. Transition from original to adapted TORE categories . . . . . 18
2.5. Simple three layered neural network (O'Shea and Nash, 2015) . . . . . 21
2.6. Basic Bi-LSTM structure . . . . . 24
2.7. The Transformer model architecture (Vaswani et al., 2017) . . . . . 25
2.8. GPT Transformer Architecture (Radford et al., 2018) . . . . . 26
2.9. SF: Start Navigation for Komoot App . . . . . 30
2.10. UT1S: Get information about the weather for SmartVernetzt . . . . . 31

4.1. Fine-grained Classification Approach Publications Per Year . . . . . 49
4.2. Dot Plot of the Dataset Sizes per Relevant Paper (SC6) . . . . . 54
4.3. Distribution of Precision, Recall, and F1 Scores Across Papers (SC10). Shown Values from Top to Bottom are: Maximum, Upper Quartile, Median, Lower Quartile, Minimum . . . . . 57

5.1. FeReRe Approach . . . . . 63

6.1. Example User Sub-Task for Komoot Hiking App . . . . . 76
6.2. Final Llama Prompts Used For TORE Classification . . . . . 83
6.3. Normalized Confusion Matrix BERT Single-Stage original (left) and combined classes (right) . . . . . 89

7.1. Feed.ai Dashboard . . . . . 96
7.2. Feed.UVL Architecture Overview . . . . . 97
7.3. Feed.UVL Services . . . . . 100
7.4. Feed.UVL Navigation Bar . . . . . 102
7.5. Feed.UVL App Review Crawler . . . . . 105
7.6. Feed.UVL Reddit Crawler . . . . . 106

7.7. Feed.UVL Upload View . . . . . 109
7.8. Feed.UVL Dataset View . . . . . 110
7.9. Feed.UVL Annotation Master View . . . . . 115
7.10. Feed.UVL Annotation Statistics View . . . . . 116
7.11. Feed.UVL Annotation Statistics Visualization View . . . . . 117
7.12. Feed.UVL Annotation Configuration View . . . . . 117
7.13. Feed.UVL Annotation Edit & Code Input View . . . . . 118
7.14. Feed.UVL Code Input View Relationship . . . . . 118
7.15. Feed.UVL Classifier Analysis View . . . . . 121
7.16. Feed.UVL Classifier Results view (Usage Information) . . . . . 122
7.17. Feed.UVL Classifier Results View (Relevance Classification) . . . . . 122
7.18. Feed.UVL Dashboard Selection and Configuration View . . . . . 126
7.19. Feed.UVL Dashboard Relation View . . . . . 127
7.20. Feed.UVL Dashboard Usage Information Classification . . . . . 128
7.21. Jira Plugin Settings View . . . . . 130
7.22. Jira Plugin Issue View . . . . . 131

C.1. Feed.UVL Domain Data Diagram . . . . . 159
C.2. Feed.UVL UI-Structure Diagram . . . . . 160

# List of Tables


1. Bachelor and Master Theses Contributing to the Dissertation . . . . . vi

1.1. Structure of the dissertation . . . . . 9
1.2. Previous Publications . . . . . 10

2.1. Analysis Types and Mined Information in App Reviews (Dabrowski et al., 2022) 13
2.2. Original TORE and Guiding Questions . . . . . 17
2.3. TORE Categories, their Definitions, and Examples . . . . . 19
2.4. Sizes of FeReRe Datasets . . . . . 31
2.5. Usage Information Dataset Characteristics . . . . . 32
2.6. SmartAge dataset Number of Sentences assigned to each Code . . . . . 33
2.7. Number of Words Assigned to each TORE Category Code . . . . . 33
2.8. Brennan & Prediger Kappa, Average Precision and Average Recall of human coders 35

