Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

4-12-2022

Modeling Users Feedback Using Bayesian Methods for Data-Driven Requirements Engineering

Moayad M. Alshangiti
mma4247@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation

Alshangiti, Moayad M., "Modeling Users Feedback Using Bayesian Methods for Data-Driven Requirements Engineering" (2022). Thesis. Rochester Institute of Technology. Accessed from

This Dissertation is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Modeling Users Feedback Using Bayesian Methods for Data-Driven Requirements Engineering

by

Moayad M. Alshangiti

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
April 21th, 2022

Modeling Users Feedback Using Bayesian Methods for Data-Driven Requirements Engineering

by

Moayad M. Alshangiti

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

____

Dr. Qi Yu — Date
Dissertation Advisor

____

Dr. Xumin Liu — Date
Dissertation Committee Member

____

Dr. Pradeep K. Murukannaiah — Date
Dissertation Committee Member

____

Dr. Mohamed Wiem Mkaouer — Date
Dissertation Committee Member

____

Dr. Jai Kang — Date
Dissertation Defense Chairperson

Certified by:

____

Dr. Pengcheng Shi — Date
Ph.D. Program Director, Computing and Information Sciences

©2022 Moayad Alshangiti
All rights reserved.

Modeling Users Feedback Using Bayesian Methods for Data-Driven Requirements Engineering

by

Moayad M. Alshangiti

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

Data-driven requirements engineering represents a vision for a shift from the static traditional methods of doing requirements engineering to dynamic data-driven user-centered methods. App developers now receive abundant user feedback from user comments in app stores and social media, i.e., explicit feedback, to feedback from usage data and system logs, i.e, implicit feedback. In this dissertation, we describe two novel Bayesian approaches that utilize the available user's to support requirements decisions and activities in the context of applications delivered through software marketplaces (web and mobile). In the first part, we propose to exploit implicit user feedback in the form of usage data to support requirements prioritization and validation. We formulate the problem as a popularity prediction problem and present a novel Bayesian model that is highly interpretable and offers early-on insights that can be used to support requirements decisions. Experimental results demonstrate that the proposed approach achieves high prediction accuracy and outperforms competitive models. In the second part, we discuss the limitations of previous approaches that use explicit user feedback for requirements extraction, and alternatively propose a novel Bayesian approach that can address those limitations and offer a more efficient and maintainable framework. The proposed approach (1) simplifies the pipeline by accomplishing the classification and summarization tasks using a single model, (2) replaces manual steps in the pipeline with unsupervised alternatives that can accomplish the same task, and (3) offers an alternative way to extract requirements using example-based summaries that retains context. Experimental results demonstrate that the proposed approach achieves equal or better classification accuracy and outperforms competitive models in terms of summarization accuracy. Specifically, we show that the proposed approach can capture 91.3% of the discussed requirement with only 19% of the dataset, i.e., reducing the human effort needed to extract the requirements by 80%.

Acknowledgments

I would like to express my sincere gratitude to Dr. Qi Yu for all his time, support, and guidance throughout my studies. Dr. Yu has been more than an advisor to me, he is a teacher, a mentor, and a friend that I truly cherish. His knowledge and expertise were invaluable to my research and growth. His high standards in research, teaching, and work set a standard that I look up to and try to achieve. It has been a pleasure working under Dr. Yu's supervision and I will be eternally indebted to him for all that I have learned from him.

Also, I would like to thank Dr. Xumin Liu for guiding me through the various projects we worked on. I truly value the amount of effort and time she put into helping me achieve my goals and succeed in my journey. I cannot thank her enough for her help.

Moreover, I want to extend my gratitude to the doctoral committee members. Dr. Pradeep Murukannaiah and Dr. Mohamed Mkaouer. I immensely value their research guidance and feedback. Their door was always open for me and for that I will always be grateful. I would like to also thank Dr. Jai Kang for sharing his time to chair my defense.

Over the years, I worked with several people at RIT. This includes Dr. Pengcheng Shi, Min-Hong Fu, Lorrie Jo Tyrner, Charles Gruener, and many others at the Golisano College of Computing and Information Sciences. Their counsel, patience, and eagerness to help made my experience at RIT stress-free and memorable.

Additionally, I want to acknowledge my lab mates. Above all is Weishi Shi. We have spent countless hours discussing classes, research, life, worries, and every other aspect of life. It has been a pleasure knowing such an extraordinary individual, someone I consider much more than a colleague, but a true friend. Additionally, I want to thank Hitesh Sapkota and Eduardo Coelho de Lima. Working with them as a team in our projects made the whole experience much more enjoyable, and I look forward to working with them on future projects.

Last but not least, I want to thank my loving parents. Unfortunately, they both passed away during my time abroad, but I can still feel all their love. They provided me with all the care, time, and support that one can ask for and beyond. I cannot thank them enough for all they have done for me. I want to extend my thanks to my lovely and caring wife Weeam. Without her love, care, and support I would not have reached this point. My kids Waleed, Layan, and Faris. My sister Dalal whom I consider my compass in life. She guides me with her wisdom and love to be the best version of me.

To my loving parents Khadija and Mohammed.
To my wife and joy in life Weeam.
To my sister and compass in life Dalal.
To my kids Waleed, Layan, and Faris.

Contents

1 Introduction **1**

1.1 Modeling popularity to support requirements decisions 2

1.2 Building an efficient and maintainable requirements elicitation approach 4

2 Background and Literature Review **9**

2.1 Data-Driven Requirements Engineering 9

2.2 Types of Users Feedback 11

2.2.1 Implicit Users Feedback 11

2.2.2 Explicit Users Feedback 12

3 Modeling Implicit Users Feedback **14**

3.1 Introduction 14

3.2 Related Work 15

3.2.1 Popularity Prediction in Software Development 15

3.2.2 Popularity Prediction in Other Domains 15

3.3 Data Collection 17

3.3.1 Measuring the Popularity 17

3.3.2 Analyzing Data for Factors Behind Popularity 18
3.4 The Bayesian Data Modeling 20
3.4.1 Constructing the Feature Space 20
3.4.2 The Prediction Model 25
3.5 Experimental Evaluation 28
3.5.1 Experimental Setup 29
3.5.2 Model Performance 29
3.5.3 Feature Analysis 30
3.6 Discussion 33
3.6.1 Insights that Supports Requirements Decisions 33

4 Modeling Explicit Users Feedback 36

4.1 Introduction 36
4.2 Related Work 39
4.2.1 Summarizing User Reviews 39
4.2.2 Classifying User Reviews 40
4.3 Data Collection 41
4.3.1 Datasets: 41
4.3.2 Measuring Representativeness 41
4.4 Minimizing Information Loss of False Negatives 45
4.4.1 Simple but Effective: Flat versus Hierarchical Classification 45
4.4.2 Evaluation and Discussion 47

4.5 Representing Explicit Users Feedback 52
4.5.1 The Short and Noisy Nature of Explicit Feedback 52
4.5.2 Proposed Representation 54
4.5.3 Training a Word Embedding Representation 55
4.5.4 Adapting the Bidirectional Encoder Representation of BERT 56
4.5.5 Evaluation and Discussion 59
4.6 The Bayesian Framework 70
4.6.1 Overview 70
4.6.2 Why Extend RVM? 71
4.6.3 Why Merge RVM with Criticism Selection? 72
4.6.4 The Proposed Approach: Relevance Vector Machine with Criticism Selection (RVMCS) 72
4.6.5 Evaluation and Experiment 75
4.6.6 Results Discussion 82

5 Future Work **88**

Appendices **102**

A Panichella Labeling Instructions **103**

List of Figures

3.1 The use count (Y) distribution in the ProgrammableWeb dataset. We show the distribution of the popularity in (a) and the log popularity distribution in (b). . . . 18

3.2 Is there a strong correlation between the word count, tag count, API count, and the popularity? We compare the popularity of a mashup against the three potential factors (a) textual length (word count), (b) search exposure (tag count), and (c) integrated functionality (API count) 18

3.3 An example of two discovered LDA topics, a travel related topic on the left, and a real state related topic on the right. The two example topics highlight LDA's ability to summarize the textual content into a set of real-world concepts. 22

3.4 Proposed model's performance vs. other regression models. We can observe that the performance of other regression models can vary greatly depending on the selected parameter value; whereas, the proposed approach provides a consistent performance as it requires no parameter tuning. 30

3.5 The two discovered topics (i.e., functionalities). We can observe that the model was able to capture the two main concepts behind the shown test mashup, in which users share fishing information and locations. The first discovered topic shown on the left can be mapped to the general concept of *Maps and Social Sharing* based on the observed terms (e.g., *map*, *Twitter*, *share*, and *user*). The second discovered topic can be mapped to the general concept *Fishing and Wildlife* based on observed terms (e.g., *campground*, *park*, *outdoor*, and *fish*). 33

4.1 The hierarchical structure in app reviews classes 45

4.2 Evaluation of flat and hierarchical app reviews classification. On the left, we have three binary classifiers, one for each label. In this setting, each classifier is working on its own. On the right, we have four binary classifiers where the parent classifier identifies informative reviews, and then passes the informative subset to the second level where we have the three children binary classifiers, one for each label. In this setting, the children classifiers are leveraging the parent's classifiers collective knowledge. 48

4.3 Given 52 app reviews with bug reports, how were they classified in flat vs hierarchical? We can observe that on the *flat classifier* we were able to identify only 20 out of the 52 existing bug reports, i.e., we captured only 38% of the information on reported issues/bugs. On the other hand, with a *hierarchical classifier*, were able to capture 44 out of the existing 52 reviews with bug reports as informative using the parent classifier, i.e., we captured 85% of the reported issues/bugs, which is a significant minimization of the information loss. Furthermore, using the child classifier, we labeled 30 out of the 44 informative reviews correctly as bug reports. The remaining 14 reviews received an informative label, but did not receive any subclass. 51

4.4 Evaluating the best pooling strategy on the Panichella dataset. We extracted embeddings from two different BERT models and evaluated the embeddings on multiple classifiers in a multi-label multi-class setup of requirements classification. We can see that the 2TL strategy consistently provided better results. 58

4.5 Evaluating the best fine-tuning strategy on the Panichella dataset. We extracted the embeddings using the 2TL pooling strategy, which we determined to be the best, and evaluated the classifiers on a multi-label multi-class setup of requirements classification. We can see that fine-tuning BERT on the MCML task consistently provided better results, whereas, the original BERT-BASE with no fine-tuning is the least performing model. 59

4.6 A comparison between the merged terms using the traditional methods of Stemming and Lemmatization versus the merged terms using the embeddings. The plots show the score for completeness, homogeneity, and vmeasure across various cosine similarity thresholds. 60

4.7 Showing examples of terms in the embedded space and all the terms around it within a 0.8 cosine distance. We can observe that we were able to capture misspelled and alternatively spelled terms. 61

4.8 Evaluating how closely are reviews with similar requirements placed under each of the representations. We can observe that the embeddings generated from masked language modeling techniques are significantly better than other approaches. 63

4.9 Evaluating the embeddings space that is created through the masked language modeling technique against a TFIDF representation in terms of their ability to place reviews with similar requirements closer to each other. The blue points represent all the data points in the Panichella dataset. The colored points represent six subsets of reviews where each discuss a similar requirement. We can observe that in the TFIDF representation, the groups are spread across the space, whereas, in the embeddings representation they are placed closely together. 65

4.10 Evaluating the best representation for the classification task. We can observe that using an embeddings representation (i.e., FastText or BERT) provides the best average performance across datasets and across classifiers for the multi-class multi-label requirements classification task. 67

4.11 Evaluating the best representation on the Panichella dataset. We can observe that the best performing representation across clustering algorithms is the one that combines BERT embeddings with TFIDF. 68

4.12 The process for extracting requirements from app reviews using the proposed example-based summary approach. The set of reviews provided as the example-based summary contain the set of the most representation reviews in the dataset which is then used for requirement extraction. This approach requires less human effort for requirement extraction because 1) less reviews need to be manually analyzed, and 2) the provided summaries retain the context. 70

4.13 The summarization results on the Panichella dataset. First, on the y-axis, we show the coverage, which is the percentage of requirements that were captured by the approach versus the ground truth. Second, inside the whitebox at the top of the barchart, we show the recall percentage of points with level four expressiveness (L4R). Lastly, the colors of the barchart shows the level of expressiveness in the selected sample, i.e., noise vs signal ratio. 80

4.14 The Figure shows a run of regular RVM and the selected relevant vectors that it picked for the Panichella dataset. In this Figure, each dot is a requirement, and the y-axis shows the number of reviews under that requirement. The higher on the y-axis the requirement, the more reviews are talking about it, and the more dense its region would be in the space. The blue colored points are the captured requirements using the selected RVM relevant vectors. The red dot colored points are requirements that were missed by RVM relevant vectors. 83

4.15 Comparison between the proposed framework for requirements extraction and existing approaches. The proposed approach is simpler to create and maintain. It also provides a more efficient method for requirements extraction, i.e., example-based summaries. 86

List of Tables

1.1 Examples of representative reviews. The first review in the block is the representative review, whereas, the others inside the block are examples of reviews summarized by the representative review. The first block shows a review that summarizes reviews with different topics, whereas, the second block shows a review that summarizes reviews with a single topic. 6

3.1 Summary of the available information for each mashup 16

3.2 Summary statistics of the ProgrammableWeb dataset 17

3.3 Demonstrating the effect of the lack of novelty with an example of a cluster with a dominating mashup, and another with no dominating mashup 20

3.4 Finding the optimal number of topics (K) for LDA. The lowest RMSE can be observed when the number of topics is 100. 21

3.5 Examples of frequent tag/API compositions. The combination of such compositions lead to unique functionalities. For example, merging Flickr's capability with Google-maps allowed users to search for their images based on where the images were taken, i.e., location. This unique functionality, captured by the composition, can be a leading factor behind the popularity of the service mashup. 24

3.6 Examples of the model's estimated popularity (on a logarithmic scale) and confidence level compared to the true mashup's popularity. We can observe that the predicted values are close to the true values, and that the behaviour of the model matches the intuition in that the lower variance (i.e., higher confidence) maps to a more accurate model. 31

3.7 Comparing our unique approach to construct the feature space versus other standard methods in terms of complexity and accuracy. We can observe that the proposed approach provides a superior accuracy while using a significantly less number of features. 31

3.8 Measuring the incremental performance boost with each added set of features starting with the LDA topics as a base, and then adding our features incrementally. . . . 32

3.9 Demonstrating a test mashup as an example of the information a software developer would provide to the proposed model (input), and the design-phase insight he/she would recieve (output). 34

4.1 Statistics of the used datasets 41

4.2 Examples of real-world reviews from the Panichella dataset and how they were labeled in terms of representativeness. First column is the list of requirement id(s) that were mentioned in the review. E.g., Requirement ID (4) refers to users requesting additional login options, and Requirement ID (1) refers to a review process requiring from all users to have a specific number of friends on Facebook to use the app. Second column is level of Expressiveness, which is a value between 1 to 4. The least expressive is labeled with 1 and most expressive is labeled with 4. 43

4.3 Classification results of flat and hierarchical app review classifiers on Maalej Dataset 49

4.4 Classification results of flat and hierarchical app review classifiers on Panichella Dataset 49

4.5 Analyzing Random Forest: The flat vs. hierarchical classifier on the Panichella dataset. 50

4.6 Examples of misspelled words or alternatively spelled words 53

4.7 The subset of requirements selected to be manually evaluated in terms of their placement under the TFIDF representation and the FT representation. We can see their visual placement as per their color code in Figure 4.9. Additionally, we calculated the average cosine similarity between the reviews of each requirement under each representation. We can observe that the reviews of each requirement are placed significantly closer to each other in the FT representation. 64

4.8 Summary of the classification results comparing proposed approach to the start of the art on the Panichella Dataset. 77

4.9 Summary of the classification results comparing proposed approach to the start of the art on the Maalej Dataset. 77

4.10 Below are three different runs with different splits of RVMCS. We summarize the behaviour of RVMCS with each run and how the reviews with level four expressiveness were selected. We first show (RVM Only), which represents the number of points selected by RVM's maximum marginal likelihood approach. Next, (Criticism Only), which represents points selected by Kim et al. Criticism selection approach. Finally, we show (Overlap), which represents the number of points selected by both approaches. 84

4.11 Example of real-world reviews from the Panichella dataset and how RVMCS was able to capture their mentioned requirements using the least number of app reviews. 85

A.1 Meta data for the reviews in Panichella dataset 103

A.2 Examples of how to label a review with Has_User_Experience 104

A.3 Examples of how to label a review with Has_Feature_Request 104

A.4 Examples of how to label a review with Has_Bug_Report 105

Chapter 1

Introduction

Requirements are the basis of every project. It defines what the stakeholders, users, developers, and business need from a system and what the system must do in order to satisfy that need [18]. Requirements cover many aspects of a project, starting from project planning, risk management, change control to validation and documentation [10]. Thus, they form an important component in any project. The Standish Group, which is an independent international IT research advisory, conducts a survey every year since 1995 to understand project success and failure factors. In their survey of over 8380 project from 350 companies [25, 26], they found that only 16.2% of the projects were completed successfully, whereas, the remaining projects met with challenges and were partially completed with time delays and over budget (52.7%), or even worst canceled (31.1%). This high failure rate costs the United States a great deal of money as it spends more than $250 billion each year on IT application development of approximately 175,000 projects, where the average cost of a project varies between $2,332,000 and $ 434,000 depending on the project's size [25]. According to the report, the most common reasons for project failures are not technical, but rather poor requirements engineering (RE). For example, 13% were lack of user involvement, 12% incompleteness of requirements, 11% changing of requirements, 6% unrealistic expectations, and 5% unclear objectives. [25, 42]. Thus, requirements engineering is a decisive factor in the success or failure of any project.

Traditionally, requirements engineering (RE) involved users through interviews, workshops, and focus groups [59]. However, with the emergence of app stores as a software marketplace, app developers now receive abundant users feedback. This feedback was found to contain valuable information that can be used to support RE activities related to elicitation, validation, prioritization,

and management of requirements [35, 48, 59, 75, 87]. For instance, it can contain reports of bugs, requests for new features, shortcomings for existing features, description of a specific way the app is used, etc. This has inspired the idea for Data-Driven Requirements Engineering, which Maalej et al. [59] describe as "*Requirements engineering by the masses and for the masses*". In recent years, data-driven requirements engineering has attracted a lot of attention. Current research efforts under data-driven requirements engineering can be categorized as research focused on implicit users feedback, and research focused on explicit users feedback. We refer to non-verbal feedback that is usually obtained through usage data and system logs (e.g., user click data) as *implicit* user's feedback. Whereas, *explicit* users feedback is verbal feedback that the user intentionally provides in a visual and readable format, e.g., user comments.

In this dissertation, we focus on supporting requirements decisions and activities in the context of applications delivered through software marketplaces (web and mobile). We aim to use machine learning techniques to make two folds of contributions to the field of requirements engineering, which we will discuss in the next sections.

1.1 Modeling popularity to support requirements decisions

Based on our analysis of current work, one aspect that was not well studied for requirements purposes is the prioritization and validation of requirements through an implicit feedback that measures popularity such as the *number of downloads*. For example, business requirements describe what an organization hopes to achieve. They are not something a system must do. They are something the business needs to have in order to stay in business. Such requirements are usually vaguely written as they are surrounded with high uncertainty. In fact, the Standish group reports [25, 26] list improper management of expectations and poor understanding of the current user/market among the major factors behind the success/failure of projects. For example, a business requirement to *reach a million download within six months of release* is very difficult to validate (i.e., is it a reasonable expectation for this kind of app?). In many cases, validation of such requirements is usually not data-driven, but rather based on the intuition and experience of stakeholders, which is subjective, potentially inconsistent, and lacks explanation [2, 16]. However, we argue that such requirements can be validated in a quantitative data-driven way. According to the IEEE SWEBOK [10], "*requirements that cannot be validated are just wishes*".

We propose to exploit the number of app downloads, which is a measurement of popularity, to accomplish this task. We believe that through the analysis of the most successful and least successful

apps, in terms of the number of downloads, we can get a sense of how widely accepted a specific idea/concept/feature is to the general public at the current point of time. Next, based on this analysis we can build a model that aims to provide an early-on insights into the potential popularity (e.g., number of downloads) for a given application. In addition to requirements validation, exploiting such implicit feedback can help in requirements prioritizing as well, e.g., prioritize one idea/feature over another, or postponing the release of a specific idea/feature to a more community appropriate time. Moreover, through the analysis of the most successful apps and their used API/services, we can recommend/suggest the use of specific popular APIs or services, which is an important decision that determines both the implementation time and the potential reliability of a new feature. Thus, to evaluate the usefulness of this type of collective implicit feedback, we propose to formulate the problem as a popularity prediction problem and investigate the following research questions:

RQ_1: What popularity factors can we learn from analyzing software individually and collectively?

RQ_2: How to model/capture the derived factors to reach an optimized and self-explanatory feature space?

RQ_3: Given that requirements decisions are made as early as design phase, i.e., prior to implementation, how accurate would be a machine learning model at estimating a software's popularity using only verbal information about its functionality (e.g, a textual description)?

RQ_4: What kind of insight can we gain from such a model to support requirements decisions?

We present our completed work on RQ1-R4 and report our findings in Chapter 3. This work was conducted in 2016 and published later in the Journal of Expert Systems with Applications (Impact Factor: 4.29) [1]. They key contributions of our work are as follows:

- We present an in-depth investigation on the popularity of web-based software using a ProgrammableWeb service mashups dataset with 7392 service mashups covering a period of five years.
- We are the first to discuss the lack of novelty observation and to exploit the use of tag/API compositions for popularity prediction.
- We suggest a unique approach to build an optimized and self-explanatory feature space that can overcome the sparse nature of the data and quantify the popularity contribution of each feature.

- We propose a Bayesian learning model that can utilize our suggested feature space to make accurate predictions, identify important features, and offer confidence level with each prediction which provides a rich platform for predictions understanding and interpretability.
- We discuss the rich early-on insight that this model can provide to requirements validation and prioritization.
- We conduct extensive experiments over real-world service mashup data to demonstrate the effectiveness of the proposed approach.

1.2 Building an efficient and maintainable requirements elicitation approach

It was reported in a recent 2020 survey paper [78] that there is very little adoption of the proposed automated approaches in the industry. Patkar et al. [78] surveyed and interviewed participants from 45 mobile app development companies in Switzerland, Germany, and the Czech Republic. They specifically looked for people who claimed to be responsible for requirements elicitation in their company. They reported that 75% of the participants (considered requirements experts in their company) utilized app store reviews for requirements. However, all of them with no exception *manually* processed the users feedback for requirements, i.e., very little adoption of the automated approaches exists. We believe the choice of manually analyzing users feedback might lead to other choices that can limit their ability to utilize the full potential of the available users feedback. For example, all the participants in [78] reported only using app store reviews for requirements elicitation, i.e., they have not utilized other social media channels for requirements (e.g., Twitter). We believe they chose to use only a single social media channel in order to keep the volume of users feedback low, i.e., minimize the effort needed to manually inspect users feedback. This is problematic as Williams et al. [105] reported that 51% of the tweets they analyzed contained useful technical information that can be used for requirements, which means a major subset of the users feedback is not leveraged. Moreover, Nayebi et al. [68, 69] studied requirements extracted from Twitter and compared it to requirements extracted from app stores. They found that they were able to mine 22.4% additional features and 12.89% additional bug reports from Twitter, concluding that app review mining is not enough and that other information sources must be considered as they provide added value to requirements. Thus, we argue that if we want to see a paradigm shift in requirements engineering and software evolution towards data-driven user centered development, prioritization, planning, and management of requirements, we should study the reasons behind the

lack of adoption and find ways to make the automated approaches more appealing and accessible to the community.

To achieve this goal, we studied existing literature to identify potential issues that would explain the limited community adoption. We believe that in terms of technical aspects, existing approaches may not be as appealing to the community as we believe them to be due to the following issues:

First, the complexity and technical knowledge needed to implement, tune, and maintain such approaches. For example, most of the existing literature would first classify the feedback, and then cluster it using two separate models. The use of a classification model followed by a clustering model complicates the pipeline. A user of such an approach would need to know how to train, debug, fine-tune, and maintain two very different models. Any issue in one of the two would breakdown the pipeline. This becomes even more critical knowing that most of the participants in [78], i.e., experts who claim to be responsible for requirements elicitation in their company, had almost no technical experience, e.g., holding a marketing or management title/degree. In addition most of the current approaches include manual steps that are difficult to maintain beyond a small experiment. For example, one common text preprocessing issue with short text is the high number of out-of-dictionary words. We found that current approaches suggest to manually create a custom dictionary of such terms to replace them with their dictionary-equivalent terms [27,100,100], which is not a scalable nor a maintainable solution. Thus, rethinking current approaches to provide a more practical framework that takes out much of the complexity and maintenance effort while maintaining performance can potentially increase the adoption of an automated approach, i.e., providing a more accessible framework to the community.

Second, most of the existing literature provided a term level summarization, which is usually visualized using a word-cloud. However, in Williams et al. [105], where a study was conducted to evaluate word-cloud summarizes and whether they would be enough to replace manual analysis. They found that software developers *did not find the word-cloud summary particularly useful* as it is very challenging to understand the context around the shown keywords. In general, *they preferred to see full text summaries over keyword summaries.* In fact, in our own analysis, we found that when looking at informative reviews, a handful of reviews can summarize the content of the complete corpus. Table 1.1 shows the two types of possible summarization that we observed. The first, most common, is a representative review that would summarize reviews discussing a single reoccurring topic (review#86). The second, less common, is a representative review that summarizes reviews with multiple and varying topics (review#2634). In the first case, ideally we want to capture the first review, in the second case, however, we can say that selecting any review in that group would

Table 1.1: Examples of representative reviews. The first review in the block is the representative review, whereas, the others inside the block are examples of reviews summarized by the representative review. The first block shows a review that summarizes reviews with different topics, whereas, the second block shows a review that summarizes reviews with a single topic.

ID	Review
2634	According to all the members of this community, got some combined suggestions. 1...gesture control for brightness, volume, fast forward and rewind 2.**Support for**AC3, DivX and... **decoders** 3.Fix for ... play video... 4.Add a lock screen... 10.On **screen option to set the screen size 16:9** or 4:3... 11... 12. Option to add **subtitles** via external files
2827	Hello friends ... I want to play many **more formats** especially MKV and more ... PLEASE :-)
2845	Video app should have **screen resizing options like 16:9** or 16:10 or 4:3...
2808	Please add ability to show **subtitle** bundled in video ..With that ill never use mx player again
86	**Waiting** for more than 2 weeks for the **activation code**... hope this will change soon ...
65	After a week of **no activation** thereby quickly you lose the desire to use this app...
13	Not good **waiting** ... more than a week and nothing has happened...
135	**No Access Code**. They have to change the fact that ...

also be considered acceptable. We believe being able to identify those reviews would provide the level of summarization and context needed for requirements elicitation. However, how accurately can we identify these representative reviews still remains an open question.

Third, in our own analysis of existing methods, we found that most research emphasised overall model's accuracy in terms of F1 measure, and in the process provided models with moderate recall. However, in this type of problem, the ability to label all existing informative reviews correctly (i.e., recall) is far more important than mis-classifying a few *non-informative* reviews as *informative* (i.e., precision). This is because all reviews labelled as non-informative are usually disregarded (i.e.,

feedback would be lost with low recall). Ignoring this aspect makes such models less appealing due to the concern of losing valuable users feedback in the automation process. Thus, to increase trust in such automated approaches, more efforts should be placed on finding techniques to minimize information loss when filtering out non-informative reviews.

We suggest that addressing these issues may help make data-driven techniques be more appealing and accessible to the community. Thus, we plan to investigate the following research questions:

RQ_5: How can we minimize information loss when filtering out users feedback (i.e., maximize the recall while maintaining precision)?

RQ_6: How can we improve the representation of users feedback to accommodate its unique and noisy language (i.e., improve context understanding of short text and capturing of misspelled and alternatively spelled words)?

RQ_7: What are the characteristics of a representative review and how accurately can we identify such reviews?

RQ_8: How accurately can we accomplish both the classification and summarization tasks using a single model compared to state of the art?

We present our work in addressing those research question and our novel Bayesian framework in Chapter 4. We summarize our key contributions as follows:

- We discuss the information loss issue due to false negatives and showed how using a hierarchical classification approach can help boost the recall, i.e., minimize information loss, through leveraging the implicit inter-class hierarchical relationship between the labels.
- We show that in addition to learning the same patterns as stemming and lemmatization, embeddings generated from neural network models trained on the left-to-right language modeling task can learn to group the misspelled and alternatively spelled terms that posed a challenge for previous approaches.
- We show that embeddings generated from a fine-tuned BERT model using the second-to-last average pooling strategy can create a space where app reviews with similar requirements are placed closer together in terms of cosine similarity compared to the more common approaches used to represent app reviews such as TFIDF.

- We found that the best representation for both the classification and summarization task is achieved through merging a TFIDF representation with embeddings generated from a BERT model that is fine-tuned on the multi-class and multi-label requirements classification task using the second-to-last average pooling strategy.

- We expand the problem of requirements extraction from only classifying requirements using predefined labels to identifying the most representative subset of reviews for requirements extraction, which aligns better with the original goal of requirements extraction.

- We propose an end-to-end Bayesian framework that can accomplish both the classification and summarization task using a single model. We conducted comprehensive experiments to evaluate our proposed Bayesian framework and showed that it can produce equal or better results than the state of the art while addressing the issues of reliability and maintainability of previous methods.

- We demonstrate that our proposed Bayesian approach outperforms the state of the art in its ability to identify the most representative subset as it is able to capture 91.3% of the discussed requirement with only 19% of the dataset, i.e., reducing the human effort needed to extract the requirements by 80%.

The remainder of this dissertation is organized as follows: Chapter 2 discusses data-driven requirements engineering and summarizes the current literature. Chapter 3 addresses the first four research questions and presents our novel Bayesian approach that exploits implicit user feedback for requirements validation and prioritization. Chapter 4 addresses the remaining research questions and discussed our novel Bayesian framework that leverage explicit user feedback for requirements elicitation from explicit users feedback. Finally, in Chapter 5 we provide additional future directions that we believe are worth investigating.

Chapter 2

Background and Literature Review

In this chapter, we will discuss the literature review and the necessary background. First, we present an overview of the requirements engineering field, and then discuss the new vision for data-driven requirements engineering. Second, we present a summary of current research efforts categorized by the type of users feedback that is being analyzed or studied.

2.1 Data-Driven Requirements Engineering

Requirements engineering (RE) has five main activities: elicitation, analysis, specification, validation, and management [10, 81]. In *elicitation*, we identify sources of information and what requirements we can elicit from these sources. In *analysis*, we analyze the set of discovered requirements to ensure that they are well-defined and clear to both the stakeholders and the developers. Also, we classify requirements based on their type. Software requirements are classifies as: 1) business requirements, 2) user requirements, 3) functional requirements, 4) non-functional requirements. [42]. Business requirements are specified to address business objective, vision, and goals. usually defined at a high level of abstraction [42]. Functional requirements are system requirements that include the main features and characteristics of the desired system [42]. Non-functional requirements are system properties and constraints. They set the criteria for judging the operation of the system, e.g., performance, availability, reliability. [15, 23, 42]. User requirements are users wishlist's for the system, they are valuable for ensuring the system performs as the users expect [42]. In *specification*, we systematically document our requirements to establish the basis for an agreement on what the software is expected to as well as what it is not expected to do. In *validation*, we aim to ensure

the documented requirements meet the quality criteria and negotiate any potential conflicts or risks that may arise from a given requirement. Validation is done in many ways, from the review of the requirements document and creation of prototypes to the creation of acceptance tests [10]. In *management*, we manage requirements changes, prioritization (e.g., based on importance, risk, duration and cost of implementation, etc), and traceability (i.e., tracing the requirement over its entire life cycle).

Traditionally, RE activities have been stakeholder driven, i.e., elicitation of requirements relied mainly on upfront requirements based on stakeholders domain knowledge. However, this setup marginalized input from users, which is an important factor to the success of any project. In fact, according to the Standish report [25], the lack of user involvement is the most important cause of projects failure. Moreover, RE decisions, e.g., what features to add, enhance, or remove to get the most business value in terms of user satisfaction, are mainly based on the intuition and experience of small group of stakeholders, which is subjective, potentially inconsistent, and lacks well explanation [2, 16]. However, with the emergence of app stores as a software marketplace, app developers now receive abundant users feedback, which sparked a new opportunity that paved the way for data-driven requirements engineering.

Data-driven requirements engineering offers a way to involve system users, capture their needs, and get their feedback on a much larger scale than anything previously done [59]. It addresses the issue of users marginalization that occur with traditional setting. It also supports a change in the decision aspect, from the stakeholder-focused and intuition/rationale based decision making, to user-centered, data-driven decision making based on real-time analysis of the collective users feedback [59]. Thus, modern software engineering processes have now evolved from traditional static upfront requirements engineering to a more continuous approach of conducting RE, particularly approaches that leverage user generated data [59, 71]. Specifically, user generated data that represents users or crowd feedback, which we can further classify as *implicit* users feedback and *explicit* users feedback. We refer to non-verbal feedback that is usually obtained through usage data and system logs (e.g., user click data) as *implicit* user's feedback. Whereas, *explicit* users feedback is verbal feedback that the user intentionally provides in a visual and readable format, e.g., user comments. The same classification can be used to describe the current research efforts under data-driven requirements engineering, i.e., based on the type of user data that is being studied or analyzed. In Section 2.2, we will discuss these efforts in more details.

2.2 Types of Users Feedback

2.2.1 Implicit Users Feedback

Implicit feedback can be described as *all the information collected automatically on users software usage* [59]. This include usage data, logs, and all user-app interaction traces. For example, user click data is one form of such feedback, as usually a User Interface (UI) is associated with a specific feature, so analyzing how the user navigates and interacts with the UI can help developers gain a better understanding of the user needs. Another example would be the app download count, which gives us an overall understanding of the app usage and outreach. Such insight can be used to, e.g., track the impact of a new feature on the popularity of the app, i.e., how much a given feature is meeting users needs and expectations. This type of feedback is usually a continuous stream of data that is processed in real-time. It provides a wealth of feedback that can be used to understand user behaviour, e.g., analyze the usage of specific feature (interaction sequence, duration, frequency, time of day, etc). Research on collecting and analyzing implicit feedback for software engineering is mainly on error reproduction and localization, improving system usability, providing recommendations to users, or conducting usability testing [85]. However, in our own analysis and according to Wang et al. [103], analyzing such feedback for requirements is not very common. We believe the reason is the difficulty in obtaining implicit user's data as most would be considered proprietary data. Using such data would also raise concerns about user and data privacy. As such, researchers seem to prefer to work on publicly available data, which mostly consist of explicit feedback.

Schuur et al. [97] studied implicit feedback from users of a Dutch software vendor. They presented an approach that monitors performance, usage, and feedback knowledge for requirements management. Their approach generates reports that describe changes in the performance or usage data of their userbase as a way to help such vendors to respond accordingly to any potential issues with their service. Unfortunately, the work does not specify any details on the type of feedback used. Liu et al. [56] offers insights into requirements elicitation from user behavioral data analysis. They summarize potential data to collect for such analysis, e.g., user click data, eye movement tracking, and time spent on different functions. Next, they analyzed the data from a specific app and discuss different scenarios in which such data can be useful for requirements decisions. Liang et al. [55] they propose a data mining approach to extract user behaviour from user logs. They analyze location and motion logs to infer user habits when using the mobile. For example, they analyze locations visited by the user, time spent in each location, etc. They assume that users with similar habits

(e.g., going to a coffe shop every morning) would need similar requirements, i.e,. similar apps or services. Thus, based on their grouping of users, they recommend a specific service or application.

2.2.2 Explicit Users Feedback

Explicit users feedback is verbal feedback that the user intentionally provides in a visual and readable format. It is has been very well studied in the literature, mostly to understand the nature of the feedback [12,21,34,35,48,64,72] and to support requirements elicitation and analysis [11,13,22,27,32,40,44,57,58,75,76,87,88,96,99,100,101,105].

In terms of the source for the feedback, we observed that most of the current work is focused on app stores, especially Apple's app store [44,45,72] and Google's Play store [13,14,46,63,74,99,100], or both [30,31,33,57,62]. Other app stores, e.g., Microsoft's app store have limited studies [76]. For example, Pagano and Maalej [72] is one of the early investigation on the type of feedback available on app reviews. The authors identified the type of available feedback through manual analysis of the reviews, e.g., praise reviews, feature shortcoming, etc. Additionally, social media has attracted attention as well, mainly studies on Twitter [9,29,32,68,105]. For example, Guzman et al. [29] studied the contents of tweets and investigated their potential for software. Next, Guzman et al. [32] proposed ALERTme which creates a TFIDF representation [61] of tweets, and then uses a Naive Bayes classifier to determine whether a tweet has an *improvement request* or not. Tweets that contain bug reports, feature requests/shortcomings are considered tweets with improvement requests. Once tweets with improvement requests are identified, they are grouped together based on topic using a topic modeling technique call Biterm Topic Model (BTM) [109], which groups terms that co-occur together under a topic, and then each tweet is assigned a probability on its likelihood to belong to one of the topics. This technique shares a lot of similarity to the traditional topic modeling technique Latent Dirichlet Allocation (LDA) [5], but [109] and [32] claim that it performs better on short text. Finally, tweets are presented to developers and ranked considering aspects such as the number of shares, likes, and sentiment score. The authors found that despite the short length of tweets, they represent a very good source for requirements. Moreover, they found that companies tend to actively engage with users to obtain additional information, which users do follow up on, and provide a more richer context to analyze. They also suggest a future merging of user feedback generated from different channels (e.g., Twitter, app store reviews, internal reports, etc) for software evolution purposes.

Finally, we observed some requirements studies on other websites such as Vu et al. work on Phrase-based extraction of user opinions from Amazon [101] and Wattanaburanon et al. [104] work on

gamers reviews on steam.

Chapter 3

Modeling Implicit Users Feedback

3.1 Introduction

In this chapter, we present our work in exploiting the use count of service mashups (i.e., web applications) as an implicit feedback to help requirements decisions. A *service mashup* is a web application that integrates multiple sources (mainly APIs) to provide a new service in a user friendly manner. For example, *The Trend Bed*[1] is a service mashup that displays and keeps record of top trending news articles, Twitter hashtags, and YouTube videos of various countries. The web application attempts to provide a platform for analyzing current trends worldwide or for a specific country. We aim to address RQ1-RQ4 in the following four Sections. In Section 3.2, we give an overview of the related work on popularity prediction related to the software engineering domain and touch on work done on other domains as well. In Section 3.3, we discuss the used dataset, our data preparation, and our exploratory data analysis on it to address RQ1. In Section 3.4, we present our proposed Bayesian model and our unique approach in creating an optimized and self-explanatory feature space that addresses RQ2. Finally, to address RQ3 and RQ4, we discuss our experiment and evaluation, and conclude with a discussion on how this work can help requirements decisions in Section 3.5 and Section 3.6.

[1]https://thetrendbed.com/

3.2 Related Work

In this section, we describe several related work and differentiate them from ours. In general, current research efforts aim to predict the popularity of a web item [3, 20, 36, 47, 50, 52, 89, 91, 110, 111], or leverage the popularity for item filtering or recommendation [38, 39, 43, 66, 102]. In this work, we focus on the earlier, specifically popularity prediction in the software domain.

3.2.1 Popularity Prediction in Software Development

In [38, 39, 43, 66, 102], the authors use popularity prediction as part of their model to recommend APIs for mashup developers. In [38, 66], they aim to help developers decide between multiple APIs that offer the same functionality. They both developed a tool that analyzes the usage information of APIs as a metric for popularity, and use such information to make recommendations. In [43], the authors suggested a recommender system that can discover and recommend relevant web APIs to developers based on their functionality, usage, and popularity. They used the number of times an API has been used in existing APIs as a way to rank their final list of recommendation. In [39], the authors developed a tool that utilizes the popularity of APIs and their elements to rank suggestions given by code completion systems, and they show that ranking suggestions based on their usage frequency (i.e. popularity) can result in better filtering than other approaches such as alphabetical ranking or relevance ranking. Thus, [38, 39, 43, 66] have used the popularity as a feature in their model/tool to filter/rank existing APIs which is different from our work where we aim to predict the popularity itself. The only exception is [102] which we have already addressed in the introduction of this paper. We differ in that we aim to predict the popularity *before* the service is released to the public.

3.2.2 Popularity Prediction in Other Domains

Current work follows one of two directions [92]. The first is predicting the popularity prior to the release of the web item [3, 95], and the second is predicting the popularity after the release of the web item [20, 36, 37, 47, 50, 52, 53, 80, 84, 89, 91, 107, 111]. The two directions are not competing with each other, but rather, they have a complimentary relationship as the pre-release prediction can address some of the post-release prediction's limitations. They key difference is that a post-release prediction exploits the time-series information for how the popularity changes over time to make a prediction. Such information is not available when the item has not been released yet, or is in

Table 3.1: Summary of the available information for each mashup

Column	**Example**
Title	Haiku
Date	2009-07-02T21:35:07Z
Description	Parses #haiku on Twitter and matches the lines with photos from Flickr
Tags	art, haiku, microblogging, ...
APIs	Flickr, Twitter
URL	http://haiku.thehempcloud.com
Use count	7097

its early stages. Furthermore, a pre-release prediction can have a significant value when the goal is to have an early-on insight into the potential popularity of a web item to make critical budgeting or marketing decisions, which is what our work aims to provide. The literature on post-release popularity prediction suffers from the same limitation as [102] where we explained that an item has to be released to the public and used for a given period of time before a prediction can be made. This kind of setting does not apply to our problem as a pre-development prediction is required. As for the work on pre-release popularity prediction [3, 95]. The authors attempt to predict the popularity of news stories using its content. However, they were not successful as they did not have access to the full body of the news story, which limited their ability to utilize the content thoroughly. Moreover, they ignored other factors that may play a major role in the popularity of news stories such as the geographical factor where the topic might be a popularity magnet, but it is too local, i.e., popular in one source, but not the others. We align ourselves with this kind of work. However, we plan to have a more thorough analysis of the content, and to investigate other factors that may contribute to the popularity. Moreover, our proposed approach *is not simply about an accurate point prediction, but rather about providing a complete prediction framework that can offer an early-on insight into the estimated popularity of a web item, the prediction's confidence level, and the reasoning behind it.*

Table 3.2: Summary statistics of the ProgrammableWeb dataset

Column	Min	Mean	3rd Quartile	Max
Use Count	3	3474	4086	24780
log(Use Count)	1.099	8.004	8.315	10.120
Word count	1	25	33	76
Tag count	0	3	4	6
API count	0	1	2	38

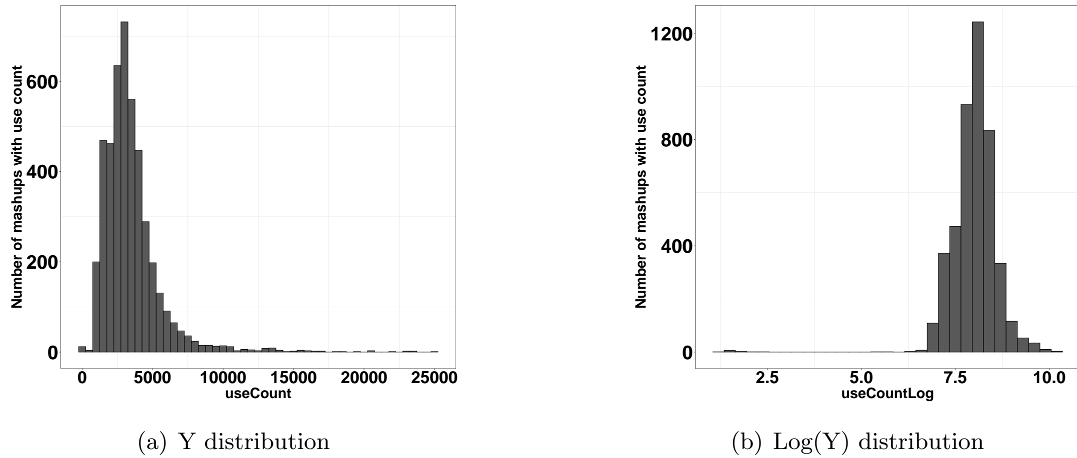
3.3 Data Collection

We used a dataset from ProgrammableWeb.com, one of the most comprehensive online directories for APIs and service mashups [43]. The website is considered a free and convenient way for developers to market their APIs and service mashups. They first started in 2005, and their directory quickly grew to over 10,000 API by 2013 [2]. We believe ProgrammableWeb.com to be a good candidate for our study because the provided list of service mashups include the list of used APIs as part of their listing, which provides us with a richer content to investigate for requirements engineering. The dataset we used was provided by [43], and it consists of 4535 mashups. In Table 3.1, we shows an example of the information provided with each service mashup as part of their listing on ProgrammableWeb. Simply put, we have a title, description, submission date, list of relevant tags, list of used APIs, URL to web application, and the use count (i.e., popularity).

3.3.1 Measuring the Popularity

We measure the popularity of a service mashup using the use count metric provided by ProgrammableWeb, which is the only provided popularity metric. Table 3.2 shows a summary statistics of the use count. The use count metric measures only the raw popularity, i.e., the level of public exposure. It does not capture other aspects of the popularity, e.g., user satisfaction, in which another metric such as the ratio of thumbs up/down would be more appropriate. Nonetheless, it is expected that in most cases the number of use count will be highly correlated with user satisfaction [8, 36].

[2]https://www.programmableweb.com/api-research

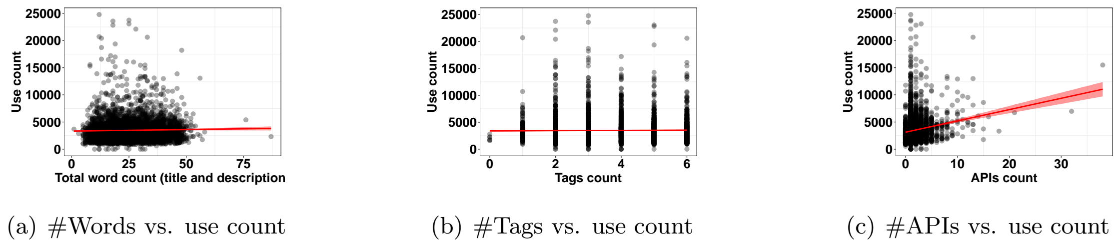


(a) Y distribution

(b) Log(Y) distribution

Figure 3.1: The use count (Y) distribution in the ProgrammableWeb dataset. We show the distribution of the popularity in (a) and the log popularity distribution in (b).

3.3.2 Analyzing Data for Factors Behind Popularity



(a) #Words vs. use count

(b) #Tags vs. use count

(c) #APIs vs. use count

Figure 3.2: Is there a strong correlation between the word count, tag count, API count, and the popularity? We compare the popularity of a mashup against the three potential factors (a) textual length (word count), (b) search exposure (tag count), and (c) integrated functionality (API count)

To address RQ_1 on the factors behind the popularity of software that we can learn from the textual content, A summary statistics can been seen in Table 3.2 where we have found that 1) seventy-five percent of service mashups use thirty-three words or less to describe their mashup, which means we have short textual information, 2) seventy percent of service mashups are tagged with two to four keywords (i.e. tags count), 3) eighty percent of service mashups use one or two APIs at most with their service mashup (i.e., API count). Based on Figure 3.2, we observed that there's no correlation between the number of words, the number of tags, and the service mashup popularity (i.e., use count). This means that having a long description or a large number of tags will have very little effect on the popularity of a mashup. However, it is also observed that having no tags will affect the popularity, as all service mashups with zero tags ended up being in the low popular range as seen in Figure 3.2. We believe that not properly tagging a service mashup when listing it

in online markets can limit the users ability to find it, which may explain this observation.

On the other hand, we can see a much stronger correlation, in Figure 3.2, between the API count and the popularity. We observed that service mashups in the *high* popular range mostly use one to three APIs; whereas, service mashups that use more than three APIs immediately lower their chances of being in the *high* popular range. When taking a closer look, we found that service mashups with a high number of APIs are mostly not targeting the general public, but a more specific audience. For example, *USPS Tracking* is a service mashup in the upper half of high popularity range (e.g., 20,699 use count) which uses only two APIs (Google Maps and USPS Track & Confirm), and offers a service to track USPS shipments with Google Maps, and is considered relevant to a wide range of audience which explains its very high popularity. Whereas, *Congress SpaceBook* is a service mashup in the lower half of medium popularity range (i.e., 4737 use count) which uses eleven APIs (e.g., Flickr, YouTube, Google Social Graph, ..etc), and basically offers a social networking platform for congress, is considered relevant to a significantly smaller audience which explains its low popularity. Thus, the general observation is that the more APIs consumed by a service mashup, the higher the chances of it being in the medium popular range (i.e., 2000-7000 use count) as it will most likely be targeting a much smaller audience, so even if it was successful in reaching it's targeted audience, it will still overall be considered within the medium-low popular range (i.e., below 3rd quantile).

When taking a closer look into the functionality the mashups offer, we found that similar mashups have an interesting relationship between them. If we consider a group of similar mashups to be forming a cluster for a specific functionality (e.g., they all offer a hotel finding service), then we can observe that they fall under one of two states: They either have a dominant mashup (i.e., a mashup that has captured most of the attention for that functionality), in which case that dominating mashup would have a significantly higher popularity than its neighbors within the cluster, or they would all be closely related in popularity with no dominant mashup. Table 3.3 shows an example of a cluster with a dominating mashup, and an example of a cluster with no dominating mashup. We can see that mashups within the same cluster offer similar functionality. For the first case, we observe that once a dominating mashup appears, all the later mashups are likely to be in the low-range popularity of that cluster. As for the second case where we do not have a dominating mashup, we believe that if a cluster has an overall mid-range popularity, then the cluster's functionality can be considered a promising open area for developers to try and build the next mashup that will dominate it. However, in case the cluster had an overall low-range popularity average, then this may indicate that this cluster offers a useless or uninteresting functionality that developers should avoid in the future. In rare cases, a cluster of similar mashups can be dominated

Table 3.3: Demonstrating the effect of the lack of novelty with an example of a cluster with a dominating mashup, and another with no dominating mashup

Cluster (8) with a dominating mashup		
Title	**Pub. Date**	**Use count**
1001 Secret Fishing Holes	Nov. 2005	23,567
Fishingnotes.com	Mar. 2003	3125
Fish Mapper	Apr. 2006	3011
Fishing Stories	Oct. 2006	2842
Flyfishmap	Jun. 2009	1673
...	...	...
Cluster (658) with NO dominating mashup		
Title	**Pub. Date**	**Use count**
Earthquake Vulnerable Cities	Aug. 2008	2785
Earthquakes in Last 7 Days	Nov. 2005	3146
Earthquakes this Week	Nov. 2005	4082
World and Regional Earthquakes	Nov. 2006	2322
...	...	...

by more than a single mashup, however, we have found that in most cases, we only have a single dominating mashup.

3.4 The Bayesian Data Modeling

In this section, we discuss our suggested approach which consists of our method to construct an optimized and self-explanatory feature space from raw sparse data, and our Bayesian learning model that can predict, select features, and offer confidence level with each prediction.

3.4.1 Constructing the Feature Space

The Functionality: To derive the functionality of a mashup, we suggest leveraging its title and description as follows. First, we apply a standard natural language processing methods, such as stop-word removal and word stemming, on the textual content of the title and the description to

Table 3.4: Finding the optimal number of topics (K) for LDA. The lowest RMSE can be observed when the number of topics is 100.

K	5	30	50	**100**	250	500	1000
RMSE	0.6302	0.6292	0.6242	**0.6181**	0.6270	0.6290	0.6355

generate a *term frequency-inverse document frequency* matrix or TF-IDF matrix [61]. The TF-IDF matrix is a representation of the content where each row is a mashup, and each column is a term. The elements in this matrix represent how relevant a given term is to a specific mashup. This representation allows us to capture the most important terms that describe the content of a mashup. However, TF-IDF usually produces a large matrix that is highly sparse, i.e., a given mashup's vector would have many zero entries as it uses only a few terms out of the available dictionary.

To address this issue, we utilize the probabilistic topic modeling technique Latent Dirichlet Allocation (LDA) [5]. The intuition behind using LDA is that given the TF-IDF matrix, LDA can leverage such representation by grouping together the frequently co-occurring terms into an approximation of a real-world concept, i.e., a topic. The set of topics discovered by LDA would represent a higher level summary of the terms discovered by the TF-IDF approach. As such, LDA is expected to provide a good and compact approximation of the TF-IDF matrix as the number of topics in the LDA matrix is significantly smaller than the number of terms in the TF-IDF matrix. LDA produces a topic proportion matrix D where each row in the matrix represents a mashup, and each column represents a discovered topic. The entries $D_{i,k}$ in the LDA matrix essentially denote the *probability* that topic k describes mashup i. As part of using LDA, we need to specify the number of topics k, and through cross-validation, as seen in Table 3.4, we found one-hundred to be a good candidate as it offers a balance between model's complexity and model's accuracy. We believe those topics represent the mashups functionalities that we aim to derive. To give a better insight into those discovered topics, Figure 3.3 shows the content of two topics, the first (left side) is about traveling, while the second (right side) is about real-estate. We learn the contribution of each discovered topic as follows:

$$d_k = \frac{\sum_{i=1}^{m} D_{i,k} \times y_i}{\sum_{i=1}^{m} D_{i,k}} \quad , \forall k \in \{1, .., K\} \tag{3.1}$$

where m is the total number of mashups, K is the total number of topics, and y_i is the corresponding

Figure 3.3: An example of two discovered LDA topics, a travel related topic on the left, and a real state related topic on the right. The two example topics highlight LDA's ability to summarize the textual content into a set of real-world concepts.

popularity (i.e., use count) for mashup i. Thus, each entry in the vector d_k is a score that indicates the topic's contribution towards the popularity. When splitting the dataset into training and testing, we learn the contribution of a new testing mashup with vector $\boldsymbol{\theta}_t \in \mathbb{R}^n$ as follows:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_t \circ \boldsymbol{d} \tag{3.2}$$

simply, we do an element-wise multiplication between the new mashup's probability vector and the topic-contribution vector that represents the contribution of each discovered topic towards the popularity. We use the generated LDA matrix with the new topics score directly in our model as features.

The Selection of Tags and Services: The standard way to capture the use of tags and the selection of services (i.e., APIs) is to create two binary frequency matrices. The rows in those matrices represent our mashups and the columns represent the used tags in the first matrix, and the selected services (i.e., APIs) in the second matrix, where each entry denotes if a given tag/API was used in a given mashup or not (i.e., binary score). However, since we have 1409 unique tag, and 788 distinct service (i.e., API), and that developers use on average 2-3 tags and 1-2 APIs per servie mashup, we have an extremely sparse matrix. Thus, we suggest a better two-step approach to replace those two sparse and large matrices with only two features: the tag score feature and the API score feature. These score features will denote the contribution of the used tags, for the tag score, and contribution of selected services (i.e., APIs) for a given mashup. We constructed those two features as follows: In step one, we learn the *averaged* contribution of each tag/API towards

popularity. To learn the contribution of each tag, we divide the use count (i.e., popularity) of each mashup in the tag matrix by the number of tags it uses, and assign that as a new score for the used tags. At this point, for each column in both the tags matrix, we have a score that represents the contribution of that tag/API towards the popularity of the mashups. We take the average of each column which represents the *averaged* contribution towards popularity for a given tag, and assign it as a score for the whole column. We do the same for the API matrix to learn the *averaged* contriubution of each API towards the popularity. In step two, given a service mashup, we add up the individual averaged contributions of the tags that it uses to create the tag score feature, and add up the individual contribuitons of APIs that it uses to create the API score feature. When splitting the data into training and testing, we use the average contribution for each tag/API that we learned from training as a score for the testing as well. We then add up the individual contributions in the same manner.