3.1. Relevant Papers for Systematic Mapping Study of Software Artifact Relation . . 40

4.1. Number of Overlapping Articles Cited by the Literature Reviews . . . . . 45
4.2. Literature Studies used for Snowballing . . . . . 46
4.3. Number of Papers matching the criteria of relevance . . . . . 47
4.4. Relevant Papers for Systematic Mapping Study of Fine-grained User Feedback Classification . . . . . 48
4.5. Relevant Papers' Evaluation Metrics and Results . . . . . 49
4.6. Relevant Papers' Goals and Classes (SC1, SC2, SC3) . . . . . 51
4.7. Relevant Papers' Used Methods (SC4) . . . . . 52
4.8. Relevant Papers' Overlap Between Number of Classes and Used Methods (SC5) . 53
4.9. Relevant Papers' Dataset Size and Sources (SC6, SC7) . . . . . 54
4.10. Relevant Papers' Evaluation Methodologies (SC8) . . . . . 55
4.11. Relevant Papers' Evaluation Metrics and Results (SC9, SC10) . . . . . 56

5.1. FeReRe Experiments, Used Model and Dataset and Relevant RQ . . . . . 65
5.2. RQ1: Best Performing Model for FeReRe approach . . . . . 66
5.3. RQ1: BERT-Large Classifier Trained and Tested on Individual Datasets . . . . . 67
5.4. RQ2: Leave-one-out Experiments for Classifier Transferability . . . . . 67
5.5. RQ3: Classifier Performance Compared To GPT4o . . . . . 68
5.6. RQ3: Classifier Performance Compared To GPT4o . . . . . 68

6.1. Experiments, Used Model and dataset and relevant RQ for each Task . . . . . 79
6.2. Sentence-based TORE Level Classifier Results (RQ1.1 & RQ1.3) . . . . . 83
6.3. Class Performance for Sentence-based TORE Level (RQ1.1 & RQ1.3) . . . . . 84

6.4. Word-based TORE Level Classifier Results (RQ2.1 & RQ2.2) . . . . . 85
6.5. Class Performance for Word-based TORE Level (RQ2.1 & RQ2.2) . . . . . 85
6.6. BERT-Large Word-Based TORE Level Transferability Results (RQ2.3) . . . . . 86
6.7. Word-based TORE Category Classifier Results (RQ3.1 & RQ3.3) . . . . . 87
6.8. Class Performance for Word-based TORE Category (RQ3.1, RQ3.2, RQ3.3, RQ3.5) 87
6.9. BERT-Large Word-Based TORE Category Transferability Results (RQ3.4) . . . 88

7.1. Feedback Collection User Task & Sub-Task Requirements . . . . . 103
7.2. Feedback Collection System Function Requirements . . . . . 103
7.3. Feedback Collection Workspace Requirements . . . . . 104
7.4. Feedback Management User Task & Sub-Task Requirements . . . . . 107
7.5. Feedback Management System Function Requirements . . . . . 108
7.6. Feedback Management Workspace Requirements . . . . . 108
7.7. Manual Feedback Analysis User Task & Sub-Task Requirements . . . . . 111
7.8. Manual Feedback Analysis System Function Requirements . . . . . 113
7.9. Manual Feedback Analysis Workspace Requirements . . . . . 114
7.10. Automatic Feedback Analysis User Task & Sub-Task Requirements . . . . . 119
7.11. Automatic Feedback Analysis System Function Requirements . . . . . 120
7.12. Automatic Feedback Analysis Workspace Requirements . . . . . 120
7.13. Feed.UVL Dashboard User Task & Sub-Task Requirements . . . . . 123
7.14. Feed.UVL Dashboard System Function Requirements . . . . . 124
7.15. Feed.UVL Dashboard Workspace Requirements . . . . . 125
7.16. Jira Plugin System Function Requirements . . . . . 129
7.17. Jira Plugin Workspace Requirements . . . . . 129

8.1. Manual Relation Scenario . . . . . 136

9.1. Risk of Imperfect Classification of Usage Information . . . . . 143

B.1. Combined Table of Relevant Paper Goals, Classes, Methods, Dataset, and Evaluation 156

C.1. Prompts used for Feedback Requirements Relation using GPT4o . . . . . 158