The Combination of Selected Tags/Services: To capture the role such compositions play in the popularity of a mashup, we suggest finding those compositions and building a binary frequency matrix that allows us to use them as features. To find those compositions, we suggest the use of Apriori algorithm [90] which is a standard technique to find frequently used compositions. The selected *support* level for Apriori should offer a balance between finding all possible compositions and maintaining a statistical meaning for the compositions. It is expected to have a large number of compositions, and that should not be a problem as our suggested Bayesian learning model can select the most relevant ones. In our dataset, we were able to find 178 frequently used compositions. Table 3.5 shows a few of the discovered compositions. For example, the first composition represents the use of (Photo and Map) as tags and (Flickr and Google-maps) as APIs. This combination created a mashup with an interesting functionality that allowed users to know the location of where their Flickr images were taken. We believe this interesting functionality, captured by the composition, is behind the popularity of the mashup. We used those frequent compositions to create the binary frequency matrix which we used directly as features in our model.

Novelty: As we have explained earlier, a mashup may fail to attract its users if similar mashups are already available and have taken up the market. We observed this through our analysis in which we found that when we cluster similar mashups together, it's common to see one of two states: A cluster with a dominant mashup, or a cluster with no dominant mashup. In the first case, we observed that once a dominant mashup appears, it would capture most of the attention for that cluster's functionality forcing all the other mashups, especially the later ones, in that cluster to settle-in for a lower popularity. In other words, we can say that the other *dominated* mashups within the cluster *lack the novelty* as the dominant mashup is presumed to be the first in the cluster

Table 3.5: Examples of frequent tag/API compositions. The combination of such compositions lead to unique functionalities. For example, merging Flickr's capability with Google-maps allowed users to search for their images based on where the images were taken, i.e., location. This unique functionality, captured by the composition, can be a leading factor behind the popularity of the service mashup.

Tags	**APIs**
Photo, Map	Flickr, Google-maps
Video, Music	YouTube
Social, Microblog	Twitter
Video, Photo	Flickr, YouTube

to successfully capture all the user's needs for that functionality. Thus, we suggest to create a new feature vector called the *lack of novelty* where we penalize all the *dominated* mashups with a score of one, as we expect them to have a low/medium popularity (i.e., use count below 3rd quantile), and assign a score of zero to all other mashups including dominating mashups and mashups in clusters with no dominating mashup as we have no evidence that they lack the novelty. We then use that vector as a feature in our model.

However, to achieve the suggestion mentioned above, we need to determine the best approach to measure the similarity between the functionality of two mashups. We suggest combining the knowledge from both the content found in the title/description of the mashup, and from the list of used tags and APIs as follows:

$$Sim_{i,j} = \alpha \times \mathbf{C}_{i,j} + (1 - \alpha) \times \mathbf{J}_{i,j} \tag{3.3}$$

where α is a learned probability weight between zero and one. $\mathbf{C}_{i,j}$ is a cosine similarity matrix [90] that measures the similarity between the title and the description of two given mashups. $\mathbf{J}_{i,j}$ is a jaccard similarity matrix [90] that measures the similarity between the list of tags and APIs of two given mashups. The α weight measures how much trust you place on your content from the title/description. If the dataset lacks proper description, but is tagged properly, then less weight can be placed on the content from the title/description so that more weight is placed on the list of used tags and APIs, and vice versa. If there is no clear pattern in the dataset, then the recommended approach in such case would be to provide equal weights to both aspects. However, if there's a clear preference in the dataset (i.e., community), the proposed approach can provide better predictions if the preference is reflected in the provided weights. In the rare case where mashups are posted

without any meta information (i.e., both the description and tags are empty), the model would not have enough data to make a confident predication. Nonetheless, it is expected that such an extreme case (i.e., no meta information) would be difficult even for human experts as they would find it impossible to make a judgement with no available information on the mashup. It is important to clarify that in this approach we assume that professional developers will put a great deal of effort in preparing the meta information (i.e., description and/or tags) of their mashup to ensure proper exposure of their work. This assumption is needed for the model to provide accurate and confident predictions. For our dataset, with cross-validation, we have found that an α value of 0.9 produced clusters that met our requirement in that mashups were clustered together based on functionality.

Next, we suggest using hierarchical clustering [90] to create the clusters using the averaged similarity matrix $Sim_{i,j}$ that we already constructed. As it's the case with most clustering algorithms, in hierarchical clustering, we need to specify the number of clusters as a parameter to the algorithm. we found 2197 to be a good number of clusters for our dataset. The number of clusters we chose is the total number of unique tags (1409) and APIs (788). Since each cluster should represent a unique possible functionality, the choice of the number of cluster should represent the number of unique possible functionality we assume to exist in the dataset. Thus, we are making the assumption that for each unique available tag and API, at least a single possible unique functionality exists.

Finally, to identify the clusters with a dominant mashup from the ones without, we looked for an outlier point in the cluster where we measured how many standard deviations each point is away from the mean using *z-score*. To determine if a point within a cluster is an outlier or not, we measure how many standard deviations it is from the mean of the cluster. If we found that it's t standard deviations away from the mean, then we declare it as a dominating mashup. The value for t has to be determined through cross validation. In our case, we have found three to be a good value for t. We can now create our lack of novelty feature vector as described above, and use it as a feature in our model.

3.4.2 The Prediction Model

We present a Bayesian learning model for popularity prediction. The proposed model offers three major advantages over other regression models. First, instead of just providing a point prediction, the Bayesian model outputs a predictive distribution for a given test mashup. The variance of the predictive distribution can be used to quantify the confidence level of the prediction. Second, we integrate the Bayesian learning model with the Auto Relevance Determination (ARD) mechanism [4], which allows us to perform feature selection and identify the most important factors that

affect mashup popularity. Third, by performing type 2 maximum likelihood, we can automatically optimize the hyperparameters of the model, which avoid the tedious process of cross-validation required by many other models.

Model inference

We start by assuming the response t is a random variable whose distribution conditioned on input x is Gaussian:

$$p(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}), \beta^{-1}) \tag{3.4}$$

where β is the precision of the Gaussian and $\boldsymbol{\phi}(\mathbf{x})$ is the feature vector of mashup $\mathbf{x}$.

The likelihood of the training data $\mathbf{X}$ then is given as:

$$p(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} p(\mathbf{t}_n|\mathbf{x}_n, \mathbf{w}, \beta^{-1}) \tag{3.5}$$

The flexibility of the Bayesian inference framework allows us to incorporate different prior knowledge for different learning effects. Specifically in this work, we assume that not all features are equally important to the prediction problem. As a result we choose a conjugate Gaussian prior(A.K.A ARD prior) on the coefficient random variable $\mathbf{w}$ to conduct feature selection:

$$p(\mathbf{w}|\boldsymbol{\alpha}) = \mathcal{N}(\mathbf{0}, A^{-1}) \tag{3.6}$$

where A is a diagonal matrix governed by hyper-parameter $\boldsymbol{\alpha}$ where α_i denotes the i-th diagonal entry of A. Section 3.4.2 provides the detailed discussion of how feature selection can be achieved by adopting ARD prior.

According to the Bayesian rule, the posterior distribution of $\mathbf{w}$ is proportion to the product of the likelihood and prior, which is also Gaussian due to conjugacy:

$$p(\mathbf{w}|\mathbf{t}, \boldsymbol{\alpha}, \beta) = \mathcal{N}(\mathbf{w}|\mathbf{m}, \Sigma) \tag{3.7}$$

where the posterior mean and the covariance are given as follows:

$$\mathbf{m} = \beta\Sigma\Phi^T, \quad \Sigma = (A + \beta\Phi\Phi^T)^{-1} \tag{3.8}$$

Φ is the design matrix. The i-th row of Φ is $\boldsymbol{\phi}(\mathbf{x}_i)$. Assume that the optimal values of the hyper-parameters, $\boldsymbol{\alpha}^*$ and β^* can be learned (see the next section for details). We can derive the predictive

distribution over a test mashup $\mathbf{x}_t$ by integrating out $\mathbf{w}$, which is also a Gaussian:

$$\begin{aligned} p(t|\mathbf{X}, \mathbf{x}_t, \boldsymbol{\alpha}^*, \beta^*) &= \int p(t|\mathbf{x}_t, \mathbf{w}, \beta^*) p(\mathbf{w}|\boldsymbol{\alpha}^*, \mathbf{X}, \beta^*) d\mathbf{w} \\ &= \mathcal{N}(\mathbf{m}^T \boldsymbol{\phi}(\mathbf{x}_t), \sigma^2(\mathbf{x}_t)) \end{aligned} \tag{3.9}$$

where the predictive mean and the covariance are given as follows.

$$\sigma^2(\mathbf{x}_t) = (\beta^*)^{-1} + \boldsymbol{\phi}(\mathbf{x}_t)^T \Sigma \boldsymbol{\phi}(\mathbf{x}_t) \tag{3.10}$$

Besides using the mean of the predictive distribution (i.e., $\mathbf{m}^T \boldsymbol{\phi}(\mathbf{x}_t)$) to predict the future use count of $\mathbf{x}_t$, the variance $\sigma^2(\mathbf{x}_t)$ provides important information to quantify the confidence level of the prediction.

Learning Process

Estimating hyper-parameters $\boldsymbol{\alpha}, \beta$ yields a type-2 maximum likelihood problem. Specifically, we maximize the log of the model evidence given by:

$$\begin{aligned} \ln p(\mathbf{t}|X, \boldsymbol{\alpha}, \beta) &= \ln \int p(\mathbf{t}|X, \mathbf{w}, \beta) p(\mathbf{w}|\boldsymbol{\alpha}) d\mathbf{w} \\ &= \ln \mathcal{N}(0, C) \\ &= -\frac{1}{2}(N \ln(2\pi) + \ln(C) + \mathbf{t}^T C^{-1} \mathbf{t}) \end{aligned} \tag{3.11}$$

where C is given by

$$C = \beta^{-1} I + \Phi A^{-1} \Phi^T \tag{3.12}$$

By setting the partial derivative of (3.11) with respect to $\boldsymbol{\alpha}$ and β to zero, we derive the solutions for both hyper-parameters

$$\begin{aligned} \alpha_i^* &= \frac{\gamma_i}{m_i^2} \\ (\beta^*)^{-1} &= \frac{||\mathbf{t} - \Phi \mathbf{m}||^2}{N - \sum_i \gamma_i} \end{aligned} \tag{3.13}$$

where γ_i is defined by

$$\gamma_i = 1 - \alpha_i \Sigma_{ii} \tag{3.14}$$

The learning proceeds by using (3.8) and (3.13) alternatively with randomly initialized $\boldsymbol{\alpha}$ and β until convergence.

Feature Selection

The first updating rule from (3.13) implies an implicit solution as the right hand side is also a funtion of α_i. To determine the stationary point of the log likelihood function (3.11) explicitly, we can extract the contribution from α_i out of the covariance matrix C in (3.11):

$$\begin{aligned} C &= \beta^{-1}I + \sum_{j\neq i}\alpha_j^{-1}\boldsymbol{\varphi}_j\boldsymbol{\varphi}_j^T + \alpha_i^{-1}\boldsymbol{\varphi}_i\boldsymbol{\varphi}_i^T \\ &= C_{-i} + \alpha_i^{-1}\boldsymbol{\varphi}_i\boldsymbol{\varphi}_i^T \end{aligned} \tag{3.15}$$

where $\boldsymbol{\varphi}_i$ denotes the i-th column of Φ and C_{-i} represents matrix C with the removal of $\boldsymbol{\varphi}_i$. Substituting (3.15) in (3.11), the log likelihood can be written as:

$$\ln \mathcal{N}(0, C) = L(\boldsymbol{\alpha}_{-i}) + \lambda(\alpha_i) \tag{3.16}$$

where $L(\boldsymbol{\alpha}_{-i})$ denotes the log likelihood function with $\boldsymbol{\varphi}_i$ omitted and function $\lambda(\alpha_i)$ is defined as:

$$\lambda(\alpha_i) = \frac{1}{2}\left[\ln\alpha_i - \ln(\alpha_i + s_i) + \frac{q_i^2}{\alpha_i + s_i}\right] \tag{3.17}$$

where q_i and s_i are defined as:

$$\begin{aligned} s_i =& \boldsymbol{\varphi}_i^T C_{-i}^{-1}\boldsymbol{\varphi}_i \\ q_i =& \boldsymbol{\varphi}_i^T C_{-i}^{-1}\mathbf{t} \end{aligned} \tag{3.18}$$

The partial derivative of (3.17) with respect to α_i is

$$\frac{\alpha_i^{-1}s_i^2 - (q_i^2 - s_i)}{2(\alpha_i + s_i)^2} \tag{3.19}$$

Setting (3.19) to zero gives two possible solutions for α_i:

$$\begin{cases} \alpha_i \to \infty, q_i^2 < s_i \\ \alpha_i = \frac{s_i^2}{q_i^2 - s_i}, \text{otherwise} \end{cases} \tag{3.20}$$

In the first case, as α_i (i.e., precision of coefficient w_i) approaches to infinity, w_i will be driven to its mean (i.e., 0). This will result in the removal of the corresponding feature from the model, which achieves feature selection.

3.5 Experimental Evaluation

We first describe the experimental setup. We then compare the prediction accuracy of our proposed Bayesian learning model with other competitive models, and show how our learning model is overall

superior. Moreover, we show examples of our model's ability to offer confidence level with each prediction. Finally, we evaluate our approach in building the feature space and ability to identify the important features.

3.5.1 Experimental Setup

The current use count (i.e., popularity) of a mashup is the total number of use count accumulated over the years since its publication date. Thus, it favors older mashups over newer ones as the newer mashups had less time to accumulate their use count. To have a more balanced and fair scale, we instead used the average yearly use count as our response in which we divided the original use count by the mashup's lifetime (i.e., number of years it's been available in the market).

To simulate a real world scenario, we used the information from the mashups listed in the first four years as the training data, to predict the popularity of the fifth's year mashups. In other words, we are training the model on service mashups that were created in the first four years, and testing the model on service mashups that were created on the fifth year. To measure the prediction accuracy, we used Root Mean Square Error (RMSE) which is a standard way to measure the difference between the true and predicted values. In general, the lower the RMSE, the more accurate the model.

3.5.2 Model Performance

To evaluate the prediction performance of our proposed Bayesian learning model versus other models, we used a feature space of 287 features where for each mashup we have a tag score feature, an API score feature, a Lack of Novelty score feature, 100 LDA features (one feature per topic), and a 174 binary API/tag composition features.

Figure 3.4 shows the prediction result of our proposed model versus Linear Regression with L1 norm regularization (i.e., Lasso) and Linear Regression with L2 norm regularization (i.e., Ridge Regression). Both ridge regression and lasso require parameter tuning (i.e., lambda) and their performance significantly rely on the selected parameter value. For ridge regression, we can see that a low or a high lambda value can drastically decrease the performance; whereas, with lasso regression, the higher the lambda value, the lower the performance as the model becomes more selective of what features to use. On the other hand, our proposed bayesian learning model does not require any parameter tuning as it can directly give the optimal or near-optimal solution.

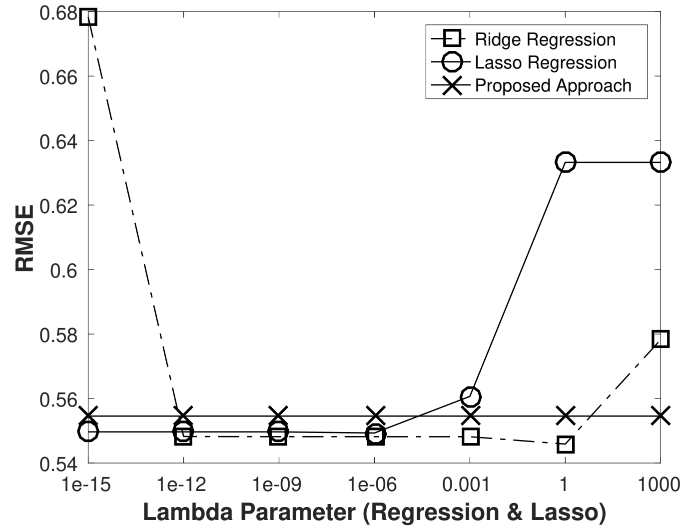


Figure 3.4: Proposed model's performance vs. other regression models. We can observe that the performance of other regression models can vary greatly depending on the selected parameter value; whereas, the proposed approach provides a consistent performance as it requires no parameter tuning.

Moreover, it can identify the important features which ridge regression does not offer, and it provides a confidence level with each prediction which both ridge and lasso regression cannot do. Thus, overall, it offers the best prediction framework.

To show how our suggested learning model can offer confidence level with each prediction, we present a few examples in Table 3.6. The first one is a mashup we predicted with a relatively large error, and the second one is a mashup we predicted with a smaller error. The general observation is that if we have a small variance, then we are more confident about the prediction, and vice versa. For example, we predicted more accurately the second mashup, and the model confirms that fact by showing a small variance for the prediction (i.e., a high confidence). On the other hand, our prediction of the first mashup is more off as the model does not have enough historical data to make a more accurate prediction, so the model presents a much higher variance which means it has a low confidence in this prediction.

3.5.3 Feature Analysis

To show the performance boost when using our unique approach to construct feature space versus simpler standard methods, we created an alternative feature space that uses word frequency matrix

Table 3.6: Examples of the model's estimated popularity (on a logarithmic scale) and confidence level compared to the true mashup's popularity. We can observe that the predicted values are close to the true values, and that the behaviour of the model matches the intuition in that the lower variance (i.e., higher confidence) maps to a more accurate model.

Mashup	True Pop.	Predicted Pop.	Variance	SD
Adult Or Not	8.4740	7.4677	0.3763	0.6134
QuoteRelish	6.9697	7.1060	0.2655	0.5152

Table 3.7: Comparing our unique approach to construct the feature space versus other standard methods in terms of complexity and accuracy. We can observe that the proposed approach provides a superior accuracy while using a significantly less number of features.

Approach	#Features	RMSE
Standard methods	10455	1.1046
Suggested approach	277	0.5545

to capture the role of the title and the description of the mashup, and a binary frequency matrix to capture the role of the tags and APIs. The result can be seen on Table 3.7 where our approach in constructing the feature space is not only offering a significantly smaller feature space, but also a drastically better prediction accuracy compared to the frequency approach.

Furthermore, we show in Table 3.8 the added value of each suggested feature using different models as follows:

- **Base**: Using the 100 topics generated from LDA, where each topic's probability is replaced with the calculated score (100 features).
- **+ Compositions**: Using the previous model features and the binary matrix of compositions generated from the Apriori algorithm as features (274 features).
- **+ API Score**: Using the previous model features and the API score feature (275 features).
- **+ Lack of Nov.**: Using the previous model features and the lack of novelty feature (276 features).

Table 3.8: Measuring the incremental performance boost with each added set of features starting with the LDA topics as a base, and then adding our features incrementally.

Model	**#Features**	**RMSE**
Base	100	0.6287
+ Compositions	274	0.5897
+ API Score	275	0.5742
+ Lack of Nov.	276	0.5688
+ Tag Score	277	0.5545

- **+ Tag Score**: Using the previous model features and the tag score feature (277 features).

As we can see, the baseline is performing quite well as expected since the discovered LDA topics are able to capture the offered functionality of the mashup, and their current score represent their contribution towards the popularity. Nonetheless, each of our added features was still able to improve the model, and collectively they improved the baseline model's accuracy by roughly 12%. The compositions offered the biggest improvement, but it added a high complexity (174 new features). Whereas, the other three features were able to collectively add the same level of improvement to the model but with drastically less added complexity which shows their significance. Furthermore, we used random forest regression and lasso regression as well as our proposed bayesian model to evaluate their importance to the model as features. We found that all three models picked the *tag score feature* and the *API score feature* as the top two most important features which confirms that they play a major role in the accuracy of the model. The models did not all agree on the rank of the *lack of novelty feature* as our bayesian learning model suggested it was the 7th most important feature, random forest as the 16th, and lasso as the 25th. We believe this is the case because the lack of novelty feature is targeting a specific observation, and thus is used for only a small subset of the mashups (roughly 16% in our dataset) which may not show an overall significance, but should be critical for those relevant mashups.

Figure 3.5: The two discovered topics (i.e., functionalities). We can observe that the model was able to capture the two main concepts behind the shown test mashup, in which users share fishing information and locations. The first discovered topic shown on the left can be mapped to the general concept of *Maps and Social Sharing* based on the observed terms (e.g., *map*, *Twitter*, *share*, and *user*). The second discovered topic can be mapped to the general concept *Fishing and Wildlife* based on observed terms (e.g., *campground*, *park*, *outdoor*, and *fish*).

3.6 Discussion

3.6.1 Insights that Supports Requirements Decisions

To address RQ_4 which aims to investigate what kind of insight can we gain from the model to support requirements decisions, we show a test mashup from our experiment in Table 3.9 where the developer created a web app that shows nearby fishing locations on Google Maps. Also, the web app allows users to post images and YouTube videos to share their experience. To evaluate the test mashup, we need only textual description of the functionality as shown in Table 3.9, which is expected to be known early-on. Given this information, here is the kind of insight the suggested model would offer to support requirements decisions:

- First, we provide an estimated popularity with a low variance of 0.27 which indicates a high confidence in the prediction. We can see that the model is quite accurate with only a 633 use count difference between the true and predicted popularity.
- Second, as seen in Figure 3.5, two functionalities were discovered. The first (i.e., maps and social sharing of images and videos) attracts a large audience as the functionality contribution score is above the 90th percentile of all the discovered functionalities scores. Whereas, the second (i.e., fishing and wildlife) attracts only a small audience as its score is below the 30th

Table 3.9: Demonstrating a test mashup as an example of the information a software developer would provide to the proposed model (input), and the design-phase insight he/she would recieve (output).

Mashup's Information	
Title	Flyfishmap
Description	User generated fly fishing information using Google...
Tags Used	#fishing #flyfishing ...
APIs used	Google-maps, YouTube...

Mashup's Design-Phase Popularity Insight	
True Popularity (Log)	7.4223 (i.e., 1673 use count)
Predicted Popularity (Log)	6.9475 (i.e., 1040 use count)
Popularity Range	Low
Prediction Variance	0.27243
Func.#1 (Maps & Social Sharing)	25.3 (above 90th Percentile)
Func.#2 (Fishing & Wildlife)	3.26 (below 30th Percentile)
Lack of Novelty	1
Tags Contribution Score	4.04 (below 10th Percentile)
APIs Contribution Score	15.41 (above 90th Percentile)
Popular Combinations	map (tag), YouTube (API), Google-maps (API)

percentile.

- Third, the idea lacks the novelty as this functionality is already offered by an existing mashup that has successfully captured the market.
- Fourth, the selected tags (all related to fishing) attract a small audience (i.e., users searching for such functionality represent a small club) as the tags contribution score is below the 10th percentile.
- Fifth, the selected APIs (i.e., Google-maps and YouTube) attract a large audience as the APIs contribution score is above the 90th percentile of the APIs scores.
- Finally, the used combination of tag (maps) and APIs (YouTube and Google-maps) is a

popular combination.

Given such insight, the developer is well-informed *early-on* of the estimated popularity, the estimated audience for each component (i.e., strength/weakness), and the confidence in the estimation. This estimation can be thought of as as measure for the *value of a feature*. Based on the Standish group reports [25,26], among the factors related to the success/failure of projects are improper management of expectations and poor understanding of the current user/market (i.e., project is not needed). We believe such insight can help in this context as it provides insights into the current user/market preferences, especially for requirements prioritization and validation. For example, through the estimated value for a feature or specific functionality, we can prioritize one feature over another, or postpone the release of a specific idea/feature to a more community appropriate time. Moreover, through the analysis of the most successful apps and their used API/services, we can recommend/suggest the use of specific popular APIs or service, which can be valuable for *effort estimation*. It can help with determining both the implementation time and the potential reliability of a new feature. Additionally, we can use it for requirements validation in the context of acceptance testing, i.e., validate whether the finished product satisfies the business requirements. if a business requirement (i.e., something the business needs to do or have in order to stay in business) is to reach one million downloads within six months. The insight provided in the form of estimated popularity can be used to tame expectations through quantifying the uncertainty around the expected user base. Moreover, the explanation provided with each prediction can be utilized as well to provide an additional valuable support to requirements decisions.

Chapter 4

Modeling Explicit Users Feedback

4.1 Introduction

In this chapter, we discuss the findings of our study and present our proposed Bayesian framework that addresses previous work's limitations while being more accurate, efficient, and maintainable than the state of the art.

First, we provide an overview of the current literature in Section 4.2. We show that current research efforts on this task fall under one of two directions: In the first direction, a classification model is constructed to classify reviews into a predefined list of labels that is considered useful for software developers (e.g., bug reports, feature requests, etc) as a way to automate the filtering process [13,33,41,57,58,75,87,88]. However, it was found that assigning such general labels was not enough to extract requirements, as you can easily find thousands of reviews that fall under one of those labels, e.g., feature requests. Thus, a second direction with the goal of summarizing or grouping together user reviews with similar topics for easier requirement extraction was established [11,22]. In this type of research, either a visualization technique is used to highlight the most frequent terms used in those reviews and it is left to the developer to infer the requested feature(s), or a clustering technique is used to group those reviews that discuss a single topic together, and then it is left to the developer to analyze each cluster and make a list of the requested feature(s) . In a more end-to-end research, both the classification and summarization tasks were attempted [27,73,76,99]. We align our work with this direction. However, unlike previous work where the classification and summarization tasks were handled separately, we propose to merge the two tasks together in a single learning process.

Second, in Section 4.3 we discuss the two datasets that were used throughout this chapter for experimentation and evaluation purposes. We provide an overview of the datasets, the process we followed in preparing and labeling each of them, and a summary statistics of their main attributes.

Third, to address ($\mathbf{RQ}_5$) on minimizing information loss when filtering users feedback, we explore the claim discussed in previous work [51, 67] which indicates that when hierarchical relationships exist between classification labels, then leveraging those relationships can improve the classification accuracy. We studied the literature and found that hierarchy does exist between the predefined requirement labels, e.g., feature requests and bug reports are all considered functional requirements. Thus, we investigated the use of such relationships as a way to minimize information loss and boost the recall in Section 4.4. We followed the technique suggested in [67] and evaluated the hierarchical approach on multiple classifiers and multiple datasets. We found that we get an average of 33% increase in the F1 measure when leveraging hierarchical relationships, mostly from a boost in recall. Thus, the use of a hierarchical classification approach can greatly boost recall, which in turn minimizes information loss. In addition, we highlight that using a Bayesian approach allows us to use the top parent classifier's variance provided with its predictive distribution as a way to keep track of the hidden cost of false negatives. A high variance can indicate a lower confidence in our top level classifier, which indicates a potential high information loss, whereas, a low variance can indicate a higher confidence at the top filtering level, which should give us more confidence that information loss is minimized.

Fourth, to address ($\mathbf{RQ}_6$) on how can we improve the representation of users feedback to accommodate its unique language, we first discuss the challenge with representing explicit users feedback in Section 4.5. We explain the short and noisy nature of app reviews due to the high amount of misspelled and alternatively spelled words, and we discuss how the current proposed solutions in the literature use manual steps to address these issues, which are difficult to create and maintain. We then exploit neural network embeddings as a potential better alternative that addresses existing issues and provides better performance and maintainability. We found that the left-to-right language modeling task (i.e., predicting the next missing word) that word embedding models such as Word2Vec [65] and FastText [7] are trained on can group words together in a similar fashion to what stemming and lemmatization can do when used on top of a traditional bag-of-words or TFIDF representation. Additionally, we found that it can successfully group misspelled and alternatively spelled words, which addresses a key limitation of previous approaches. Moreover, to build a more context aware representation, we studied BERT(Bidirectional Encoder Representations from Transformers) method of pre-training language representations [17], which is the current state of the art. By using a pre-trained BERT model to generate embeddings, we can leverage the patterns

it learned on syntax, semantics, structure, and language from training on billions of examples. We found that embeddings generated from the pre-trained BERT can produce decent representations, but they only reach their full potential if the model was fine-tuned on the specific downstream task. We also found that the best representation that offers a balance between performance and maintainability is the one that merges a TFIDF representation with embeddings generated from a BERT model that is fine-tuned on the multi-class and multi-label requirements classification task using the second-to-last average pooling strategy.

Finally, in Section 4.6, we address ($\mathbf{RQ}_7$) on the characteristics of a representative review and whether we can accurately identify such reviews, and ($\mathbf{RQ}_8$) on how accurately we can accomplish the classification and summarization tasks using a single model compared to the state of the art. We described the representativeness of reviews vary in terms of coverage (i.e., number of requirements mentioned) and expressiveness (i.e., ease of requirement extraction). Additionally, we demonstrate that our proposed Bayesian approach can capture 91.3% of the discussed requirement with only 19% of the dataset, i.e., reducing the human effort needed to extract the requirements by 80%.

We summarize our main contributions as follows:

- We discuss the information loss issue due to false negatives and showed how using a hierarchical classification approach can help boost the recall, i.e., minimize information loss, through leveraging the implicit inter-class hierarchical relationship between the labels.
- We show that in addition to learning the same patterns as stemming and lemmatization, embeddings generated from neural network models trained on the left-to-right language modeling task can learn to group the misspelled and alternatively spelled terms that posed a challenge for previous approaches.
- We show that embeddings generated from a fine-tuned BERT model using the second-to-last average pooling strategy can create a space where app reviews with similar requirements are placed closer together in terms of cosine similarity compared to the more common approaches used to represent app reviews such as TFIDF.
- We found that the best representation for both the classification and summarization task is achieved through merging a TFIDF representation with embeddings generated from a BERT model that is fine-tuned on the multi-class and multi-label requirements classification task using the second-to-last average pooling strategy.
- We expand the problem of requirements extraction from only classifying requirements using

predefined labels to identifying the most representative subset of reviews for requirements extraction, which aligns better with the original goal of requirements extraction.

- We propose an end-to-end Bayesian framework that can accomplish both the classification and summarization task using a single model. We conducted comprehensive experiments to evaluate our proposed Bayesian framework and showed that it can produce equal or better results than the state of the art while addressing the issues of reliability and maintainability of previous methods.
- We demonstrate that our proposed Bayesian approach outperforms the state of the art in its ability to identify the most representative subset as it is able to capture 91.3% of the discussed requirement with only 19% of the dataset, i.e., reducing the human effort needed to extract the requirements by 80%.

4.2 Related Work

In this section, we summarize existing studies related to app reviews classification and/or summarization.

4.2.1 Summarizing User Reviews

We can find several works with the goal of summarizing or visualizing the overall topics found in user reviews. In [40] an approach to summarize the most discussed aspects of a product and the opinions of users on them (i.e, positive or negative) is presented. In [11] a topic modeling techniques is exploited to discover the topics found in the reviews along with the sentences that best describe those topics. In [13] a clustering algorithm (DBSCAN) is used to group together similar reviews. In [100], the authors proposed an information retrieval framework that pre-process the reviews and put them in a knowledge database, and then given a set of developer's selected keywords, their framework would return the most relevant reviews that discuss the provided topics. In [22, 27, 76] different visualization tools/techniques are presented. For example, in [76] an HTML tool that can visualize the content of the reviews, e.g., terms plotted as a word cloud, is used. in [22, 101] they focused on providing an interface that summarizes and tracks the change in the volume of reviews under specific topics between different versions to highlight abnormal changes, e.g., version 2 has significantly higher bug reports than all other versions. In [96], the authors mine user opinions on

APIs from user reviews, and provide a search engine to developers to utilize when searching for opinions on APIs.

4.2.2 Classifying User Reviews

As for app reviews classification, in [13], the first attempt to classify app reviews into *informative* and *non-informative* was conducted. The authors used a bag of words representation as it is the case with many other studies as well [57, 58, 99]. In fact, the novel angles that are taken into consideration with every approach is one interesting aspect of the related work. For example, in [99] the authors included N-gram extraction in the creation of the bag of words representation to account for context that require two or three words, e.g, *not laggy*. If we process that term separately, then we wont' understand the actual intention. In [57] the tense of the verb was incorporated into the feature space as the authors argue that verbs in the past are usually associated with users reporting bugs, whereas, verbs in the future are usually correlated with hope and requests for additions (i.e., feature requests). In [75], the authors claim that most reviews follow a specific linguistic patterns, and that identifying those patterns can help with the classification task. Thus, they created 246 linguistic patterns that describe the general form in which a review would be in to fall under a specific label, e.g., *[someone] should add [something]*. In [27] the bag of words representation is replaced with a representation generated from parsing sentences as parsing trees and then traversing the tree to construct the representation. The authors claim this approach can take word semantics into consideration. In [87, 88], the authors suggest to classify on the sentence level instead of the review level to allow for multi-label classification. It is also worth mentioning that some studies investigate connections beyond the classification of app reviews, e.g., in [73] the authors investigate the possibility of linking user feedback to the source code components.

4.3 Data Collection

Table 4.1: Statistics of the used datasets

	Maalej	**Panichella**
Feature Request	252 (7%)	391 (13%)
Bug Report	370 (10%)	271 (9%)
User Experience	607 (16%)	334 (11%)
Total Info	1229 (33%)	880 (30%)
Total Non-Info	2455 (67%)	2024 (70%)
Total Reviews	3684	2904

4.3.1 Datasets:

To address our research questions, we will report results on two real-world datasets that were provided by previous research. The first is the *Maalej dataset* [57,58] where reviews were randomly selected from both Apple and Google Play stores. The authors crawled over a million app reviews and followed a sampling strategy with the goal of picking a stratified and a representative sample (e.g., equal number of free and paid apps, equal number of iOS and Android app, etc). The second is the *Panichella dataset* [76,87] where the authors favored an app specific sampling approach. The dataset contains reviews of 17 apps coming from the Google Play, Microsoft, and Apple app stores. Unfortunately, the ground truth was not provided for this dataset so will need to label this dataset ourselves. Thus, we asked two teams of graduate students to label the dataset separately according to a labelling guide that can be found in Appendix A. The guide follows closely the guidance of the original paper. Once the two teams completed their labelling task, we compared the two labels and went over all disagreements to make sure they receive the appropriate label. The statistics of both datasets can be found in Table 4.1. For the *Panichella dataset*, the sum of the individual labels is greater than the total informative labels as some reviews are assigned multiple labels.

4.3.2 Measuring Representativeness

Data Labeling: To identify the set of the most representative reviews for requirements extraction, we need to first define what aspects and criteria we will use to evaluate each review for representa-

tiveness, and then label the dataset accordingly. We define the most representative set of reviews for requirements extraction as the set with the highest coverage, expressiveness, and endorsement. First, the **coverage of requirements** measures the number of requirements captured by the selected set of reviews. The more requirements captured by the selected set, the more representative the set is considered. Second, the **expressiveness of reviews** describes the ease of requirement extraction from each of the reviews in the selected set. A review that well describes a request/issue with a developer friendly language is more useful than a review with a vaguely described content. For example, a review that simply states a crash occurred is less useful for requirement extraction than a review that describes what the user was doing when the crash occurred. Thus, the more expressive the selected reviews, the more representative the set is considered.

To capture this intuition, we asked two Ph.D students to label each review with two labels as seen in Table 4.2. The first label is *requirement id(s)*, which contain a list of all the mentioned requirements in a review. Creating this label is a two step process. The annotators would first need to establish a unified list of all the discussed requirements for a given app, and then use that list to label each review with a requirement id or more based on its content. We asked each annotator to create his own list of requirements separately after going through the reviews, and then in a group meeting we decided on the final set of requirements to be used for labeling. Once the list was decided, the annotators started the labeling process separately. Once both completed the process, final labels were assigned in a group discussion. Second, the *level of expressiveness* in a review. We decided on four levels and we show examples of each in Table 4.2. Each review will be assigned a number from one to four based on the level of expressiveness. Level one represents a non-informative or barely readable review. Level two represents a review that is somewhat readable but does not provide the minimum expected context for requirement extraction. Level three represents a review that is readable and contains the minimum needed context for a single requirement extraction. Level four represents a superstar review in the sense that it is readable and contains enough context for multiple requirements extraction.

We found a substantial inter-rater agreement (kappa=0.87) between the annotators for the Panichella dataset. However, we decided to exclude the Maalej dataset from this labeling effort because the reviews in the Maalej dataset were randomly selected from the app store with the no connection to the original app. This makes the labeling effort challenging as we need to know that the reviews share the same context (i.e., same app) to assume that they are discussing the same requirement, For example, if one review said "*The UI button is not displaying properly*" and another review said "*The button is cut off*", we cannot assume that they are talking about the same issue as they can be talking about two completely different apps and two different issues. However, if we know that

Table 4.2: Examples of real-world reviews from the Panichella dataset and how they were labeled in terms of representativeness. First column is the list of requirement id(s) that were mentioned in the review. E.g., Requirement ID (4) refers to users requesting additional login options, and Requirement ID (1) refers to a review process requiring from all users to have a specific number of friends on Facebook to use the app. Second column is level of Expressiveness, which is a value between 1 to 4. The least expressive is labeled with 1 and most expressive is labeled with 4.

Review	**Req. Id(s)**	**Expressiveness**
Example 1: Reviews for Blinq (Social Dating Application)		
Blinq Okay	NA	1
Login Facebook? Nope. App immediately deleted	4	2
FB and without FB can Blinq not work?? There must also be an alternative logon options!	4	3
Facebook and many data are required There is no way login without FB account and admin permission... I had ... create account but now I can not get access because I have too few contacts ... !?	1,4	4
Example 2: Reviews for Lifelog (Health and Fitness Application)		
Any chance of an export option so I can open the data from lifelog in Excel and analyse it? ...	1086	3
Still experiencing .. fonts problem. Cant see text.	1068	2
I would like ... to refresh and load my activity ... without having to connect to the network...	1079	3
I like to suggest the following: 1. Allow users to download their tracking data... 2.have the app...work offline w/o needing to sync all time...3. More options to customize the font, color, appearance.. the latest copy of lifelog on my Note 4 Samsung has a transparent font in sub menus .. I cant see anything!)	1068, 1086, 1079	4

the two reviews are referring to the same app, then we can assume that they potentially refer to

the same issue.

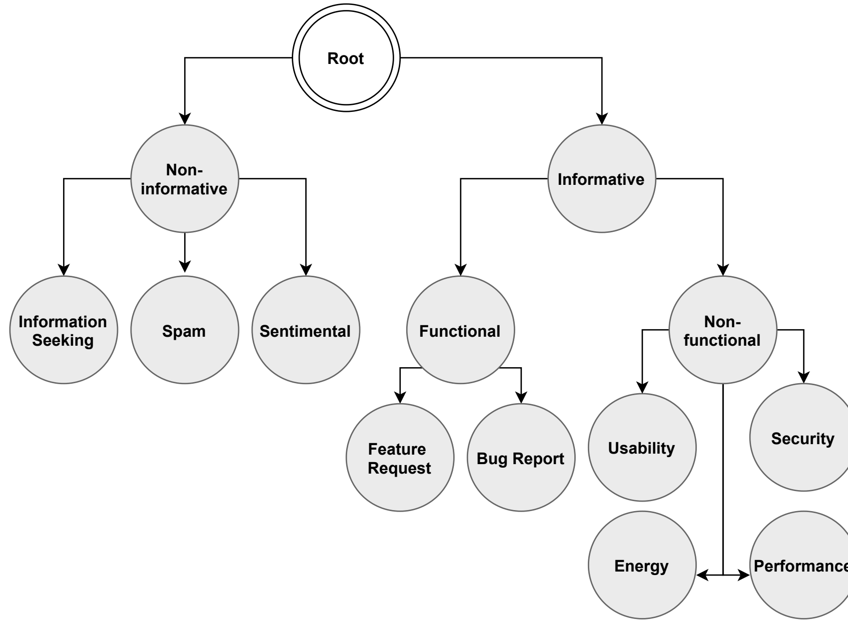


Figure 4.1: The hierarchical structure in app reviews classes

4.4 Minimizing Information Loss of False Negatives

In this section, we present our effort in addressing ($\mathbf{RQ}_5$)on minimizing information loss when filtering users feedback, i.e., minimizing false negatives. We first first discuss the limitation of current approaches. We then explain our proposed approach for minimizing information loss through the use of a hierarchical classification approach. Finally, we evaluate our proposed approach using multiple baselines, classifiers, and datasets on our specific downstream task to determine how much value it adds compared to other approaches.

4.4.1 Simple but Effective: Flat versus Hierarchical Classification

We observed that all the previous work has approached the problem as a flat classification problem. In [13], a binary classifier that determines whether an app review is *informative* or *non-informative* was used, introducing the first two types of classes. A follow up work [58] further studied the app reviews and introduced a new set of labels *rating*, *bug reports*, *feature requests*, and *user experience* reviews. A more recent study used feedback from the industry to further break down the *user experience* label into reviews reporting *security* concerns, *energy* concerns, etc. This increasing number of requirement labels shows the increasing complexity and the level of information that

can be found in reviews, however, we observed two missing aspects. First, there seems to be a hierarchical structure between the predefined set of labels that previous work did not attempt to leverage. Second, while there's a strong emphasis on attempting to classify reviews under each label, there is no emphasis on the level of information loss when working on a problem with many labels. In particular, when all the binary classifiers provide a negative class to a data point, i.e., it is not assigned any class, then it is assumed to be non-informative and filtered out. If this data point is actually informative, then we lost the information it provides in the process, i,e., a false negative. We believe the amount of information loss as part of the automation process of requirements extraction should always be considered and emphasised to ease developers concerns on incorrectly filtering out informative reviews. However, this was not addressed in previous work. We can address both points through the use of a hierarchical classification approach.

Exploiting the hierarchical relationship: In traditional flat classification the hierarchical relationship between the classes is ignored. For example, a binary flat classifier would attempt to distinguish app reviews with *feature requests* from all other classes, i.e., reviews with *non-informative* content, reviews with *bug reports*, etc. This ignores the fact that reviews with *feature requests* and/or *bug reports* are all considered as *informative* reviews, i.e., they share a common parent class. Taking this information into consideration when training the classifier can help us build a better classifier that attempts to first distinguish the *informative* reviews from the *non-informative* reviews as they share common characteristics, and then further classify those *informative* reviews into their appropriate class. Moreover, using this top-down classification strategy, we can achieve a shorter overall training/testing time as we are filtering down the number of training examples with each level, i.e., classes further down the hierarchical relationship tree would train on a subset of the original training examples.

We argue that based on the analysis of previous work, it is clear that the classes of app reviews can be organized into a fairly complex hierarchy as shown in Figure 4.1. It was reported in multiple studies [72] that the *informative* subset of app reviews seem to represent 30%-35% at max of the whole corpus. If we break down the *informative* subset further into multiple classes, we can observe that some classes can be as rare as 5%-10%. As such, using traditional flat classification will create classifiers dominated by the negative class, hence, will not be able to accurately discriminate between the two classes. We believe that this limitation can be addressed when a hierarchical top down classification approach is used. We aim to address the following research question:

- **$\mathbf{RQ}_5a$:** Do we gain any app reviews prediction accuracy from leveraging the existing hierarchical relationships that exist between the predefined requirements labels?

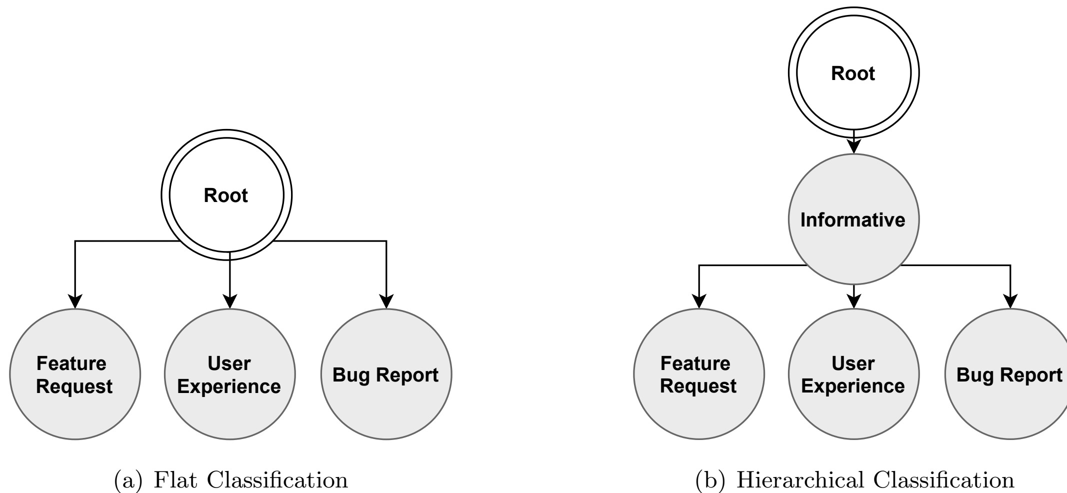
4.4.2 Evaluation and Discussion

In this section, we present our evaluation and discuss our results for our proposed approach.

RQ$_5a$: Do we gain any app reviews prediction accuracy from leveraging the existing hierarchical relationships that exist between the predefined requirements labels?

Experiment Setup: To evaluate the model's accuracy gained from leveraging the hierarchical relationship embedded within the labels, we will use a simple feature space consisting of a bag-of-words representation with term frequency–inverse document frequency (TF-IDF) [82], i.e., the most standard representation. For this evaluation, we will not attempt to add any additional features such as meta-data features (e.g., rating, review length, etc) as we want to focus on the added benefit of hierarchical versus flat app reviews classification. Moreover, to make sure the results are not due to a specific classifier or to a specific dataset, we will evaluate on both the *Maalej* and *Panichella* datasets, and on four different classifiers: Logistic Regression (*L1 Regularization*), Random Forest (*200 trees*), Support Vector Machines (*Linear Kernel)*, Relevant Vector Machines (*Linear Kernel)*, and Naive Bayes (*Multinomial*). However, as we are limited to the three mutual labels (bug report, feature request, user experience) provided with those datasets, we will build the experiment around them. Finally, to make sure both the flat and hierarchical classifiers were exposed to the same set of reviews during training and testing, we used a single train/test split of 80/20 for both. For flat classification, as shown in Figure 4.2(a), we are training three one-vs-rest binary classifiers, one classifier per label (e.g., bug report or not). We prefer to use binary classifiers instead of a multi-class classifier as this setup allows for multi-label classification. This means an app review can be given a single or multiple labels. For example, an app review with multiple labels from the Panichella dataset is "*This is a great app for keeping track of weight ... there should be a way to turn off daily reminder ... also I notice it keeps changing the year I was born...*". However, using this setup, it is also possible for an app review not to be assigned any of the three classes. For that purpose, in Figure 4.2(a) we show a *non-informative* node that captures all such cases. For hierarchical classification, as shown in Figure 4.2(b), we use a top-down approach for training and classification purpose. At the first level, we are using a binary classifier that classifies all app reviews as either *informative* or *non-informative*, and on the second level we use three one-vs-rest binary classifiers that attempt to further classify what passes as *informative* under one or none of the three classes (i.e., bug report, feature request, user experience). Thus, in hierarchical classification we are training one more classifier than flat classification. This may seem as added complexity, however, the top down approach actually has a better overall computational cost because only the

informative classifier is trained on all the training examples, the remaining three classifiers train only on the *informative* subset. For example, if we had a training data set of 10k app reviews, 3k of those are informative, then the first level classifier will train on all 10k app reviews, but the second level will only have to train on the 3k app reviews. Whereas, in flat classification, each of the classifiers would need to trained on the complete 10k dataset.



(a) Flat Classification (b) Hierarchical Classification

Figure 4.2: Evaluation of flat and hierarchical app reviews classification. On the left, we have three binary classifiers, one for each label. In this setting, each classifier is working on its own. On the right, we have four binary classifiers where the parent classifier identifies informative reviews, and then passes the informative subset to the second level where we have the three children binary classifiers, one for each label. In this setting, the children classifiers are leveraging the parent's classifiers collective knowledge.

Experiment Results: We report the average AUC computed from precision and recall (AUC_{PR}), macro F1 (MF_1), and macro recall (MR) in table 4.4. We can make a couple of observation. First, Naive Bayes seem to outperform the other classifiers when a simple bag of words model is used, which was also observed in a previous study [58], because a term count representation aligns perfectly with how Naive Bayes works. Second, overall, formulating the problem using hierarchical classification increases the model's accuracy, especially with recall (i.e., increases the chance that we do not miss any informative app reviews). On Maalej dataset, we observed on average a 8.4% better AUC_{PR}, 49.8% better F1 measure, and 108% better recall. Similarly on Panichella dataset we observed 13% better AUC_{PR}, 17% better F1, and 33% better recall. To better understand the results, we analyzed the performance of Random Forest on the Panichella dataset where the recall had an improvement of 61%. It's important to mention that in app reviews classification, the ability to label all existing *informative* reviews correctly (i.e., recall) is more important than

Table 4.3: Classification results of flat and hierarchical app review classifiers on Maalej Dataset

Classifier	**Flat Approach**			**Hierarchical Approach**		
	AUC$_{PR}$	**M**F_1	**M**R	**AUC**$_{PR}$	**M**F_1	**M**R
Logistic Reg.	0.349	0.369	0.381	0.393 (+12%)	0.433 (+17%)	0.562 (**+47%**)
Random Forest	0.399	0.195	0.136	0.433 (+8%)	**0.531** (**+172%**)	0.603 (**+343%**)
SVM	0.346	0.358	0.385	0.353 (+2%)	0.423 (+18%)	0.561 (**+45%**)
Naive Bayes	0.458	**0.474**	**0.529**	0.497 (+8%)	0.507 (+7%)	**0.623** (+17%)
RVM	**0.459**	0.375	0.309	**0.514** (+12%)	0.505 (**+35%**)	0.591 (**+91%**)

Table 4.4: Classification results of flat and hierarchical app review classifiers on Panichella Dataset

Classifier	**Flat Approach**			**Hierarchical Approach**		
	AUC$_{PR}$	**M**F_1	**M**R	**AUC**$_{PR}$	**M**F_1	**M**R
Logistic Reg.	0.622	0.599	0.594	0.699 (+12%)	0.681 (+13%)	0.731 (**+23%**)
Random Forest	**0.739**	0.541	0.428	0.768 (+4%)	0.699 (**+29%**)	0.692 (**+61%**)
SVM	0.482	0.523	0.572	0.625 (**+30%**)	0.617 (+17%)	0.701 (**+22%**)
Naive Bayes	0.681	**0.630**	**0.624**	0.768 (+13%)	**0.705** (+10%)	0.736 (+17%)
RVM	0.686	0.591	0.512	0.734 (+7%)	0.702 (+18%)	**0.747** (**+45%**)

mis-classifying a few *non-informative* reviews as *informative* (i.e., precision) because all reviews labelled as *non-informative* are usually disregarded (i.e., feedback would be lost with low recall). Thus, this significant improvement on the recall when using a hierarchical approach is a perfect

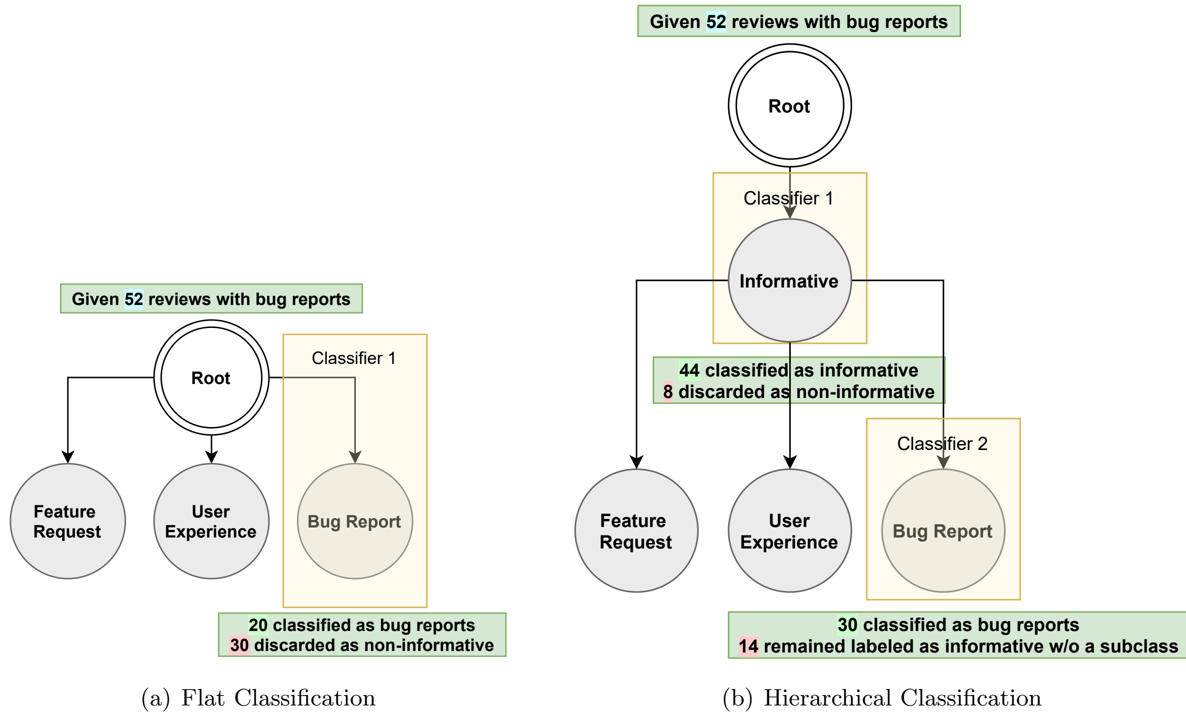
match with the app reviews classification problem, and such would be our focus in this section.

Table 4.5: Analyzing Random Forest: The flat vs. hierarchical classifier on the Panichella dataset.

Type	**Informative**	**Feature Request**	**Bug Report**	**User Experience**
	Recall Measure			
Flat	0.517	0.666	0.385	0.145
Hierarchical	0.727	0.831	0.682	0.526

We report in Table 4.5 the recall of each classifier. In flat classification, we can observe that the classifier's ability to correctly classify all the *bug report* and *user experience* instances is quite poor. As we believe the *bug report* is more a critical category, we further investigated the instances and how they were labelled in both classifiers as shown in Figure 4.3. In our experiment, the testing sample had 52 app reviews with bug reports. In the case of flat classification, we clearly observe that the classifier missed 32 of the bug reports (62%). However, the hierarchical classifier mislabelled 8 bug reports out of the 52 as *non-informative*, and mislabeled 14 bug reports out of the 44 *informative* reviews as *other* type of informative reviews. Overall, the classifier mislabelled 42% of the bug reports, a much better recall than the flat classifier. Upon further checking, we can observe that the first level performance in the hierarchical classifier is excellent as we were able to capture 85% of the bug reports as informative reviews. However, the second level performance was not ideal (i.e., missed 14 out of 44), but we can argue that it is still better than the flat classifier as we were still able to label those app reviews as *informative*, i.e., they were not completely missed, but were incorrectly classified as *other* type of informative reviews.

We credit the hierarchical classifier better performance to two main factors. First, it is not affected as much by the class imbalance as the flat classifier. In the case of flat classification, the frequency of each class is dominated by the negative class, e.g., the *bug report* classifier had 91% instance of the negative class as it would need to distinguish itself from the *non-informative* and other *informative* classes which is quite challenging. However, in hierarchical classification, the first level uses the combined knowledge from all three classes to first filter out *informative* from *non-informative* app reviews, which is an easier task, i.e., due to the different nature of *non-informative* reviews from *informative* and due to having a much higher positive class frequency. Second, we observed that, e.g., the *bug report* classifier can distinguish itself better from other *feature request* and *user experience* reviews (i.e., informative reviews) when *non-informative* reviews are removed,



(a) Flat Classification (b) Hierarchical Classification

Figure 4.3: Given 52 app reviews with bug reports, how were they classified in flat vs hierarchical? We can observe that on the *flat classifier* we were able to identify only 20 out of the 52 existing bug reports, i.e., we captured only 38% of the information on reported issues/bugs. On the other hand, with a *hierarchical classifier*, were able to capture 44 out of the existing 52 reviews with bug reports as informative using the parent classifier, i.e., we captured 85% of the reported issues/bugs, which is a significant minimization of the information loss. Furthermore, using the child classifier, we labeled 30 out of the 44 informative reviews correctly as bug reports. The remaining 14 reviews received an informative label, but did not receive any subclass.

which is what the hierarchical top-down classification is inherently doing, i.e., leveraging hierarchical structure.

4.5 Representing Explicit Users Feedback

In this section, we present our effort in addressing ($\mathbf{RQ}_6$) on how can we improve the representation of users feedback to accommodate its unique language. We first first discuss the challenges and limitation of current approaches in representing explicit users feedback. We then explain our proposed approach for constructing a representation that would address current limitations and provide better model performance on our specific task. Next, we discuss the details of how we trained and implemented our proposed approach. Finally, we evaluate our proposed representation using multiple baselines, classifiers, and datasets on our specific downstream task to determine how much value it adds compared to other approaches.

4.5.1 The Short and Noisy Nature of Explicit Feedback

It was observed in [58] that classifying the reviews coming from the iOS app store was significantly more accurate than those coming from the Android store. They attribute this difference to the language and vocabulary difference from those two app stores. They claim that the iOS store reviews were less noisy (i.e., had less typos) and used a much more homogeneous vocabulary of terms. This observation highlights the effect of the noise found in user reviews on the classification task and the impact it has on the learning task. In [100], this observation was studied further as the authors also highlighted and described the observation that app reviews suffer from a high percentage of typos (misspelled words), acronyms, and abbreviations. They performed a preliminary analysis of 300,000 reviews and compared their textual content against an English dictionary of 150,000 common words, and found that a large portion of the used words in app reviews do not match any words in the English dictionary, i.e., due to abbreviations and typos. Having such high noise and unique language (e.g., *wait* is written as *w8*) creates an issue for traditional data mining techniques that relies on stemming and dictionary creation as both *wait* and *w8* will still exist as two unique different words. They hypothesized that this observation might be due to the fact that reviews are written using mobile devices which lack a physical keyboard, hence, it is more likely to have typos, acronyms, and abbreviations. To overcome this issue, the authors in [100] manually created a custom dictionary that attempts to replace the most frequent out-of-dictionary words with their dictionary-equivalent (e.g., replace *exelent* with *excellent*). Moreover, in [27], a similar observation was made, and the authors manually constructed a collection of 60 different typos and contractions, and replaced them using regular expression. We have observed a similar pattern of noise with app reviews where a large portion of words in the post-processing and stemming dictionary seem to represent the same word but written differently due to misspelled words (e.g.,

fantastic vs *fantastick*) or alternatively spelled words either for abbreviations purposes (e.g., *thanks* vs *thx*), or to represent a stronger emotion, (e.g., *loved* vs *loooved*). In Table 4.6, we show a few examples. We believe this might be more common in app reviews than in other forms of social media (e.g., Tweets) because of the difference in public exposure. People tend to be more conscious about what they write on their social media profiles (e.g., Twitter, Facebook) as that content is shared with their circle of family and friends, which is not the case with app reviews. Thus, people put less effort into checking their review for grammar or spelling mistakes. We agree with prior work

Table 4.6: Examples of misspelled words or alternatively spelled words

Term	**Observed noise**
amazing	amaazing, amaaazing, amassing, amazeng
thanks	thx, thanx, tx, tnx, 10x, thnx, ty
wait	wt, w8, waait, w8t
awesome	awasome, awesomeeee, awsome, owesome, asssome
love	lov, luv, lovve, looove, loveee
because	bc, b/c, cuz, coz, bcz, caus

that merging misspelled or alternatively spelled words would improve the textual representation and the model's performance overall. However, we argue that using a manually created custom dictionary would be too difficult to create and maintain overtime.

4.5.2 Proposed Representation

It was shown in previous literature [58] that most of the accuracy comes from the textual representation, i.e., bag-of-words or other representations that builds upon the text of the review. This highlights the inherent valuable information that exist in the text of the review. As such, any issues that exist with textual representation can hinder any model, and vice versa, any improvement that can help better represent such information would be valuable. We studied how previous work created a representation for the textual content of the users feedback and found that most applied traditional text preprocessing techniques that is used in data mining to prepare the text, i.e., stemming or lemmatization, and then used a bag-of-words representation or a TFIDF representation. While this is a reasonable proof-of-concept representation, it comes with many limitations. For example, it does not capture word semantics, structure, or context, which is is an important aspect due to the unique and noisy nature of explicit users feedback.

In recent years, embeddings generated from neural network models revolutionized the natural language processing (NLP) space. For example, the word2vec [65], the GloVe [79], and the FastText [7] are all techniques that learn representations from training on a left-to-right language modeling task (i.e., predicting the next missing word) over millions of examples. The assumption is that the final representation learns word semantics and meaning. These techniques are built on the notion that words with similar semantic meaning will have the same set of words around them. For example, the words *love* and *like* are used in similar manners, i.e., *I love that app* and *I like that app.* As a result, they would be closely placed in the embedded space as they share a similar semantic meaning.

Following up on this work, researchers started expanding on neural network architectures to enrich the embeddings and they had great success. The most notable being the Transformer's architecture [98], where the authors introduced the attention mechanism where certain tokens are emphasised more than others tokens during the training instead of giving all tokens in a sentence the same level of attention. Additionally, the language modeling task was expanded from a left-to-right modeling task to a bidirectional masked language modeling task. The most successful Transformer-based model is BERT(Bidirectional Encoder Representations from Transformers) [17], which we believe to be the current state of the art.

Leveraging Embedded Representations: Traditional text preprocessing techniques such as stemming/lemmatization are powerful techniques used to merge words with similar meaning together, but they suffer from out of dictionary terms that are either completely new terms, mis-

spelled, or alternatively spelled terms, which are present heavily in online users feedback. This can greatly affect the representation and the pattern learning of machine learning models. Alternatively, embeddings generated from neural network models trained on masked language modeling (i.e., predict the missing word) claim to learn many things on language modeling, one of which is word semantics. We propose to study whether such approaches complement or replace the traditional stemming/lemmatization techniques (i.e., do they learn similar or different patterns). Additionally, whether they can address the issues of the noisy and unique language used in online users feedback. Moreover, whether they provide better contextual understanding of such short text that is crucial for distinguishing feedback with different requirements/topics. As a result, we propose to exploit neural network embeddings as a potential better alternative that addresses existing issues and provides better performance and maintainability through the analysis of the following research questions:

- $\mathbf{RQ}_6a$**:** Traditional text preprocessing techniques such as stemming/lemmatization are used to merge words with similar meaning together, does the masked language modeling approach complement or replace these techniques (i.e., do they learn similar or different patterns)?
- $\mathbf{RQ}_6b$**:** Can embeddings trained on masked language modeling provide a better representation for grouping reviews with similar requirements (i.e., provide a representation with a better context understanding)?
- $\mathbf{RQ}_6c$**:** What is the best representation for our specific classification and summarization tasks that offers best balance between performance and maintainability?

4.5.3 Training a Word Embedding Representation

To address $\mathbf{RQ}_6a$, we need to investigate the patterns learned using the MLM objective and whether it addresses the limitations we discussed. As such, we looked at word embedding techniques and found the most prominent to be the Facebook *FastText* model [7] as it is not only trained using the MLM objective, but it also expanded previous approaches in that it trains on tokens generated from subwords, which allows it to generate embeddings even to never-seen-before terms. In this subsection, we will discuss the training process used to create the word embedding model used in our experiment.

We trained a FastText [7] model on 1,673,672 app reviews collected from [77] and [24]. First, the training method of either continuous bag of words or skip-gram. We selected skip-gram because

.. Second, the vector's length. The larger the vector's length, the larger the information it can capture, and the larger the training data needs to be. A popular choice for length is a value between 100 and 300. In our analysis, we observed that as we increase the vector's length beyond 100, the cosine similarity between words increases, and the performance on the downstream task decreases. Thus, we found a vector of length 100 is a good choice for our case. Third, the range of size for subwords. A popular choice is 3 and 6. In our own analysis, we experimented with a few different ranges, but we did not observe a difference in the downstream task. we found the 3 to 6 range to offer a balance between performance and training time.

4.5.4 Adapting the Bidirectional Encoder Representation of BERT

To address $\mathbf{RQ}_6 b$ and $\mathbf{RQ}_6 c$, we need to investigate how to best leverage the success with transfer learning and the effect of using a bidirectional masked language objective (MLM) with a Next Sentence Prediction (NSP) learning objective on providing better context understanding. For that purpose, we reviewed the current state of the art approaches that can help in accomplishing this task and determined that BERT [17] is the best choice. In this subsection, we will discuss the process we used to exploit BERT for the purpose of generating the best sentence embeddings for our specific problem.

To generate the embeddings, we can directly use BERT's pre-trained model, which was trained on billions of sentences from a large books corpus and a Wikipedia's corpus. In such setting, the embeddings are expected to provide the best generalization. However, the original BERT paper [17] suggested that a fine-tuning of the model on the downstream task or a related task can improve the performance. They proposed to fine-tune the pre-trained BERT model by attaching a classification layer with a softmax activation function at the end of the architecture, and then train the model on the downstream task. Once training is done, then sentence embeddings can be extracted from the network based on the selected strategy of pooling, e.g., averaging across all transformer layers. This approach is not expected to drastically change the embeddings, but it is expected to fine-tune it enough for a measurable improvement on the downstream task. Additionally, the authors in [83] suggested that fine-tuning the pre-trained BERT model using a Siamese and triplet network structures that train on learning similarity between sentences can generate sentence embedding vectors that are richer and more appropriate for sentence similarity comparisons within a vector space. This fine-tuning approach is in particular interesting as we want the selected representation to help us in grouping reviews with similar requirements together.

We evaluated multiple options for the purpose of exploiting the best way to use BERT for our

specific problem as follows: First, a **pooling strategy**, i.e., the embeddings extraction strategy. The authors of the original BERT paper [17] experimented with multiple strategies. For example, using embeddings from the last hidden layer only, or from the concatenation of the last four layers, or from the sum of the twelve hidden layers, etc. The found that all were reasonable ways to extract the embeddings, but it turns out using embeddings extracted from the layers in the first half of the architecture only (e.g., the first hidden layer) produced inferior result compared to using embeddings extracted from the second half of the architecture (e.g., last 4 hidden layers). Overall, they reported the strategy that produced the best results for their task was extracting embeddings from the last four hidden layers. Following up on their a work, a more in-depth analysis on [108] found that different layers capture different information. The first few layers seem to capture low level information and might be too noisy on their own, whereas, the last few layers capture a higher level of information that is producing better general representations. The author recommended approach is extracting embeddings from the second-to-last layer. This is because the last layer is too close to the training output which makes its representation biased to the training targets. Therefore, to identify the best pooling strategy for our specific problem, we compared embeddings generated from the following strategies:

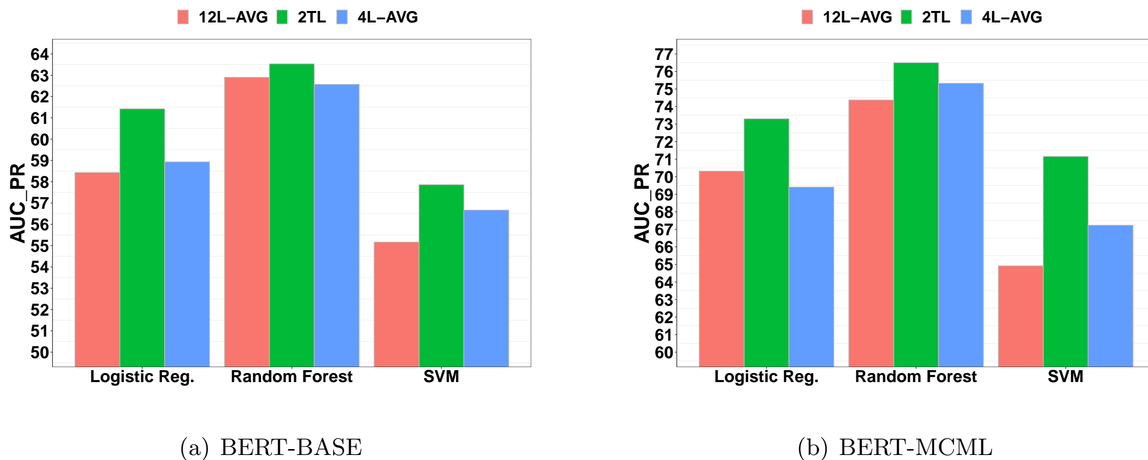
- **The second-to-last Layer (2TL)**: We used the second-to-last hidden layer to extract an embedding per token, and then we averaged across all tokens to represent a document.
- **The Last Four Layers (4L-AVG)**: We averaged the last four hidden layers to extract an embedding per token, and then we averaged across all tokens to represent a document.
- **The 12 Layers (12L-AVG)**: We averaged over all the 12 layers to extract an embedding per token, and then we averaged across all tokens to represent a document.

Second, **the fine-tuning strategy**. It is not clear what is the affect of using different downstream tasks to fine-tune BERT on the final embeddings representation. As such, we created the following models to evaluate whether a difference exists:

- **BERT-BASE**: The pre-trained BERT model with no modification.
- **BERT-INFO**: The pre-trained BERT model fine-tuned on the downstream task of informative binary classification (i.e., review is informative or not).
- **BERT-MCML**: The pre-trained BERT model fine-tuned on the downstream task of multi-class and multi-label classification of the three classes feature request, bug report, and user experience.

- **BERT-SB**: The pre-trained BERT model fine-tuned using Siamese and triplet network structures with a pooling operation added to the output of BERT to derive a fixed-sized sentence embedding vector. This fine-tuning model is based on [83] work in trying to build a richer sentence embeddings that is more appropriate for sentence similarity comparisons within a vector space.

We evaluated the different strategies on a multi-class multi-label requirements classification setup using multiple different classifiers. The results for the Panichella dataset are shown in Figure 4.4. We found that extracting embeddings from the second-to-last layer showed between 2% to 4% improvement over embeddings extracted from the last four hidden layers or the full 12 hidden layers. Also, we present our results for evaluating the best fine-tuning strategy in Figure 4.5. We found that fine-tuning BERT on the specific downstream task provided the best result ad BERT-MCML consistently provided the best performance. We observed as well that while BERT-SB provided better results than BERT-BASE, it is far behind BERT-MCML. This concludes that using BERT-MCML with a 2TL pooling strategy provides the best BERT representation for our specific task.



(a) BERT-BASE

(b) BERT-MCML

Figure 4.4: Evaluating the best pooling strategy on the Panichella dataset. We extracted embeddings from two different BERT models and evaluated the embeddings on multiple classifiers in a multi-label multi-class setup of requirements classification. We can see that the 2TL strategy consistently provided better results.

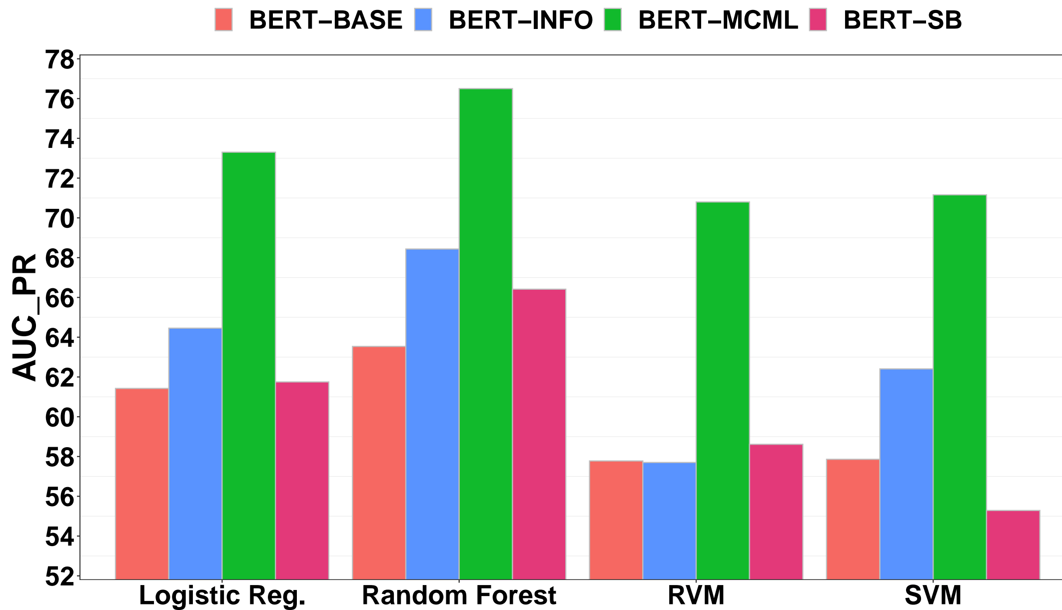


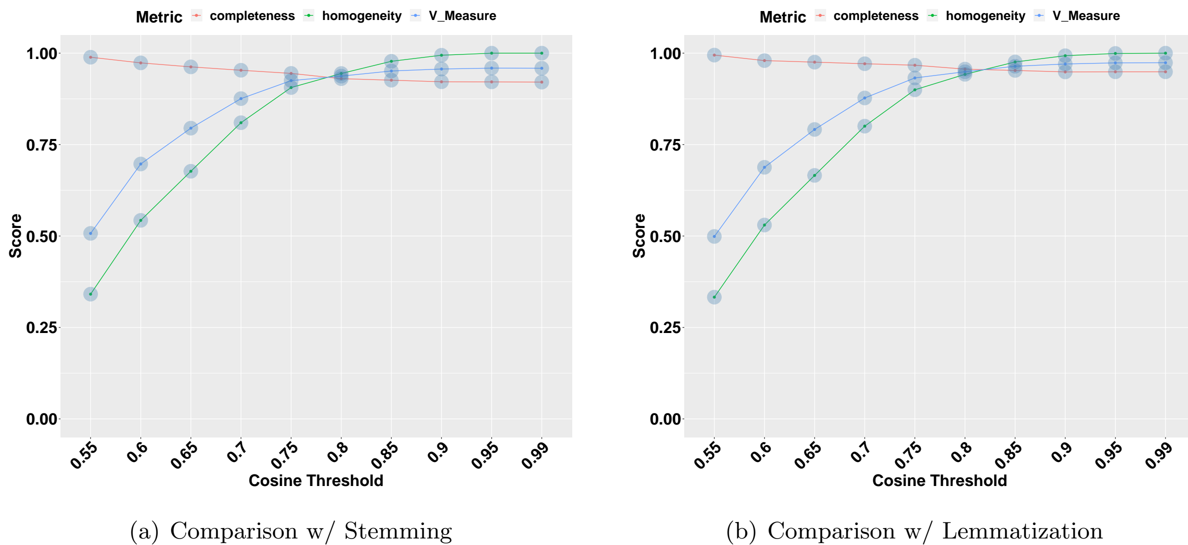
Figure 4.5: Evaluating the best fine-tuning strategy on the Panichella dataset. We extracted the embeddings using the 2TL pooling strategy, which we determined to be the best, and evaluated the classifiers on a multi-label multi-class setup of requirements classification. We can see that fine-tuning BERT on the MCML task consistently provided better results, whereas, the original BERT-BASE with no fine-tuning is the least performing model.

4.5.5 Evaluation and Discussion

In this section, we conduct our evaluation and discuss the results for our proposed approach. To address RQ_6b and RQ_6c, we will compare the following representations and their mixture:

- **TFIDF**: We preprocessed the text using standard data mining techniques (i.e., removing stop words, applying lemmatization) and then created a representation using TFIDF with a minimum term frequency of five. We included bigrams and trigrams. This representation should capture all the main terms and their importance based on their frequency in the corpus.
- **LDA**: We used the TFIDF representation created from the last step to generate a representation using LDA. We set the number of topics to 85, which we determined using cross validation. This representation should provide a higher level representation that captures the main discussed topics instead of focusing on specific terms.

- **FT**: We generated embeddings from the FastText model we trained in section 4.5.3. We averaged embeddings to represent a review. This representation should also provide a high level representation that captures the main discussed topics, but it should be better at handling misspelled and alternatively spelled terms that TFIDF and LDA.
- **BERT**: We generated embeddings from the BERT-MCML model we trained in section 4.5.4. We use embeddings from the second-to-last layer and averaged embeddings to represent a review. This representation should have the main advantages of the FastText model but with a richer sentence context understanding.



(a) Comparison w/ Stemming (b) Comparison w/ Lemmatization

Figure 4.6: A comparison between the merged terms using the traditional methods of Stemming and Lemmatization versus the merged terms using the embeddings. The plots show the score for completeness, homogeneity, and vmeasure across various cosine similarity thresholds.

RQ$_6a$:Traditional text preprocessing techniques such as stemming/lemmatization are used to merge words with similar meaning together, does the masked language modeling approach complement or replace these techniques (i.e., do they learn similar or different patterns)?

Experiment Setup: To address this research question, we will use embeddings generated from the model that we trained on a large corpus of app reviews in Section 4.5.3. We will treat both techniques as a way to merge words with similar meaning together and compare the two lists.

We will first create a shared dictionary of the original unique words. Next, we will apply stemming/lemmatization to each term and consider those terms that share the same *base term* as a merged set. Every term in the merged set will be assigned the same cluster id. For example, in a traditional preprocessing setting, the words *waits, waited, waiting* would be replaced with the base term *wait*, i.e, we can say the four terms were merged together. We want to measure how closely those terms would be placed in the embedded space to determine whether embeddings would cluster such terms together (i.e., *waits, waited, waiting* would be clustered with the base term *wait*). Specifically, for each unique term in our dictionary, we query our embedded space for its location and create a cluster of the term and all the terms that exist within a given t cosine similarity. As such, we would have a cluster id assigned for each term. We will then compare the two sets and evaluate their similarity using the standard clustering evaluation metrics [86] of *Homogeneity* (each cluster consists of a single class), *Completeness* (observations that belong to the same class ended up being in the same cluster), and *V-measure* (harmonic mean between the two).

Experiment Results: Analyzing Figure 4.6, we can observe that the behaviour with stemming and lemmatization is closely similar. Evaluating completeness, we can observe that its score is mostly high and is decreasing as we increase the value for the threshold t. Whereas, homogeneity starts low and increases as we increase threshold t. This makes sense as with a low threshold t the embededd space clusters are expected to be big and they become smaller and more concise as we increase the cosine similarity threshold value. We can also observe that using a cosine similarity threshold of 0.8 offers the best balance between all three metrics. At t=0.8, we get a score of roughly 94% across all three metrics. Therefore, we can conclude based on the results that the clusters constructed from the embedded space closely positioned all the terms that stemming and lemmatization grouped together. Thus, we can say that it learned all the patterns that stemming and lemmatization encapsulate.

Figure 4.7: Showing examples of terms in the embedded space and all the terms around it within a 0.8 cosine distance. We can observe that we were able to capture misspelled and alternatively spelled terms.

Furthermore, we investigated the content of the clusters and show examples in Figure 4.7. We can

observe that the embedded space captured more than the terms merged with traditional stemming and lemmatization approaches to include the misspelled and alternatively spelled terms as well, which posed a challenge to the traditional approach. Thus, we concluded that the embedded space is a suprior approach that not only can replace the traditional approaches of stemming and lemmatization, but also address the limitation of misspelled and alternatively spelled terms.

$\mathbf{RQ}_6 b$: Can embeddings trained on masked language modeling provide a better representation for grouping reviews with similar requirements (i.e., provide a representation with a better context understanding)?

Experiment Setup: We will evaluate the different representations on their ability to group reviews with similar requirements together. One way to measure how close or far specific points within a space, is with the use of cosine similarity. In this experiment, we plan to measure the average cosine similarity between points that mention the same requirement and compare the global averaged similarity across different representations. We will use this as a measure to determine whether such embeddings can create a space where reviews with similar requirements are placed closer together or further apart than other approaches. To accomplish this task, we will use the labels we created for each review, as discussed in Section 4.3, where each review was assigned the list of requirement ids that it mentioned in its text. For the sake of simplicity, we will assign reviews that discuss multiple requirements only one of them, i.e., to have one requirement per review. We will assign the requirement with the highest endorsement, i.e., the requirement that is most discussed. Once data is prepared, we will calculate the averaged cosine similarity between the reviews of each requirement, and then calculate the global averaged cosine similarity across all requirements for a given representation. Additionally, we will select a subset of requirements and visualize how their reviews are placed in the space using t-Distributed Stochastic Neighbor Embedding (t-SNE) [60], which is a commonly used approach for visualizing data in high dimensional spaces. We aim to visually study how the TFIDF representation, the most common representation, places reviews with similar requirements, in terms of distance, compared to the FT representation and/or BERT representation that is trained on the masked language modeling task.

Experiment Results: We report our results in Figure 4.8 for the Panichella dataset. Surprisingly the most noisy representation is TFIDF. We believe for the task of grouping reviews with similar requirements, a term level representation might be too low level to be meaningful. It is highly affected by the noise. This idea is further supported with the observation that a higher level representation such as LDA, which is a topic level representation, is outperforming TFIDF by a

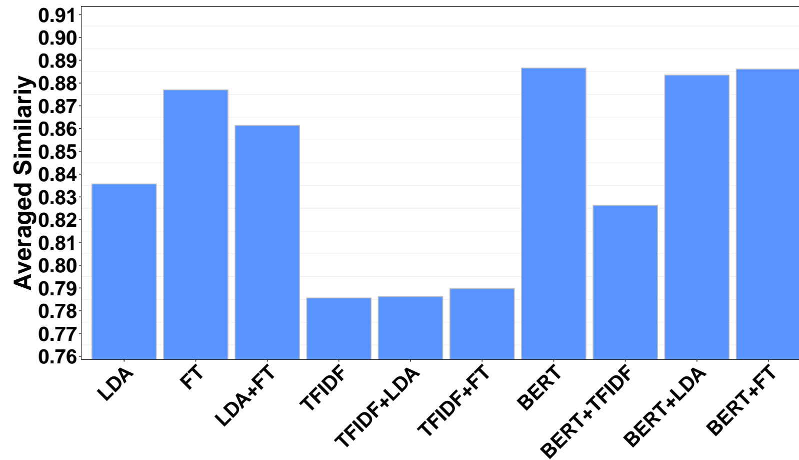


Figure 4.8: Evaluating how closely are reviews with similar requirements placed under each of the representations. We can observe that the embeddings generated from masked language modeling techniques are significantly better than other approaches.

good margin. The best performing representations for the task are clearly the embeddings generated from the FT and BERT models. For example, the FT representation created a space where reviews with similar requirements are 12% more closer to each other than the TFIDF representation.

We further investigated this observation in Figure 4.9, where we plotted all the data points in the Panichella dataset using t-SNE with perplexity of 40. The intuition behind the perplexity parameter is that it is a rough estimation for the expected the number of close neighbors each point has. According to the original authors of t-SNE, the performance should be fairly robust to changes in the perplexity, and the recommended range is 5 to 50 [60]. We experimented with values between 5 to 100, and found that the same patterns are observed with all these values. The only observed difference between the different perplexity values is the spread of the data points in the new space. We picked 40 as it seems to show the clearest picture for how points are placed in the space. In addition to plotting the reviews, we randomly selected five requirements and their reviews and highlighted each with a different color. The selected requirements and their color coding is shown in Table 4.7. Looking at Table 4.7, we can observe that the averaged cosine similarity between the reviews under each requirement is substantially closer under the FT representation compared to the TFIDF representation. Additionally, looking at Figure 4.9, we can observe that in the TFIDF representation, the data points are spread all over the place for all the five requirements. For example, the data points highlighted with a black marker, which correspond to users requesting additional game levels, can be seen spread in all the directions in the TFIDF representation. On the other hand, we can observe the same data points are well concentrated in the bottom left side

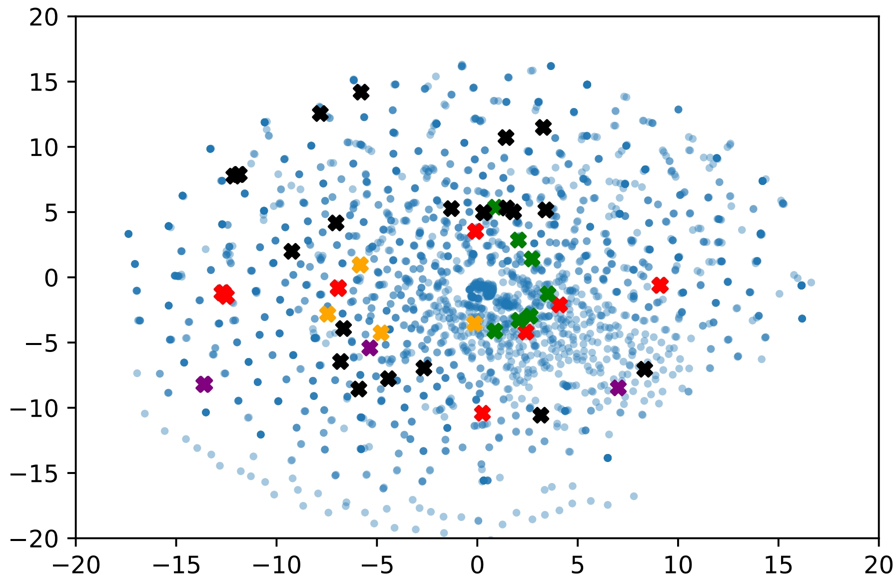
Table 4.7: The subset of requirements selected to be manually evaluated in terms of their placement under the TFIDF representation and the FT representation. We can see their visual placement as per their color code in Figure 4.9. Additionally, we calculated the average cosine similarity between the reviews of each requirement under each representation. We can observe that the reviews of each requirement are placed significantly closer to each other in the FT representation.

Requirement	**#Reviews**	**Color**	**Sim(TFIDF)**	**Sim(FT)**
Request for additional login options other than Facebook.	16 Reviews	Red	0.2318	0.5559
Review process requires too many Facebook friends.	7 Reviews	Green	0.2406	0.6523
Add more levels to the game	23 Reviews	Black	0.1211	0.4844
Add option to select theme	7 Reviews	Orange	0.4399	0.7823
Allow forward and rewind gestures	6 Reviews	Purple	0.4109	0.5492
Averaged Cosine Similarity			0.2889	0.6048

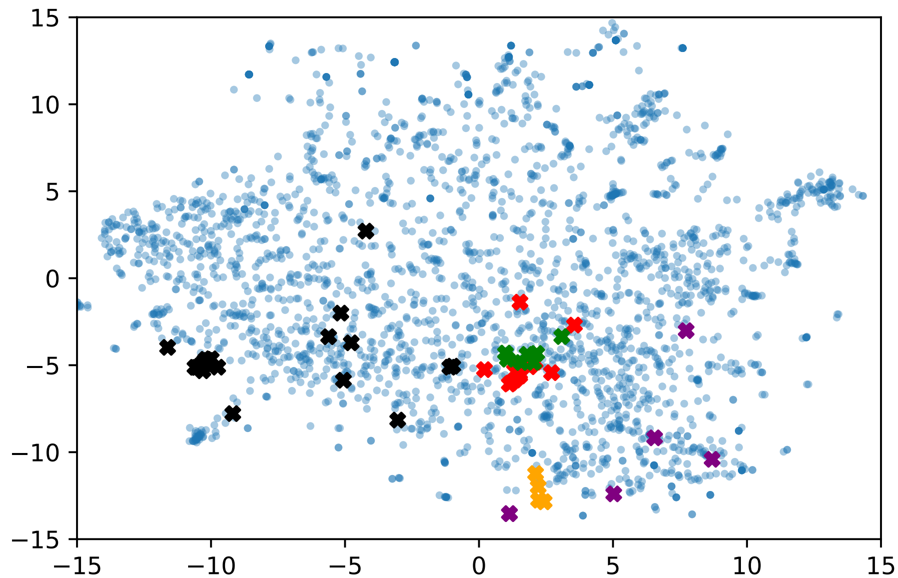
of the FT representation, i.e., they are grouped closer to each other. In fact, the reviews under each requirement are better grouped under the TF representation. The only overlap observed is between the green and red data points. This is due to the nature of the requirements. They both discuss an aspect of Facebook integration that the app uses. The first is a request for additional login options other than Facebook. The second is criticism that the app requests the users to have a specific number of friends on Facebook to create an account. Unlike the other requirements, those two share similar language, which is the reason behind their high overlap within the space.

$RQ_{6}c$: What is the best representation for our specific classification and summarization tasks that offers best balance between performance and maintainability?

Experiment Setup: First, to evaluate the best representation for the classification task, we will evaluate the different representations using multiple dataset and multiple classifiers, i.e., to make sure performance is not due to a specific dataset or classifier. We will report results for Logistic Regression (*L1 Regularization*), Random Forest (*200 trees*), Support Vector Machines (*Linear Kernel)*, Relevant Vector Machines (*Linear Kernel)*, and Naive Bayes (*Multinomial*). For each



(a) TFIDF Representation w/ Avg Sim = 0.289 between points w/ similar requirements



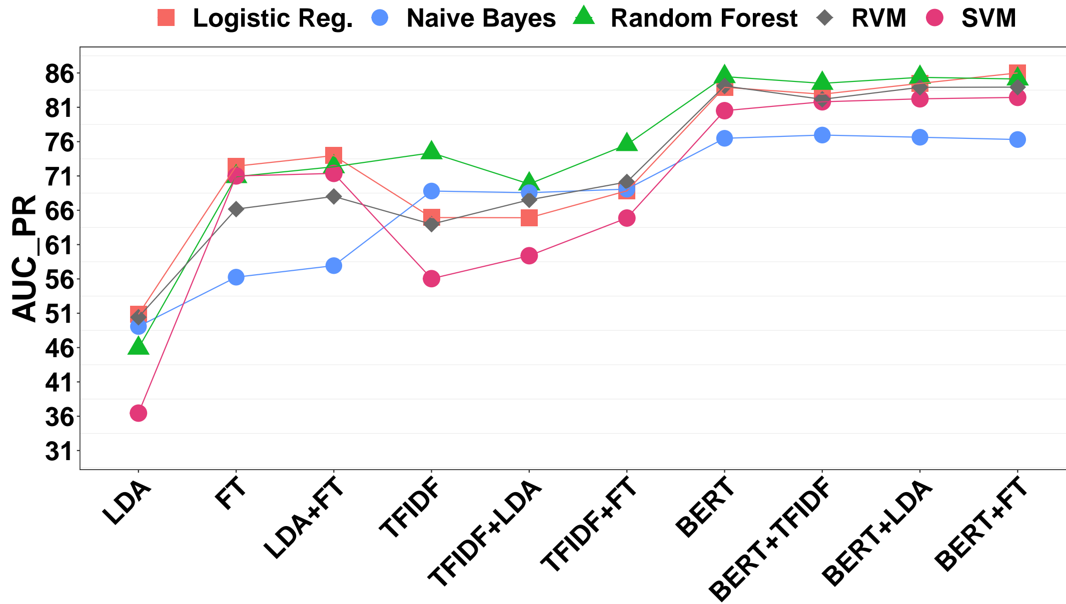
(b) FT Representation w/ Avg Sim = 0.605 between points w/ similar requirements

Figure 4.9: Evaluating the embeddings space that is created through the masked language modeling technique against a TFIDF representation in terms of their ability to place reviews with similar requirements closer to each other. The blue points represent all the data points in the Panichella dataset. The colored points represent six subsets of reviews where each discuss a similar requirement. We can observe that in the TFIDF representation, the groups are spread across the space, whereas, in the embeddings representation they are placed closely together.

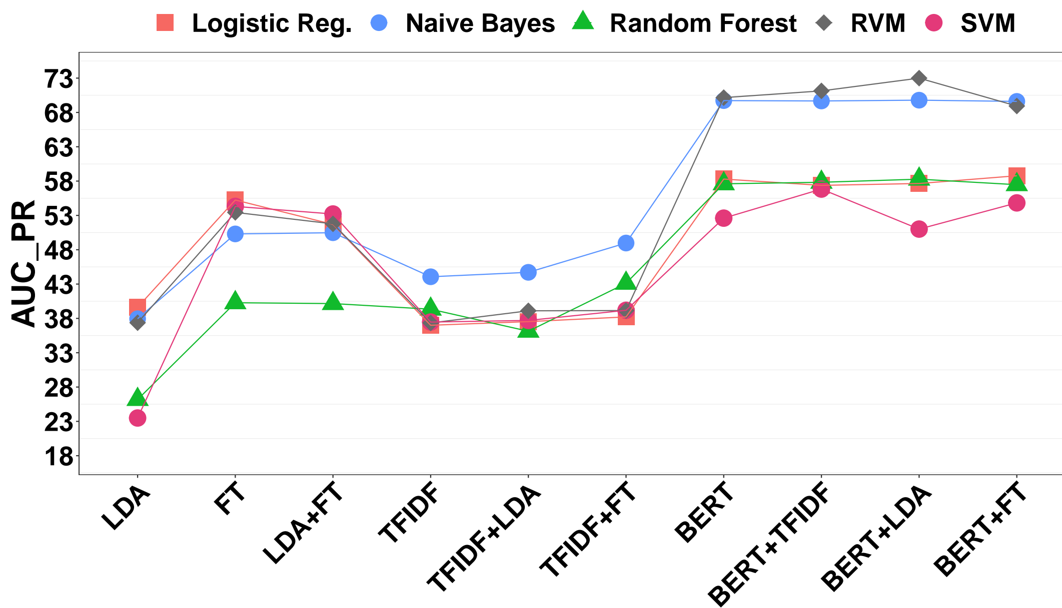
classifier, we will use multiple binary classifiers to classify reviews under one of the three major classes of feature request, bug report, and or user experience. We used binary classifiers to allow for a multi-class and multi-label setup. For evaluation, we will report the averaged AUC generated from the precision and recall across all classes. Additionally, to make sure results are not due to a specific dataset, we will evaluate the performance on both the Panichella and Maalej datasets.

Second, to evaluate the best representation for the summarization task, we will evaluate the different representations on their ability to group reviews with similar requirements together using multiple clustering algorithms. We will evaluate this aspect on the Panichella dataset using the labels we created for each review, which list the discussed requirements as explained in Section 4.3. For the sake of simplicity, we will assign reviews that discuss multiple requirements only one of them, i.e., to have one requirement per review. We will assign the requirement with the highest endorsement, i.e., the requirement that is most discussed. We will compare the performance of each representation using the following clustering algorithms. First, K-means, which is a centroid based clustering algorithm. The only hyper-parameter needed is the number of clusters, which is already known to us. Second, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [19], which is a density based clustering algorithm. DBSCAN requires two hyper-parameters epsilon and minimum sample. We will experiment with different epsilon values ranging from 0.1 to 50 while we keep the value for minimum sample fixed to 1. The results for the best performing model will be reported. Third, Agglomerative Hierarchical Clustering (HAC), which is a hierarchical clustering algorithm. HAC requires two parameters, the number of clusters, and the linkage type. We will keep the number of clusters fixed and experiment with four linkage types: single, average, ward, and complete. The results for the best performing model will be reported. We will evaluate performance using the V-Measure, which is the mean of homogeneity (i.e., each cluster contains only members of a single class) and completeness (i.e., all members of a given class are assigned to the same cluster).

Experiment Results: We can make several observations from the results reported in Figure 4.10. First, the FastText and BERT approaches, which use embeddings, perform better than traditional approaches on the classification task. We can also observe that the embeddings are more resilient to noise than other approaches. They presented the same high performance on both the very noisy dataset of Maalej and the less noisy dataset of Panchilla, whereas, other approaches showed reasonably good performance on Panchilla (the less noisy dataset) but suffered on the the Maalej (the more noisy dataset). Second, for both datasets, we can observe that the best average performance across all classifiers is seen with the embeddings generated from the BERT-MCML model using the 2TL pooling strategy. There is a significant difference between the representation generated from the fine-tuned BERT model and all other approaches. For example,



(a) Panichella dataset



(b) Maalej Dataset

Figure 4.10: Evaluating the best representation for the classification task. We can observe that using an embeddings representation (i.e., FastText or BERT) provides the best average performance across datasets and across classifiers for the multi-class multi-label requirements classification task.

we can observe an average improvement of 26% on Panichella and 48% on Maalej in the performance

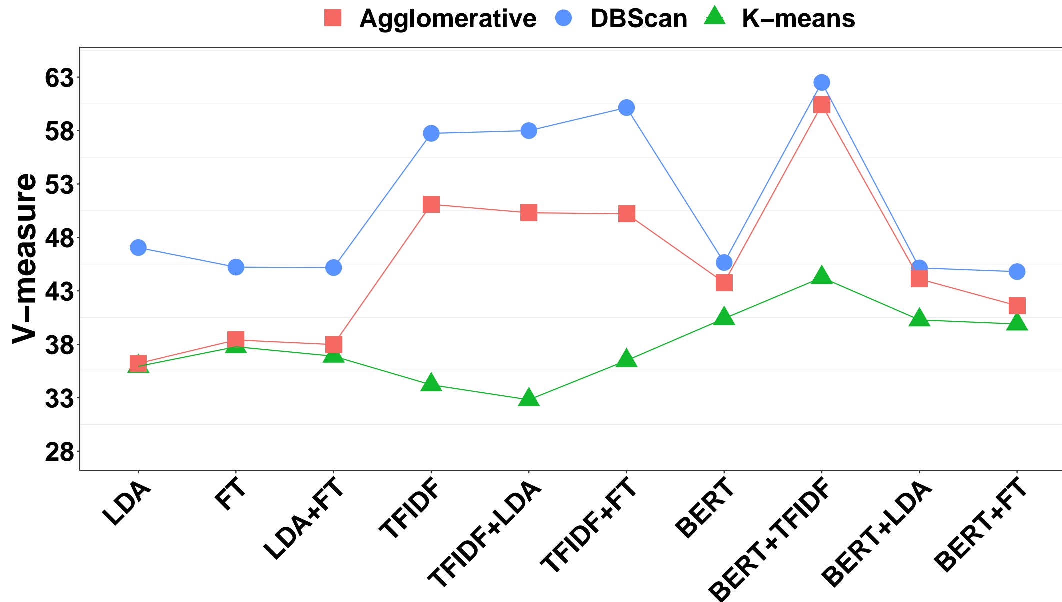


Figure 4.11: Evaluating the best representation on the Panichella dataset. We can observe that the best performing representation across clustering algorithms is the one that combines BERT embeddings with TFIDF.

between BERT's representation and the TFIDF representation, which is the most commonly used representation in current approaches. Third, for both datasets, in terms of average performance across classifiers, the most robust representation across both datasets is the one that merged BERT an TFIDF. We assume this is the case because it combines the best of both worlds. The TFIDF representation is powerful in the sense it allows the models to learn patterns at the term level, i.e., the low level, and it assigns higher weights to the most important terms in each corpus, but it suffers when the corpus is filled with misspelled and alternatively spelled terms. On the other hand, the embeddings generated from BERT can address the limitation of TFIDF as it can better represent the misspelled and alternatively spelled terms, but its representation learns high level patterns and does not assign importance weights at the term level, which TFIDF can address.

Finally, analyzing Figure 4.11, we can observe that K-mean, a centroid-based algorithm, was the only one that benefited from the close placement of reviews with similar requirements under the LDA, FT, and BERT representations. The Agglomerative, a hierarchical-based algorithm, and DB-SCAN, a density-based algorithm, are under performing with those representations while benefiting the most from the TFIDF representation. We assume some models benefit more from high level features (e.g., BERT), while others benefit most from low level features (e.g., TFIDF). Nonetheless,

despite the used learning algorithm, combining the embeddings from the BERT representation with the TFIDF representation is, again, showing that it is the most robust representation for the task.

4.6 The Bayesian Framework

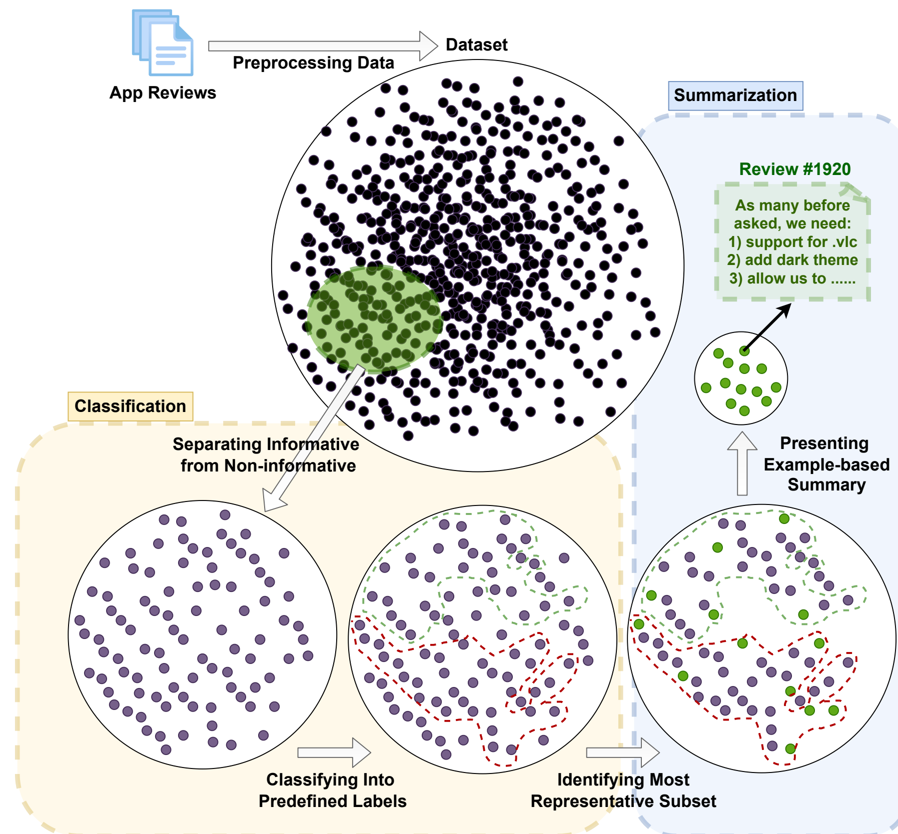


Figure 4.12: The process for extracting requirements from app reviews using the proposed example-based summary approach. The set of reviews provided as the example-based summary contain the set of the most representation reviews in the dataset which is then used for requirement extraction. This approach requires less human effort for requirement extraction because 1) less reviews need to be manually analyzed, and 2) the provided summaries retain the context.

4.6.1 Overview

We propose to use an example-based approach for requirements extraction, which is a better alternative for keyword-based approaches. We aim to create a Bayesian approach that can summarize the mentioned requirements through the identification of the most representative subset of reviews for requirements extraction. This representative subset is a summarized version of the dataset. It should ideally consist of the most descriptive and developer friendly reviews to ease the requirements extraction. Additionally, the representative subset should be much smaller in size to reduce

the human effort needed for requirement extraction. Figure 4.12 provides an overview for the process where we would first filter out the non-informative reviews and classify remaining reviews into a specific set of predefined labels. Next, we would select the most representative reviews and present it to the requirements engineers for requirements extraction.

Thus, our objective is to identify the most representative subset for requirements extraction, which we define as the **smallest** set of informative reviews with the **highest** *coverage* and *expressiveness*. It should allow us to extract the highest number of existing requirements from the most well expressed reviews with least amount of human effort. As a reminder, we repeat the definition for coverage and expressiveness in here. First, the **coverage of requirements** measures the number of requirements captured by the selected set of reviews. The more requirements captured by the selected set, the more representative the set is considered. Second, the **expressiveness of reviews** describes the ease of requirement extraction from each of the reviews in the selected set. A review that contains a well described request or issue is more useful than a review with a vaguely described content. For example, a review that simply states a crash occurred is less useful for requirement extraction than a review that describes what the user was doing when the crash occurred. Thus, the more expressive the selected reviews, the more representative the set is considered.

To achieve this task, we propose to extend Relevance Vector Machines (RVM) [93] with the Criticism selection approach described by Kim et al. [49].

4.6.2 Why Extend RVM?

The way that RVM works is how we propose to accomplish both the classification and summarization tasks using a single model, i.e., this is why RVM specifically is used. As part of RVM's learning process, it identifies the set of most representative points that are referred to as *relevant vectors*. It uses those *relevant vectors* to make predictions. The model attempts to optimize two aspects when it selects the relevant vectors: *Sparsity* and *Quality*. In *quality*, the model measures alignment of the current selected point with the error between training and prediction that would result from excluding it. In *sparsity*, the model measures the extent to which current selected point overlaps with the other selected points. This process will result in a selection of highly *unique* points from *representative* regions. We believe this learning mechanism aligns well with the goal of identifying the set of the most representative reviews for requirements extraction, i.e., summarizing the users feedback. As such, we propose to use those points not only for classification, but also for summarization.

4.6.3 Why Merge RVM with Criticism Selection?

We hypothesize that since RVM focuses on selecting points from representative regions, i.e., points that best represent the distribution of the data, it can miss points that are equally important but less represented. In our specific problem, we suffer from the existence of many such points. We found that a large portion of requirements are represented by a few reviews only, i.e., likely to be missed as it does not belong to a representative region. We also hypothesize that many of the app reviews with level four expressiveness, i.e., most valuable points for requirements extraction, would be missed as most contain mention of multiple requirements, and as such, we assume they will construct their own unique regions that represent the overlap between the different requirements. One way to address this potential issue is the use of Criticism Selection as described by Kim et al. [49]. The authors explain that selecting representative points that summarize the underlying distribution, which they refer to such points as prototypes, is not enough on its own. They explain that the selection process should extend to regions in the input space where the selected prototypical points do not represent well, which they refer to as Criticism. Such points is meant to criticize the selected representative set as they help identify where a particular model may fail to explain the data. Together, prototypes and criticisms construct a better summarization for the underlying data. As a result, we propose to extend RVM to pick criticism points as part of its relevant vector's selection process. We suggest that this may lead to a more comprehensive model that is not only accurate for data points labeling, but also accurate for data points summarization.

4.6.4 The Proposed Approach: Relevance Vector Machine with Criticism Selection (RVMCS)

Let $X = \{\mathbf{x}_1, \mathbf{x}_2, ..\mathbf{x}_N\}$ denote the set of N training instances where $\mathbf{x}_i \in \mathbb{R}^D$.We limit the introduction to Relevance Vector Machines (RVM) [93] to binary classification problem for simplicity where each data instance $\mathbf{x}_i$ is assigned with a label $t_i \in \{0, 1\}$. Later, the binary classification solution can be directly generalized to multi-class problem with one-vs-the-rest prediction approach. The RVM is a probabilistic model in which the label follows the Bernoulli distribution $t_i \sim \text{Bernoulli}(\sigma)$:

$$p(t_i = 1) = y_i = \sigma(\sum_{m=1}^{M} \phi_m(\mathbf{x}_m) w_m) = \sigma(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_m)) \tag{4.1}$$

where $\phi(\mathbf{x}_i)$ is a vector of M basis functions that span the feature space from D to M: $\phi(\mathbf{x}_i) = [\phi_1(\mathbf{x}_i), \phi_2(\mathbf{x}_i), ..\phi_M(\mathbf{x}_i)]$. In RVM, the basis functions are specified with a kernel function $k(\cdot, \cdot)$:

$\phi_i(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}_i)$. The kernel view of (4.1) is given by:

$$p(t_i = 1) = y_i = \sigma(\sum_{n=1}^{N} w_n k(\mathbf{x}, \mathbf{x}_n)) \quad (4.2)$$

where $\mathbf{w}$ are model parameters that follow a Gaussian distribution $p(\mathbf{w}; \boldsymbol{\alpha}) \sim \mathcal{N}(\mathbf{0}, A^{-1})$, with A being a diagonal matrix $A = \text{diag}(\alpha_1, ..., \alpha_N)$. The goal of RVM is to learn the posterior distribution $p(\mathbf{w}|\mathbf{t}, \mathbf{X})$ as well as to give a point estimation on the hyper-parameter $\boldsymbol{\alpha}$.

The posterior distribution can be inferred via Bayesian rule:

$$\ln p(\mathbf{w}|\mathbf{t}, X) \propto \ln p(\mathbf{t}|\mathbf{w}) + \ln p(\mathbf{w}; \boldsymbol{\alpha}) \quad (4.3)$$

Then applying Laplace approximation, the posterior distribution also follows the Gaussian distribution $\mathcal{N}(\mathbf{w}^*, \boldsymbol{\Sigma})$ whose mean and covariance are given by $\mathbf{w}^* = A^{-1}K^T(\mathbf{t} - \mathbf{y})$ and $\Sigma = (K^T BK + A)^{-1}$, respectively. Here, $B = \text{diag}(\mathbf{y}(1 - \mathbf{y}))$.

The hyper-parameter $\boldsymbol{\alpha}$ can be derived using type II maximization. To do that, we first compute the data evidence

$$p(\mathbf{t}|\boldsymbol{\alpha}) \propto \int p(\mathbf{t}|\mathbf{w})p(\mathbf{w}; \boldsymbol{\alpha})d\mathbf{w} \simeq p(\mathbf{t}|\mathbf{w}^*)p(\mathbf{w}^*|\boldsymbol{\alpha}) \quad (4.4)$$

where we used Taylor expansion on the integrant at $\mathbf{w}^*$ to remove the integral. Then the optimal value of $\boldsymbol{\alpha}$ is obtained by solving $\frac{\partial p(\mathbf{t}|\boldsymbol{\alpha})}{\partial \boldsymbol{\alpha}} = 0$:

$$\alpha_i^* = \frac{1 - \alpha_i \Sigma_{ii}}{(w_i^*)^2} \quad (4.5)$$

In the training phase, RVM takes an iterative process of updating $\mathbf{w}^*$, Σ, and $\boldsymbol{\alpha}_i^*$ until (4.4) converges. In the prediction phase, the predictive distribution of a test data point $\mathbf{x}'$is given by $p(t'|\mathbf{x}', \mathbf{w}^*) = \text{Bernoulli}(\sigma(\mathbf{w}^{*T}\mathbf{x}'))$. The prior distribution adopted by RVM is commonly referred as auto relevance detection (ARD) prior. It makes the model prefer simpler explanations than complex explanations so that over-fitting can be automatically addressed. Specifically, during the training process, a certain number of $\boldsymbol{\alpha}$'s components will be driven to infinity, making their corresponding training data instances independent to the prediction and the remaining few determinant training data instances are called relevant vectors.

We make a general extension to RVM through updating its Marginal Likelihood Maximisation algorithm [94] that iteratively updates $\mathbf{w}^*$, Σ, and $\boldsymbol{\alpha}_i^*$ to include criticism points as follows:

1. Initialize RVM with a single basis vector $\boldsymbol{\phi}_i$. This could be the bias, i.e., $\boldsymbol{\phi}_i = (1, 1, ..., 1)^T$.

2. Explicitly compute $\mathbf{\Sigma}$, $\boldsymbol{\mu}$, and initial values of s_m and q_m for all M bases $\boldsymbol{\phi}_m$.
3. Using an alternating mechanism, *select a candidate basis vector* $\boldsymbol{\phi}_i$ from the set of all M as described in the section below.
4. Compute $\theta_i \triangleq q_i^2 - s_i$.
5. If $\theta_i > 0$ and $\alpha_i < \infty$, *re-estimate* α_i.
6. If $\theta_i > 0$ and $\alpha_i = \infty$, *add* $\boldsymbol{\phi}_i$ with updated α_i.
7. If $\theta_i \leq 0$ and $\alpha_i < \infty$, *delete* $\boldsymbol{\phi}_i$ from the model and set $\alpha_i = \infty$.
8. Update $\mathbf{\Sigma}$, $\boldsymbol{\mu}$, and values of s_m and q_m for all M bases $\boldsymbol{\phi}_m$.
9. Terminate if *convergence* is reached, otherwise go to 3.

Selecting a Candidate Basis Vector: The selection process would alternate between selecting a candidate basis vector that gives the greatest increase in marginal likelihood and a candidate that gives the maximum witness function score, i.e., best criticizes the currently selected basis vectors, until convergence or maximum number of iterations is reached. To select the candidate with the greatest increase in marginal likelihood, we need to compute values for α_i and θ_i for all bases, and then the change in marginal likelihood can be computed for each potential update. The candidate giving greatest increase will then be selected. As for selecting a candidate that criticizes the currently selected basis vectors, we would pick the candidate that maximizes our witness function as described in [49]. The witness function is expected to tell us how different two distributions are at a particular data point as follows:

$$witness(x) = \frac{1}{N}\sum_{i=1}^{N} k(x, x_n) - \frac{1}{M}\sum_{j=1}^{M} k(x, z_j) \tag{4.6}$$

where i is a training example from the set of all training examples N and j is a basis vector from the set of all selected basis vectors M. As such, if the witness function for a point x is close to zero, the density function of the data and the prototypes are close together, which means that the distribution of prototypes resembles the distribution of the data at point x. A negative witness function at point x means that the prototype distribution overestimates the data distribution, e.g., if we select a prototype but there are only few data points nearby. a positive witness function at point x means that the prototype distribution underestimates the data distribution, e.g., if there are many data points around x but we have not selected any prototypes nearby.

4.6.5 Evaluation and Experiment

In this section, we will evaluate the proposed framework. We aim to address the following research questions:

- **RQ$_7a$:** How accurate is the proposed approach in terms of the multi-class and multi-label requirements classification?
- **RQ$_7b$:** How accurate is the proposed approach in terms of summarizing the data and identifying the set of the most representative reviews?

To address the research questions, we will first evaluate the proposed approach on both the classification and summarization aspect. Next, we will provide a comprehensive analysis on the inner workings of the proposed approach and discuss its strength and weakness. Finally, we will conclude with a discussion on the framework as a whole compared to other approaches for the task of requirements extraction.

RQ$_7a$: How accurate is the proposed approach in terms of multi-class and multi-label requirements classification?

Experiment Setup: To evaluate our proposed approach, we will compare it against multiple baselines. The hyper-parameters of each baseline were optimized for the purpose of achieving the highest model's performance.

- **AR-Miner** [13], used a Naive Bayes model [70] where the hidden topics of the reviews were discovered using Latent Dirichlet Allocation (LDA) [6] and used alongside the rating of the app review to construct the feature space. To implement their approach, We selected the number of topic k for LDA using cross validation, specifically, we found 85 topics for both the *Maalej* and *Panichella* datasets.
- **Maalej** [58] where also a Naive Bayes model was used, due to its previously reported high performance with text classification. However, [58] used a bag of words approach and extracted the ratio of past, present, and future tenses in the review to represent the textual content, claiming that reviews with bug reports tend to use past tenses, whereas, reviews with feature requests tend to use future tenses. Additionally, they used the review's rating, length, and sentiment score as part of their feature space.

- **ARdoc** [75, 76], where a Decision Tree (J48) was used. The authors manually constructed a set of 246 linguistic patterns each mapping to a specific app review label/category, e.g., reviews with pattern *[someone] should add [something]* are mapped to *feature requests*. Moreover, they generate a Term Frequency - Inverse Document Frequency (TF-IDF) [54] from the textual content of the reviews, and used the review's sentiment score in their feature space. Due to the difficulties in recreating the 246 linguistic patterns, we did not implement this approach ourselves, but rather used the tool provided by the authors to generated labels. As such, we do not have the AUC score for this baseline, since computing them requires access to the model itself to evaluate performance under different decision thresholds.

- **Naive Bayes (NB Classifier)**: We used a Naive Bayes model (Multinomial) with our proposed representation that combines the BERT representation with the TFIDF representation as discussed in Section 4.5.5. This baselines is used to directly compare an alternative Bayesian approach that uses the same representation as RVM.

- **Relevant Vector Machines (RVM)**: We used the Relevant Vector Machines model (Linear Kernel) [94] with our proposed BERT and TFIDF representation described in Section 4.5.5.

- **Relevant Vector Machines with a Hierarchical Classification approach(RVM-HC)**: We used the same exact setup as the RVM baseline, but we used a hierarchical approach to conduct the classification.

- **Relevant Vector Machines with Criticism Selection (RVMCS)** : This is our proposed approach that merges RVM [94] with Criticism Selection [49]. We used a Linear Kernel as well and utilized the same proposed BERT and TFIDF representation described in Section 4.5.5.

Experiment Results: We can make several observations from the results in Tables 4.9 and 4.8.

First, we notice a better overall performance for all the baselines on the Panichella dataset than the Maalej dataset. We attribute this difference to the fact that the Maalej dataset contains a lot more short and noisy app reviews than the Panichella dataset. When such reviews constitute a large portion of a dataset, it becomes hard for any model to find patterns in the data, hence the lower performance.

Second, we can observe that the RVMCS, i.e., proposed approach, constantly outperformed regular RVM by an average of 9.4% in terms of AUC_{pr}. We attribute this difference to the notion that RVMCS can select more representative points than regular RVM, which led to a better classification

Table 4.8: Summary of the classification results comparing proposed approach to the start of the art on the Panichella Dataset.

Approach	**Panichella Dataset**				
	AUC$_{PR}$	**m**F_1	**M**F_1	**M**P	**M**R
AR-miner [13]	0.432	0.472	0.444	0.345	0.699
Maalej [58]	0.668	0.677	0.640	0.645	0.647
ARdoc [75, 76]	-	0.376	0.307	0.642	0.344
NB Classifier	0.702	0.666	0.634	0.681	0.603
RVM	0.676	0.664	0.619	0.690	0.563
RVM-HC	0.725	0.713	0.685	0.648	**0.738**
Proposed Approach	**0.759**	**0.758**	**0.728**	**0.735**	0.722

Table 4.9: Summary of the classification results comparing proposed approach to the start of the art on the Maalej Dataset.

Approach	**Maalej Dataset**				
	AUC$_{PR}$	**m**F_1	**M**F_1	**M**P	**M**R
AR-miner [13]	0.402	0.496	0.445	0.363	**0.634**
Maalej [58]	**0.463**	**0.559**	**0.510**	**0.463**	0.587
ARdoc [75, 76]	-	0.338	0.267	0.341	0.325
NB Classifier	0.447	0.526	0.473	0.439	0.524
RVM	0.405	0.418	0.380	0.417	0.351
RVM-HC	0.425	0.521	0.447	0.401	0.541
Proposed Approach	0.462	0.540	0.479	0.448	0.543

accuracy. We will investigate this assumption further when evaluating the performance on the summarization aspect.

Third, we can observe that RVMCS provided equal or better results than the state of the art on both datasets. On the Panichella dataset, the proposed approach, RVMCS, outperformed all the baselines by a good margin. While on the Maalej dataset, we can observe that the Maalej approach performed the best, the proposed approach, RVMCS, came in second with a small marginal difference in performance.

RQ$_7b$: How accurate is the proposed approach in terms of summarizing the data and identifying the set of the most representative reviews?

Experiment Setup: To evaluate the proposed approach (RVMCS) ability to summarize the data and identify the set of the most representative reviews, we will compare it against multiple baselines. Each baseline will select a subset of reviews as the set of the most representative reviews using a different approach as follows:

- **AR-Miner** [13], used a Naive Bayes model [70] where the hidden topics of the reviews were discovered using Latent Dirichlet Allocation (LDA) [6] and used alongside the rating of the app review to construct the feature space. To evaluate the summarization aspect, LDA will be used to group the set of reviews predicted as *informative*, and then the review with the highest probability for each topic will be picked as the most informative review. The size of the final list of selected reviews will be equal to the number of topics.
- **Maalej** [58] where also a Naive Bayes model was used, due to its previously reported high performance with text classification. However, [58] used a bag of words approach and extracted the ratio of past, present, and future tenses in the review to represent the textual content, claiming that reviews with bug reports tend to use past tenses, whereas, reviews with feature requests tend to use future tenses. Additionally, they used the review's rating, length, and sentiment score as part of their feature space. The original authors did not propose any summarization approach, so we will apply K-means to the set of reviews classified as *informative* to cluster them, and then use the core samples as the most representative review for each cluster.
- **Latent Dirichlet allocation (LDA)**: This is a probabilistic baseline with a well established ability to summarize textual data. We applied LDA to our proposed representation that combines the BERT representation with the TFIDF representation, which we discussed in Section 4.5.5, to summarize the data to k topics. For each topic, the review with the highest probability for that topic will be picked as the most representative review. We will then evaluate the set of most representative reviews under each topic, hence, the size of the final list of selected reviews will be equal to the number of topics.
- **DBscan**: This is a well established density-based clustering approach that was used in many similar problems and showed promising results for summarizing data. We will apply DBSCAN to the set of reviews classified as *informative* to cluster them, and then use the core samples as the most representative review for each cluster.

- **Star Clustering**: This is a graph-based clustering approach [?] that can be used to summarize data with network like connections. The approach creates a graph where each node is a review, and then creates edges whenever the cosine similarity between two reviews is larger than a given alpha. We will apply Star Clustering to our proposed representation that combines the BERT representation with the TFIDF representation, which we discussed in Section 4.5.5. Once the graph is constructed, we will use the set of nodes with the highest degree to be the set of *center stars*, i.e., most representative reviews for requirements extraction.

- **Kim Et al. Approach (Kim Et al.)**: This baseline uses the original work by Kim et al. [49] that introduced the idea of using prototypes and criticism points to summarize the data. We will apply it to the our proposed representation that combines the BERT representation with the TFIDF representation, which we discussed in Section 4.5.5 to sample prototypes and criticism points. Following the steps of the original paper [49], we will use the approach to identify a sample where the majority of points are prototypes and the remaining points as criticisms. The sample picked by the model will be set of the most representative points.

- **Prototype and Criticism Selection (PCS)**: This baseline represents a follow up by IBM research [28] on the work of Kim et al. [49]. Unlike the original work which separates the selection of prototypes from the selection of criticisms, the researchers in [28] combine the selection of both under a single a framework. Given a dataset and a number of points k to be selected, the model will determine the optimal ratio of prototypes and criticism points and return the set of the most representative points where both are represented.

- **Relevant Vector Machines (RVM)**: This baseline will use the Relevant Vector Machines model (Linear Kernel) [94] with our proposed BERT and TFIDF representation described in Section 4.5.5. The selected relevant vectors by the model will be used as the set of the most representative points.

- **Relevant Vector Machines with a Hierarchical Classification approach(RVM-HC)**: We used the same exact setup as the RVM baseline, but we used a hierarchical approach to conduct the classification. The selected relevant vectors by the model will be used as the set of the most representative points.

- **Relevant Vector Machines with Criticism Selection (RVMCS)** : This is our proposed approach that merges RVM [94] with Criticism Selection [49]. We used a Linear Kernel as well and utilized the same proposed BERT and TFIDF representation described in Section 4.5.5. The selected relevant vectors by the model will be used as the set of the most representative points.

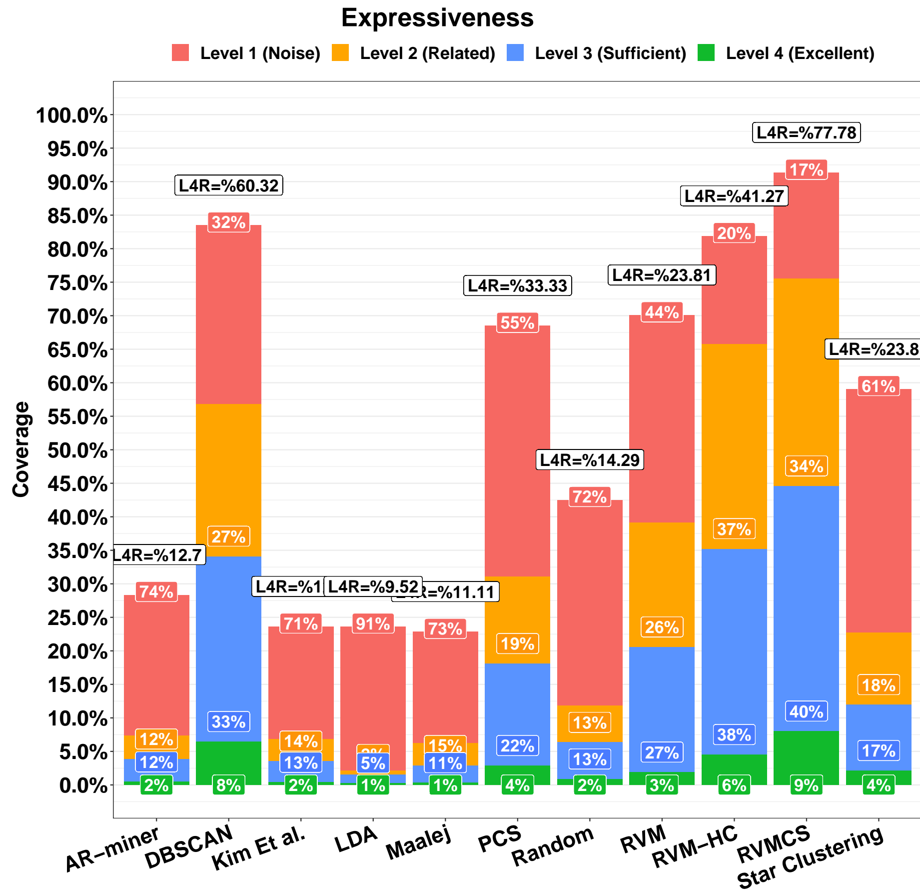


Figure 4.13: The summarization results on the Panichella dataset. First, on the y-axis, we show the coverage, which is the percentage of requirements that were captured by the approach versus the ground truth. Second, inside the whitebox at the top of the barchart, we show the recall percentage of points with level four expressiveness (L4R). Lastly, the colors of the barchart shows the level of expressiveness in the selected sample, i.e., noise vs signal ratio.

All the baselines were optimized for the purpose of achieving the highest accuracy while maintaining a sample size that is within 17%-19% of the original dataset. This fixed size range is used to make sure a fair comparison is maintained between baselines, where roughly the same number of points is selected for the set of the most representative reviews across all baselines. Also, keeping a small sample size compared to the original dataset assures maximum minimization of the human efforts needed to extract the requirements, which is a goal we aim to achieve.

As for the evaluation of the summarization capability, we will evaluate each baseline on three aspects. First, the **coverage**, i.e., the ability to capture as many of the discussed requirements

as possible. The higher the coverage, the better the model. Second, the **sample's signal/noise ratio**, i.e,. the ability to capture as much of the signal (reviews with expressiveness of level three and four) and filter as much of the noise (reviews with expressiveness of level one and two) within the selected sample. As a reminder, a review with a level one expressiveness is a review that does not discuss or mention any requirements in its content, i.e., considered pure noise, whereas, a level four app review is a review that discusses multiple requirements and uses a developer friendly language that can ease the extraction of the requirements. We want the highest signal ratio and the lowest noise ratio. Third, as we find points with level four expressiveness especially valuable, we will use the **Level Four Recall (L4R)** as an evaluation metric as well. This means we will evaluate each baseline on how many level four reviews it captured with its selected sample compared to the existing level four reviews within the dataset. The higher the recall, the better the model.

Finally, since requirements with a single review do not have enough statistical presence, we will only focus on capturing requirements with at least two reviews. We believe this will not be an issue with a larger dataset as each requirement should have at least two reviews. As such, we argue that the exclusion of requirements with a single review will not have any effect on the results of the experiment, nor its real-world applicability.

Experiment Results: We can observe from the results shown in Figure 4.13 that the proposed approach, RVMCS, outperforms all baselines in all three evaluation aspects.

Evaluating coverage, we can see that RVMCS outperformed all the baselines by achieving a 91.3% coverage, i.e., the representative sample picked by the model for requirements extraction was able to capture over 90% of the requirements discussed in the dataset with only a fraction of the original size (19%). This performance filters out 80% of the human effort needed for requirements extraction as a requirement engineer would only need to manually analyze this set to capture 91.3% of the discussed requirement, which is a very reasonable trade between effort and completeness.

Looking at the sample's signal/noise ratio, we can observe that RVMCS selected only 17% app reviews with level one expressiveness, which is the lowest level of noise selected across all the baselines. Additionally, if we merge the ratio for app reviews with level one and level two expressiveness, we find that RVMCS also provided the lowest total noise within its sample with only 51% (17% + 34%). Moreover, the proposed approach selected the highest ratio of app reviews with expressiveness level of three and four with a total of 49% (40% + 9%). Thus, RVMCS was able to select a representative sample that achieve the highest signal ratio and the lowest noise ratio.

Evaluating the level four expressiveness recall, we can also observe that RVMCS was able to out-

perform all the baselines by achieving a recall of 77.78%. This means the proposed approach was able to include 78% of the most useful app reviews for requirements extraction as part of the final selected representative sample.

When comparing RVMCS to its the original RVM and its hierarchical version HRVM, we can observe that it outperformed both on all aspects. More importantly, RVMCS showed a significant improvement in the selection of app reviews with level four expressiveness. E.g., compared to regular RVM, it selected more than two times the number of level four points, showing an improvement of 226.6%.

Finally, when analyzing other baselines, we can see that most suffer from selecting more noise than signal which heavily impacted their coverage score. E.g., 71% of LDA's set is pure noise, i.e., app reviews that do not discuss or mention any requirements. We hypothesize this might be due the way those approaches select their representative set. In the case of LDA, it is the points with the highest topic probability. In the case of Kim Et al., it is prototypes that are selected from concentrated regions. In the case of K-means, it is the points in the center of the cluster. For all those approaches, when the noise constitutes the majority of the dataset, they will be drawn to picking points from the noise more than the signal, which we believe to be the reason behind the poor performance. Perhaps the only promising baseline is DBSCAN. We attribute its performance to the feature space we constructed and discussed in Section 4.5.5. Under such a space a density-based approach is expected to shine for two reasons: having a feature space where app reviews with different requirements are well separated into different regions, and knowing that most of those regions contain a small number of app reviews. This allows a density-based approach to have clusters where each is a requirement, i.e, picking a point per cluster will guarantee a high coverage. Also, having a small number of app reviews under each cluster means that the approach will have a high chance of picking good points than approaches that pick from a larger pool where majority is noise.

4.6.6 Results Discussion

How much of RVMCS performance is due to the addition of criticism selection?

The two approaches complement each other nicely for our specific problem, which explain the added performance. To understand this better, we show, in Figure 4.14, a run of regular RVM and the selected relevant vectors that it picked for the Panichella dataset. In this Figure, each

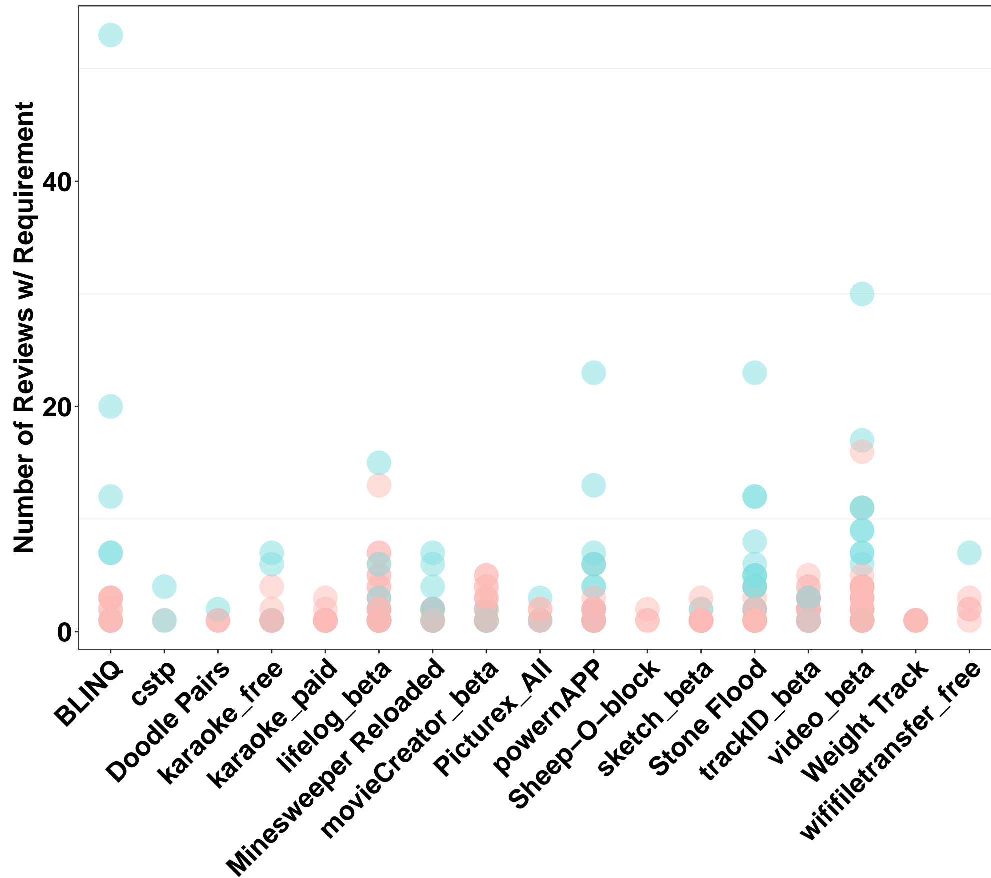


Figure 4.14: The Figure shows a run of regular RVM and the selected relevant vectors that it picked for the Panichella dataset. In this Figure, each dot is a requirement, and the y-axis shows the number of reviews under that requirement. The higher on the y-axis the requirement, the more reviews are talking about it, and the more dense its region would be in the space. The blue colored points are the captured requirements using the selected RVM relevant vectors. The red dot colored points are requirements that were missed by RVM relevant vectors.

dot is a requirement, and the y-axis shows the number of reviews under that requirement. The higher on the y-axis the requirement, the more reviews are talking about it, and the more dense its region would be in the space. The blue colored points are the requirements captured by the RVM's selected relevant vectors. We can observe that RVM is mostly picking points from from highly representative regions. While this behaviour is expected and is great for our task, it limits RVM's ability to capture all the points we aim to capture as it will miss those points that belong to a less represented regions. For example, RVM is expected to miss requirements with a few reviews. We can see this clearly in Figure 4.14 as most of the missed requirements are the bottom, where

Table 4.10: Below are three different runs with different splits of RVMCS. We summarize the behaviour of RVMCS with each run and how the reviews with level four expressiveness were selected. We first show (RVM Only), which represents the number of points selected by RVM's maximum marginal likelihood approach. Next, (Criticism Only), which represents points selected by Kim et al. Criticism selection approach. Finally, we show (Overlap), which represents the number of points selected by both approaches.

Run	**RVM Only**	**Criticism Only**	**Overlap**
1	3 (6%)	24 (49%)	22 (45%)
2	6 (12%)	29 (58%)	15 (30%)
3	8 (14%)	31 (54%)	18 (32%)

requirements with less dense regions are placed on the Figure. It is clear that RVM is picking more points from the top compared to the bottom. Additionally, we argue that many of the reviews with level four expressiveness might as well be missed by regular RVM as we assume they will construct their own unique regions that represent the overlap between the different requirements. This is where criticism selection role comes in. It forces RVM to include points from those less represented regions. As such, the overall framework merging the two would have a more regional comprehensive selection, which is what we believe is behind the performance we observed from merging the two approaches. We can observe that this is true from comparing the performance of regular RVM against the RVMCS.

By analyzing Table 4.8 and Figure 4.13, we can observe the difference in performance between regular RVM and the proposed RVMCS. We can observe that when we added criticism selection to the inner workings of RVM, we got two main improvements: We got an average of 9.4% improved classification accuracy in terms of AUC_{pr}, and we observed a substantial increase that is up to 226.6% in the app reviews with level four expressiveness. Those two improvements show the added value of the merge between RVM and criticism selection for our specific problem. We can observe that regular RVM was able to capture only 23.81% of the existing reviews with level four expressiveness, whereas, RVMCS was able to capture 77.78% of those points, which is a significant boost that was only possible after the merge between the two approaches. Moreover, RVM selected sample had 30% level three and four points, whereas, RVMCS had 49% level three and four points as part of its selected sample. This is also evidence that RVMCS is able to identify representative points with a higher accuracy than regular RVM.

Additionally, by analyzing RVMCS's behaviour, we made multiple observations that show the addition of criticism selection played a significant role in the selection of the most representative data points. In Table 4.10, we show three runs of RVMCS and the analysis of the selected app reviews with level four expressiveness, i.e., the most representative data points. In each run we evaluated the contribution of each approach in selecting the level four points. We can observe that for all the three runs, criticism selection proposed over 50% of selected level four points on average and that those reviews were uniquely proposed by criticism selection. This shows the added value of the merge between the two and we argue that this is another evidence that explain the improvement observed in the performance over regular RVM.

Table 4.11: Example of real-world reviews from the Panichella dataset and how RVMCS was able to capture their mentioned requirements using the least number of app reviews.

Review	**Req. Id(s)**	**Is RV?**
Reviews for Lifelog (Health and Fitness Application)		
Any chance of an export option so I can open the data from lifelog in Excel and analyse it? ...	1086	No
Still experiencing .. fonts problem. Cant see text.	1068	No
I would like ... to refresh and load my activity ... without having to connect to the network...	1079	No
I like to suggest the following: 1. Allow users to download their tracking data... 2.have the app...work offline w/o needing to sync all time...3. More options to customize the font, color, appearance.. the latest copy of lifelog on my Note 4 Samsung has a transparent font in sub menus .. I cant see anything!)	1068, 1086, 1079	**Yes**

What are some examples of the type of summarization this framework offers?

One of the main objectives of this framework is minimize human effort needed to extract requirements. To accomplish this task, we replace the need to manually analyze the complete set of informative reviews, as it is the case with previous approaches, with only using a subset that is substantially smaller in size. This subset is what we refer to as the set of the most representative

reviews. We will use this set for requirements extraction.

To better explain this idea, we show an example of the type of summarization achieved in Table 4.11. We can see four reviews in the table. The first three reviews discuss three different requirements. However, the fourth review summarizes the previous three reviews as it lists all the discussed requirements. Using previous approaches, the requirement engineer would need to read all four reviews. However, in our case, RVMCS was able to select only the fourth review and presented as part of the set of the most representative reviews. As such, the requirement engineer would only need to manually analyze the fourth review and would still be able to capture all the three discussed requirements.

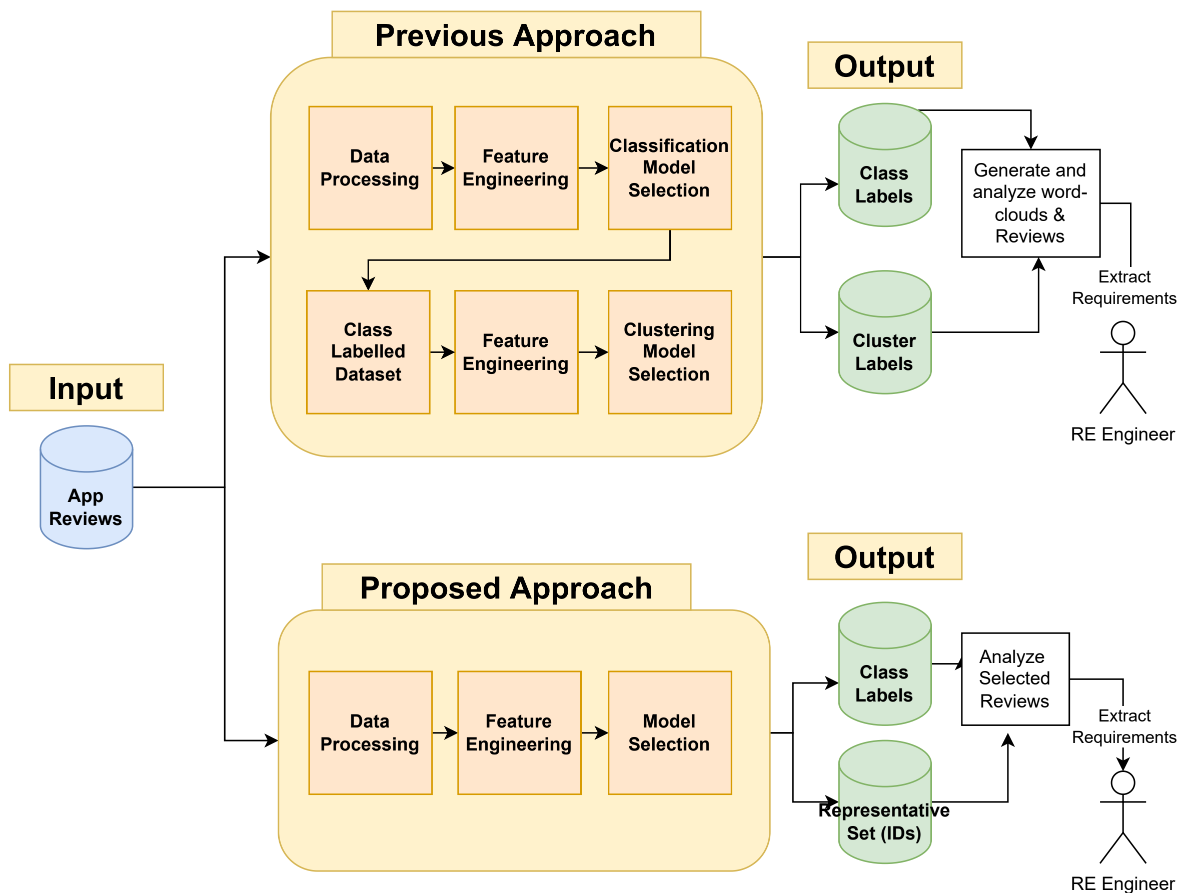


Figure 4.15: Comparison between the proposed framework for requirements extraction and existing approaches. The proposed approach is simpler to create and maintain. It also provides a more efficient method for requirements extraction, i.e., example-based summaries.

What is the added value of RVMCS over existing approaches?

In Figure 4.15, we show the difference in the requirements extraction pipeline between previous approaches and our proposed approach. In previous approaches, to extract the requirements, we would need to first classify the reviews into the predefined labels using a classification model, and then use a clustering model to group reviews with similar requirements together, and then we would need to summarize the data using a visualization technique such as word-clouds. Finally, using the word-clouds summary we would try to make sense of the existing requirements, and then go over the reviews to extract it. This two-step sequential pipeline is complex and difficult to create and maintain as it requires the training and tuning of two models where the quality of the second model relies on the quality of the first model. Also, using word-clouds to summarize the data, i.e., keyword summarizes, was reported to be inefficient for requirements extraction as it lacks context.

Alternatively, our approach uses example-based summaries where the data is summarized using a representative subset that is used for requirements extraction. This is the first added value for RVMCS as this approach retains context and can significantly reduce human effort needed to extract the requirements due to the fact that the RE engineer would only need to review and analyze the representative subset to extract the requirements. In addition to providing a better method for requirements extraction, a second added value is its performance where we demonstrated that RVMCS can provide equal or better results than the state of the art in terms of classification accuracy and that it can summarize the data better than all the alternative approaches as it was able to capture 91.3% of the discussed requirement with only 19% of the reviews. Moreover, a third added value is the fact that this performance was reached using a single model that requires little to none fine-tuning. The proposed RVMCS can do both the classification and summarization in one step, which significantly reduces the effort and domain knowledge needed to create and maintain the pipeline.

Chapter 5

Future Work

First, for modeling implicit feedback using popularity prediction, the functionality is learned only from the description which mostly provide a high level description of the app, but does not contain details on the features provided. One potential extension for this work is to include additional information in the functionality learning process, e.g, the description of the used APIs and libraries (e.g., from stackshare.io), or discussions on public forums such as reddit.com where many apps maintain subreddit forums that discuss aspects of the functionality, e.g., answers to the question what is dropbox and how to use it. Second, the work provides a popularity estimation for a given functionality along with an explanation for the prediction. However, it does not recommend or suggest *how to improve the popularity.* An extension to current work would be to investigate how to provide such insight. One potential answer would be to create a recommender system that suggests the use of a specific API or library based on the change in the estimated popularity when such API/library is used. This change in popularity between the two APIs/libraries can be explained by the higher reliability of service or the potential to provide more functionality in one over the other. Third, we are using a single type of feedback which may limit our evaluation ability. For example, the use count can capture exposure but it does not capture sentiment. One possible extension for this work is to model users feedback on public forums to understand how they feel about the given functionality. This would help developers to prioritize requirements, i.e., pick the one with the most appeal to be implemented first.

Second, for explicit feedback modeling, even though the goal of data-driven requirements engineering is to combine data from different sources, we only consider app reviews in this work. The addition of more sources can help in capturing new requirements that are not present in app re-

views. Nayebi et al. [68, 69] found that they were able to mine 22.4% additional features and 12.89% additional bug reports from Twitter, concluding that app review mining is not enough and that other information sources must be considered as they provide added value to requirements. Additionally, we argue that it can help with context understanding of ambiguous requirements, i.e., due to ambiguity around the app review itself. According to Maalej et al. [57], one of the main challenges with requirements elicitation from app reviews is attempting to understand the context around a reported problem. We propose to study whether we can better understand the context of such ambiguous requests/problems by considering other sources of information such as Twitter or reddit.com. We may find a higher quality user feedback on the same topic that better describes the same request/issue. Additionally, the current approach is limited to a single RE activity, i.e., requirements elicitation. One potential extension is to consider other activites, e.g., to provide techniques that can help decision makers in requirements prioritization, i.e., identify which requirements should be addressed in the next release through metrics such as popularity, affected number of users, estimated effort, etc. Additionally, we can build up on more recent work by Winkler et al. [106] that attempts to predict what type of validation a given requirements would need, e.g., by manual review, testing, or simulation; also called potential verification method. Moreover, current work does not consider the potential of using the reviews of competitive apps to suggest new requirements, e.g., feature requests, or to provide an alert that a competitor is facing issues. For example, Telegram, which is a mobile text messaging app, gained three million users in 24 hours following the outage in competitor WhatsApp [1], which highlights the importance of monitoring competition.

[1] https://thehackernews.com/2019/03/encrypted-telegram-messenger.html

Bibliography

[1] Moayad Alshangiti, Weishi Shi, Xumin Liu, and Qi Yu. A bayesian learning model for design-phase service mashup popularity prediction. *Expert Systems with Applications*, 149:113231, 2020.

[2] Aybüke Aurum and Claes Wohlin. The fundamental nature of requirements engineering activities as a decision-making process. *Inf. Softw. Technol.*, 45(14):945–954, 2003.

[3] Roja Bandari, Sitaram Asur, and Bernardo A. Huberman. The pulse of news in social media: Forecasting popularity. In *AAAI International Conference on Weblogs and Social Media ICWSM*, 2012.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[5] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[7] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *TACL*, 5:135–146, 2017.

[8] Hudson Borges, André C. Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *IEEE International Conference on Software Maintenance and Evolution*, pages 334–344, 2016.

[9] Gargi Bougie, Jamie Starke, Margaret-Anne Storey, and Daniel M. German. Towards understanding twitter use in software engineering: Preliminary findings, ongoing challenges and future questions. In *Proceedings of the 2nd International Workshop on Web 2.0 for Software*

Engineering, Web2SE '11, page 31–36, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Washington, DC, USA, 3rd edition, 2014.

[11] Laura V. Galvis Carreño and Kristina Winbladh. Analysis of user comments: an approach for software requirements evolution. In *Proceedings of the 35th International Conference on Software Engineering, ICSE*, pages 582–591. IEEE Computer Society, 2013.

[12] Eya Ben Charrada. Which one to read? factors influencing the usefulness of online reviews for RE. In *Proceedings of the 24th IEEE International Requirements Engineering Conference, RE*, pages 46–52. IEEE Computer Society, 2016.

[13] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. Ar-miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering, ICSE*, pages 767–778. ACM, 2014.

[14] Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald C. Gall. Analyzing reviews and code of mobile apps for better release planning. In Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors, *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 91–102. IEEE Computer Society, 2017.

[15] Alan M. Davis. *Software requirements - objects, functions, and states*. Prentice Hall international editions. Prentice Hall, 1993.

[16] Alan M. Davis. The art of requirements triage. *IEEE Computer*, 36(3):42–49, 2003.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[18] Jeremy Dick, Elizabeth Hull, and Ken Jackson. *Introduction*, pages 1–32. Springer International Publishing, Cham, 2017.

[19] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231. AAAI Press, 1996.

[20] Flavio Figueiredo. On the prediction of popularity of trends and hits for user generated videos. In *ACM International Conference on Web Search and Data Mining*, pages 745–754, 2013.

[21] Bin Fu, Jialiu Lin, Lei Li, Christos Faloutsos, Jason I. Hong, and Norman M. Sadeh. Why people hate your app: making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, pages 1276–1284. ACM, 2013.

[22] Cuiyun Gao, Jichuan Zeng, David Lo, Chin-Yew Lin, Michael R. Lyu, and Irwin King. INFAR: insight extraction from app reviews. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 904–907. ACM, 2018.

[23] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*, pages 21–26. IEEE Computer Society, 2007.

[24] Giovanni Grano, Andrea Di Sorbo, Francesco Mercaldo, Corrado Aaron Visaggio, Gerardo Canfora, and Sebastiano Panichella. Android apps and user feedback: a dataset for software evolution and quality improvement. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics, WAMA@ESEC/SIGSOFT FSE 2017, Paderborn, Germany, September 5, 2017*, pages 8–11. ACM, 2017.

[25] The Standish Group. The standish group report, 1995.

[26] The Standish Group. The standish group report, 2014.

[27] Xiaodong Gu and Sunghun Kim. "what parts of your apps are loved by users?" (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 760–770. IEEE Computer Society, 2015.

[28] Karthik S. Gurumoorthy, Amit Dhurandhar, Guillermo A. Cecchi, and Charu C. Aggarwal. Efficient data representation by selecting prototypes with importance weights. In *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*, pages 260–269. IEEE, 2019.

[29] Emitza Guzman, Rana Alkadhi, and Norbert Seyff. A needle in a haystack: What do twitter users say about software? In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*, pages 96–105. IEEE Computer Society, 2016.

[30] Emitza Guzman, Omar Aly, and Bernd Bruegge. Retrieving diverse opinions from app reviews. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*, pages 21–30. IEEE Computer Society, 2015.

[31] Emitza Guzman, Muhammad El-Haliby, and Bernd Bruegge. Ensemble methods for app review classification: An approach for software evolution (N). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 771–776. IEEE Computer Society, 2015.

[32] Emitza Guzman, Mohamed Ibrahim, and Martin Glinz. A little bird told me: Mining tweets for requirements and software evolution. In Ana Moreira, João Araújo, Jane Hayes, and Barbara Paech, editors, *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*, pages 11–20. IEEE Computer Society, 2017.

[33] Emitza Guzman and Walid Maalej. How do users like this feature? A fine grained sentiment analysis of app reviews. In Tony Gorschek and Robyn R. Lutz, editors, *Proceedings of the IEEE 22nd International Requirements Engineering Conference, RE*, pages 153–162. IEEE Computer Society, 2014.

[34] Elizabeth Ha and David A. Wagner. Do android users write about electric sheep? examining consumer reviews in google play. In *Proceedings of the 10th IEEE Consumer Communications and Networking Conference, CCNC*, pages 149–157. IEEE, 2013.

[35] Mark Harman, Yue Jia, and Yuanyuan Zhang. App store mining and analysis: MSR for app stores. In *Proceedings of the 9th IEEE Working Conference of Mining Software Repositories, MSR*, pages 108–111. IEEE Computer Society, 2012.

[36] Xiangnan He, Ming Gao, Min-Yen Kan, Yiqun Liu, and Kazunari Sugiyama. Predicting the popularity of web 2.0 items based on user comments. In *ACM International Conference on Research and Development in Information Retrieval*, pages 233–242, 2014.

[37] Liangjie Hong, Ovidiu Dan, and Brian D. Davison. Predicting popular messages in twitter. In *Proceedings of the ACM 20th International Conference on World Wide Web, WWW*, pages 57–58, 2011.

[38] André C. Hora and Marco Tulio Valente. Apiwave: Keeping track of API popularity and migration. In *IEEE International Conference on Software Maintenance and Evolution*, pages 321–323, 2015.

[39] Daqing Hou and David M. Pletcher. Towards a better code completion system by API grouping, filtering, and popularity-based ranking. In *ACM International Workshop on Recommendation Systems for Software Engineering*, pages 26–30, 2010.

[40] Minqing Hu and Bing Liu. Mining and summarizing customer reviews. In *Proceedings of the Tenth ACM International Conference on Knowledge Discovery and Data Mining, SIGKDD*, pages 168–177. ACM, 2004.

[41] Claudia Iacob and Rachel Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR*, pages 41–44. IEEE Computer Society, 2013.

[42] Tahira Iqbal, Parisa Elahidoost, and Levi Lucio. A bird's eye view on requirements engineering and machine learning. In *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018*, pages 11–20. IEEE, 2018.

[43] Aditi Jain, Xumin Liu, and Qi Yu. Aggregating functionality, use history, and popularity of apis to recommend mashup creation. In *International Conference on Service-Oriented Computing*, volume 9435, pages 188–202. Springer, 2015.

[44] Wei Jiang, Haibin Ruan, Li Zhang, Philip Lew, and Jing Jiang. For user-driven software evolution: Requirements elicitation derived from mining online reviews. In Vincent S. Tseng, Tu Bao Ho, Zhi-Hua Zhou, Arbee L. P. Chen, and Hung-Yu Kao, editors, *Advances in Knowledge Discovery and Data Mining - 18th Pacific-Asia Conference, PAKDD 2014, Tainan, Taiwan, May 13-16, 2014. Proceedings, Part II*, volume 8444 of *Lecture Notes in Computer Science*, pages 584–595. Springer, 2014.

[45] Timo Johann, Christoph Stanik, Alireza M. Alizadeh B., and Walid Maalej. SAFE: A simple approach for feature extraction from app descriptions and app reviews. In *25th IEEE International Requirements Engineering Conference, RE*, pages 21–30. IEEE Computer Society, 2017.

[46] Swetha Keertipati, Bastin Tony Roy Savarimuthu, and Sherlock A. Licorish. Approaches for prioritizing feature improvements extracted from app reviews. In Sarah Beecham, Barbara A.

Kitchenham, and Stephen G. MacDonell, editors, *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE 2016, Limerick, Ireland, June 01 - 03, 2016*, pages 33:1–33:6. ACM, 2016.

[47] Yaser Keneshloo, Shuguang Wang, Eui-Hong Sam Han, and Naren Ramakrishnan. Predicting the popularity of news articles. In *Proceedings of the 2016 SIAM International Conference on Data Mining*, pages 441–449, 2016.

[48] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[49] Been Kim, Oluwasanmi Koyejo, and Rajiv Khanna. Examples are not enough, learn to criticize! criticism for interpretability. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2280–2288, 2016.

[50] Su-Do Kim, Seon-Yeong Kim, and Hwan-Gue Cho. A model for popularity dynamics to predict hot articles in discussion blog. In *International Conference on Ubiquitous Information Management and Communication*, pages 10:1–10:8, 2012.

[51] Daphne Koller and Mehran Sahami. Hierarchically classifying documents using very few words. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML*, pages 170–178. Morgan Kaufmann, 1997.

[52] Jong Gun Lee, Sue B. Moon, and Kavé Salamatian. An approach to model and predict the popularity of online contents with explanatory factors. In *IEEE/WIC/ACM International Conference on Web Intelligence*, pages 623–630, 2010.

[53] Kristina Lerman and Tad Hogg. Using a model of social dynamics to predict popularity of news. In *ACM International World Wide Web Conference*, pages 621–630, 2010.

[54] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets, 2nd Ed*. Cambridge University Press, 2014.

[55] Wenjie Liang, Wenyi Qian, Yijian Wu, Xin Peng, and Wenyun Zhao. Mining context-aware user requirements from crowd contributed mobile data. In Hong Mei, Jian Lü, Xiaoxing Ma, Qianxiang Wang, Gang Yin, and Xiaofei Liao, editors, *Proceedings of the 7th Asia-Pacific Symposium on Internetware, Internetware 2015, Wuhan, China, November 6, 2015*, pages 132–140. ACM, 2015.

[56] Lin Liu, Qing Zhou, Jilei Liu, and Zhanqiang Cao. Requirements cybernetics: Elicitation based on user behavioral data. *J. Syst. Softw.*, 124:187–194, 2017.

[57] Walid Maalej, Zijad Kurtanovic, Hadeer Nabil, and Christoph Stanik. On the automatic classification of app reviews. *Requir. Eng.*, 21(3):311–331, 2016.

[58] Walid Maalej and Hadeer Nabil. Bug report, feature request, or simply praise? on automatically classifying app reviews. In *Proceedings of the 23rd IEEE International Requirements Engineering Conference, RE*, pages 116–125. IEEE Computer Society, 2015.

[59] Walid Maalej, Maleknaz Nayebi, Timo Johann, and Guenther Ruhe. Toward data-driven requirements engineering. *IEEE Software*, 33(1):48–54, 2016.

[60] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[61] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. 2008.

[62] Stuart McIlroy, Nasir Ali, Hammad Khalid, and Ahmed E. Hassan. Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering*, 21(3):1067–1106, 2016.

[63] Stuart McIlroy, Weiyi Shang, Nasir Ali, and Ahmed E. Hassan. Is it worth responding to reviews? studying the top free apps in google play. *IEEE Software*, 34(3):64–71, 2017.

[64] Stuart McIlroy, Weiyi Shang, Nasir Ali, and Ahmed E. Hassan. User reviews of top mobile apps in apple and google app stores. *Commun. ACM*, 60(11):62–67, 2017.

[65] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *ICLR (Workshop Poster)*, 2013.

[66] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining API popularity. In *International Conference on Testing, Practice and Research Techniques*, volume 6303, pages 173–180. Springer, 2010.

[67] Azad Naik and Huzefa Rangwala. *Large Scale Hierarchical Classification: State of the Art*. Springer Briefs in Computer Science. Springer, 2018.

[68] Maleknaz Nayebi, Henry Cho, and Guenther Ruhe. App store mining is not enough for app improvement. *Empirical Software Engineering*, 23(5):2764–2794, 2018.

[69] Maleknaz Nayebi, Homayoon Farrahi, Guenther Ruhe, and Henry Cho. App store mining is not enough. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 152–154. IEEE Computer Society, 2017.

[70] Kamal Nigam, Andrew McCallum, Sebastian Thrun, and Tom M. Mitchell. Text classification from labeled and unlabeled documents using EM. *Machine Learning*, 39(2/3):103–134, 2000.

[71] Nan Niu, Sjaak Brinkkemper, Xavier Franch, Jari Partanen, and Juha Savolainen. Requirements engineering and continuous deployment. *IEEE Software*, 35(2):86–90, 2018.

[72] Dennis Pagano and Walid Maalej. User feedback in the appstore: An empirical study. In *21st IEEE International Requirements Engineering Conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15-19, 2013*, pages 125–134. IEEE Computer Society, 2013.

[73] Fabio Palomba, Pasquale Salza, Adelina Ciurumelea, Sebastiano Panichella, Harald C. Gall, Filomena Ferrucci, and Andrea De Lucia. Recommending and localizing change requests for mobile apps based on user reviews. In *Proceedings of the 39th International Conference on Software Engineering,ICSE*, pages 106–117. IEEE / ACM, 2017.

[74] Fabio Palomba, Mario Linares Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In Rainer Koschke, Jens Krinke, and Martin P. Robillard, editors, *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 291–300. IEEE Computer Society, 2015.

[75] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 281–290. IEEE Computer Society, 2015.

[76] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. Ardoc: app reviews development oriented classifier. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, pages 1023–1027. ACM, 2016.

[77] Dae Hoon Park, Mengwen Liu, ChengXiang Zhai, and Haohong Wang. Leveraging user reviews to improve accuracy for mobile app retrieval. In *Proceedings of the 38th International*

ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015, pages 533–542. ACM, 2015.

[78] Nitish Patkar, Mohammad Ghafari, Oscar Nierstrasz, and Sofija Hotomski. Caveats in eliciting mobile app requirements. *CoRR*, abs/2002.08458, 2020.

[79] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *EMNLP*, pages 1532–1543. ACL, 2014.

[80] Henrique Pinto, Jussara M. Almeida, and Marcos André Gonçalves. Using early view patterns to predict the popularity of youtube videos. In *Proceedings of the ACM 6th International Conference on Web Search and Data Mining, WSDM*, pages 365–374, 2013.

[81] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[82] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[83] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *EMNLP/IJCNLP (1)*, pages 3980–3990. Association for Computational Linguistics, 2019.

[84] Georgios Rizos, Symeon Papadopoulos, and Yiannis Kompatsiaris. Predicting news popularity by mining online discussions. In *ACM 25th International Conference on World Wide Web, WWW*, pages 737–742, 2016.

[85] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann, editors. *Recommendation Systems in Software Engineering*. Springer, 2014.

[86] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, pages 410–420. ACL, 2007.

[87] Andrea Di Sorbo, Sebastiano Panichella, Carol V. Alexandru, Junji Shimagaki, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, pages 499–510. ACM, 2016.

[88] Andrea Di Sorbo, Sebastiano Panichella, Carol V. Alexandru, Corrado Aaron Visaggio, and Gerardo Canfora. SURF: summarizer of user reviews feedback. In *Proceedings of the 39th International Conference on Software Engineering, ICSE*, pages 55–58. IEEE Computer Society, 2017.

[89] Gábor Szabó and Bernardo A. Huberman. Predicting the popularity of online content. *Commun. ACM*, 53(8):80–88, 2010.

[90] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[91] Alexandru-Florin Tatar, Panayotis Antoniadis, Marcelo Dias de Amorim, and Serge Fdida. Ranking news articles based on popularity prediction. In *IEEE International Conference on Advances in Social Networks Analysis and Mining*, pages 106–110, 2012.

[92] Alexandru-Florin Tatar, Marcelo Dias de Amorim, Serge Fdida, and Panayotis Antoniadis. A survey on predicting the popularity of web content. *J. Internet Services and Applications*, 5(1):8:1–8:20, 2014.

[93] Michael E. Tipping. The relevance vector machine. In *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, pages 652–658. The MIT Press, 1999.

[94] Michael E. Tipping and Anita C. Faul. Fast marginal likelihood maximisation for sparse bayesian models. In Christopher M. Bishop and Brendan J. Frey, editors, *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics, AISTATS 2003, Key West, Florida, USA, January 3-6, 2003*. Society for Artificial Intelligence and Statistics, 2003.

[95] Manos Tsagkias, Wouter Weerkamp, and Maarten de Rijke. Predicting the volume of comments on online news stories. In *Proceedings of the ACM 18th Conference on Information and Knowledge Management, CIKM*, pages 1765–1768, 2009.

[96] Gias Uddin and Foutse Khomh. Automatic summarization of API reviews. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 159–170. IEEE Computer Society, 2017.

[97] Henk van der Schuur, Slinger Jansen, and Sjaak Brinkkemper. Becoming responsive to service usage and performance changes by applying service feedback metrics to software maintenance. In *23rd IEEE/ACM International Conference on Automated Software Engineering -*

Workshop Proceedings (ASE Workshops 2008), 15-16 September 2008, L'Aquila, Italy, pages 53–62. IEEE, 2008.

[98] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[99] Lorenzo Villarroel, Gabriele Bavota, Barbara Russo, Rocco Oliveto, and Massimiliano Di Penta. Release planning of mobile apps based on user reviews. In *Proceedings of the 38th International Conference on Software Engineering, ICSE*, pages 14–24. ACM, 2016.

[100] Phong Minh Vu, Tam The Nguyen, Hung Viet Pham, and Tung Thanh Nguyen. Mining user opinions in mobile app reviews: A keyword-based approach (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 749–759. IEEE Computer Society, 2015.

[101] Phong Minh Vu, Hung Viet Pham, Tam The Nguyen, and Tung Thanh Nguyen. Phrase-based extraction of user opinions in mobile app reviews. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 726–731. ACM, 2016.

[102] Yao Wan, Liang Chen, Jian Wu, and Qi Yu. Time-aware API popularity prediction via heterogeneous features. In *IEEE International Conference on Web Services*, pages 424–431, 2015.

[103] Chong Wang, Maya Daneva, Marten van Sinderen, and Peng Liang. A systematic mapping study on crowdsourced requirements engineering using user feedback. *J. Softw. Evol. Process.*, 31(10), 2019.

[104] Artinat Wattanaburanon and Nakornthip Prompoon. Method for classifying usability qualities and problems for action games from user reviews using text mining. In *15th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2016, Okayama, Japan, June 26-29, 2016*, pages 1–6. IEEE Computer Society, 2016.

[105] Grant Williams and Anas Mahmoud. Mining twitter feeds for software user requirements. In Ana Moreira, João Araújo, Jane Hayes, and Barbara Paech, editors, *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*, pages 1–10. IEEE Computer Society, 2017.

[106] Jonas Paul Winkler, Jannis Grönberg, and Andreas Vogelsang. Predicting how to test requirements: An automated approach. In Daniela E. Damian, Anna Perini, and Seok-Won Lee, editors, *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019*, pages 120–130. IEEE, 2019.

[107] Bo Wu, Tao Mei, Wen-Huang Cheng, and Yongdong Zhang. Unfolding temporal dynamics: Predicting social media popularity using multi-scale temporal decomposition. In *Proceedings of the AAAI 30th Conference on Artificial Intelligence, AAAI*, pages 272–278, 2016.

[108] Han Xiao. bert-as-service. `https://github.com/hanxiao/bert-as-service`, 2018.

[109] Xiaohui Yan, Jiafeng Guo, Yanyan Lan, and Xueqi Cheng. A biterm topic model for short texts. In Daniel Schwabe, Virgílio A. F. Almeida, Hartmut Glaser, Ricardo Baeza-Yates, and Sue B. Moon, editors, *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013*, pages 1445–1456. International World Wide Web Conferences Steering Committee / ACM, 2013.

[110] Zijun Yao, Yanjie Fu, Bin Liu, Yanchi Liu, and Hui Xiong. POI recommendation: A temporal matching between POI popularity and user regularity. In *Proceedings of the IEEE 16th International Conference on Data Mining, ICDM*, pages 549–558, 2016.

[111] Peifeng Yin, Ping Luo, Min Wang, and Wang-Chien Lee. A straw shows which way the wind blows: ranking potentially popular items from early votes. In *ACM International Conference on Web Search and Web Data Mining*, pages 623–632, 2012.

Appendices

Appendix A

Panichella Labeling Instructions

This Appendix contains a short summary of the guide that was provided to two graduate students groups as part of our effort to label the Panichella dataset with requirements related labels.

Table A.1: Meta data for the reviews in Panichella dataset

Column	**Explanation**
id	The review id (3439 reviews in total)
app_name	The application name (17 total applications)
version	The application version.
userid	The id of the user who wrote the review
date	The review date
rating	The review rating (1-5)
title	The review title
text	The review body

The provided dataset contains 3439 user reviews collected from 17 different applications with the metadata shown in Table A.1. First, you are expected to read the description of each application, go over their screen images, and make sure you understand the application functionality and purpose before you do any data labeling.

Second, you are expected to go over each review and provide a 0/1 value for each of the three labels.

Has_User_Experience. To set this binary column's value to 1, a review must contain at least a single sentence that discusses or expresses emotions towards a specific aspect of the application. They key idea here is that it describes a specific aspect/feature of the app that the user likes/hates, i.e., to capture reviews that tell the developers what features/aspects of the app are liked/hated by their users. In Table A.2 we provide some examples to help guide you in this process.

Table A.2: Examples of how to label a review with Has_User_Experience

Review	**Has_User_Experience?**
The UI is amazing	**Yes**. The sentence clearly describes a specific aspect of the app, i.e., the UI. In other words, the user is giving us information that he likes the UI or User Interface of the application
I like the predictive text	**Yes**. The expresses that he likes the aspect of the app where 'predictive text' is used
I love this app	**No**. While the sentence is describing an experience, it does NOT contain a specific aspect/feature of the app

Has_Feature_Request. To set this binary column's value to 1, a review must contain at least a single sentence expressing ideas, suggestions or needs for improving or enhancing the app or its functionalities. In Table A.3 we provide some examples to help guide you in this process.

Table A.3: Examples of how to label a review with Has_Feature_Request

Review	**Has_Feature_Request?**
It's a pity it doesn't support Chinese.	**Yes**. The user is asking to add Chinese support.
Messing around, wish there was a paint bucket, couldn't fill in face...	**Yes**. The user wants a 'paint bucket' feature added to the app
Facebook Login? Nope. App immediately deleted.	**Yes**. The user wants a 'facebook login' feature added to the app

Has_Bug_Report. To set this binary column's value to 1, a review must contain at least a single sentence describing technical issues with the app or unexpected behaviors. In Table A.4 we provide some examples to help guide you in this process.

Table A.4: Examples of how to label a review with Has_Bug_Report

Review	**Has_Bug_Report?**
The pop-up ads block the actual game area.	**Yes**. The user is describing a technical issue. The ads are blocking the game view
When I put in a photo to edit. I want it to be full screen. After the edit the image just looks weird and small. That's a problem I've had with the app.	**Yes**. The user is describing a technical issue. The edit feature seem to have a bug
Really enjoyed it till IOS8, can't get it to load properly now.	**Yes**. The user seem to have problems opening the application after iOS version 8

Why were those labels selected? Well, the user is providing information on a specific aspect of the application, i.e., not liking the ads (user experience). The review also does not have any feature requests and no application technical problems (no feature request and no bug report).

Important notes

- Please be aware that those label columns are inclusive. This means that a review can be labelled as *Has_User_Experience* and as *Has_Feature_Request* at the same time as a review may consist of several sentences each may fall under a different label. You're not supposed to pick one label per review, but rather analyze each sentence and find all the labels that apply to the given review.
- Please note that this a real-world dataset so you may have empty values for any of the review columns.