Extracting Non-Functional Requirements from Unstructured Text

by

Sahba Ezami

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

Abstract

Non-functional requirements (NFRs) of a software system describe desired quality attributes rather than specific user-visible features; NFRs model stakeholder expectations about pervasive system properties such as performance, security, reliability, and usability. Failing to meet these expectations can result in systems that, while functionally complete, may lead to user dissatisfaction and ultimately to the failure of the product. While NFRs may be documented, tracked, and evaluated in a variety of ways during development, there is no single common approach to doing so.

In this work, we investigate extracting information about NFRs that may be contained in source code and source code comments, since they are often considered to be the ultimate source of ground truth about a software system. Specifically, we examine how often NFRs are mentioned explicitly or implicitly in source code comments by using natural language processing (NLP) techniques, and we evaluate how effectively they can be identified using machine learning (ML). We modeled the problem as a text classification problem in which the goal is to identify comments about NFRs, and we evaluated the classifiers using example systems from the electronic health records (EHR) domain. The best performance was achieved using SVM classifier, with an F1 measure of 0.86. Our results indicate that using supervised method for our problem outperforms unsupervised methods which try to find common NFR patterns in comments. Comparing our results to previous studies shows that NFRs can be extracted more accurately from source code comments compared to other software artifacts (e.g., SRS or RFP documents). Moreover, we found that bag-of-words features are more effective compared to more complicated features (i.e., *doc2vec*) for the problem of extracting NFRs from source code comments.

Keywords: requirements engineering, non-functional requirements, natural language processing, machine learning, text classification

Acknowledgements

I am deeply grateful for my supervisor, Michael Godfrey, for his helpful guidance on my thesis.

I would like to thank Mei Nagappan, Ian Davis, and all other members of Software Architecture group (SWAG) for their great help and useful suggestions to my thesis.

Special thanks go to my family and friends for their support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the process of developing a software product, developers usually focus on designing and implementing a system with specific user-visible behaviors, or *functional requirements* (FRs). In addition to functional requirements, there is also another type of requirements that describe a set of quality attributes that a certain system should exhibit [33]. Such attributes enforce operational constraints on different aspects of the system's behavior such as its usability, security, reliability, performance, and portability [15]. These requirements specify criteria that can be used to judge the operation of a system as a whole, focusing on pervasive quality attributes such as performance rather than specific behaviors; these are called *non-functional requirements* (NFRs).

The ultimate success of a software system depends on how well both FRs and NFRs are elicited, modelled, and implemented [57]. The implementation of NFRs is often key to the user experience, and poor attention to quality attributes — such as performance, security, and usability — can lead to project failure [67]. One well known survey found that in software projects where NFRs are not considered, a failure rate of 60% or higher has been observed [3]. As an example case, a U.S. Army intelligence-sharing application, which cost USD$ 2.7 billion to develop, was found to be unusable when deployed within a realistic operating environment due to capacity, performance, and usability issues [57]. As another example, in the London Ambulance tragedy, the computer-aided dispatch system for ambulances lost track of its locating system, which led to the death of 46 people within only a few hours of its initial deployment [24]. It was reported that one of the main reasons for this failure was an incomplete set of requirements found in the requirement specification phase due to leaving out key stakeholders in the elicitation process. The delivered system had issues with reliability, performance, and scalability which made it fail in cases with invalid data and receiving a significant amount of incident information [21].

While FRs focus on the “what” functionalities are being implemented, NFRs strongly influence “how” they are implemented: e.g., how does the security model affect the user experience. Ideally, the NFRs of a system are consciously considered throughout all stages of development, from initial design through implementation and quality assurance to deployment and ongoing evolution [32]. In practice, they are often explicitly addressed late in development, during system-level testing, and in an ad hoc manner [39]. Recent studies have focused on the importance of extracting NFRs in the early stages of software development and from software documents produced before actual implementation of projects [17, 12, 26, 57]. An example of these documents is Software Requirements Specification (SRS) [20] document and Request For Proposal (RFP) [7].

Although developers may understand the importance of NFRs in large and complex systems, they may also neglect them if the NFRs are improperly documented within the development artifacts and processes [3]. Moreover, developers consume a considerable amount of their time and effort in dealing with functional requirements and have been found to often ignore NFRs due to schedule pressures. As a result, even in a well-documented system in which all of the requirements are elicited in the early stages, there is always a significant risk that the delivered system may not reflect the NFRs adequately and the software does not meet the desired quality criteria.

Motivated by these observations, we propose a method to extract NFRs from comments in source code. We believe that source code is often the best reference for understanding how and why a system has been implemented, and if the code is well-documented — which is often the case in large software projects — the focus is also reflected in the comments. Thus, by detecting comments about the NFRs in source code, we can track which parts of the code have changed to fulfill this kind of software requirements and better understand why a software system has evolved the way it has. Ultimately, the extracted NFRs can be compared to the NFR goals of the system that are mentioned in the documents generated in earlier stages of software development — such as SRS and RFP — to discover whether the system requirements are actually met in the delivered system or not.

The goal of this study is to assist analysts in extracting relevant NFRs from comments available in source code of software systems using automated natural language processing (NLP). To meet this goal, we gathered a dataset of comments from software systems in the electronic health records (EHR) domain and their corresponding labels which indicates whether a comment references NFRs or not. Knowing that a noticeable proportion of the comments are about the NFRs, we modeled our study as a text classification problem and identified the comments that are about either *security* or *reliability* as important software quality attributes in software systems. Using our proposed technique, developers can locate NFRs in the source code and develop high-quality software in a more organized manner.

1.1 Research Questions

This dissertation addresses four research questions:

RQ1: *How often are NFRs mentioned explicitly or implicitly in source code comments?*

RQ2: *How effectively can NFRs be extracted from source code comments?*

RQ3: *Which sentence characteristics are the most useful for extracting NFRs from source code comments?*

RQ4: *What are the differences and similarities between extracting NFRs from source code comments and other software documents?*

1.2 Contributions

This research makes the following contributions:

- We presented a dataset of source code comments that we gathered in the EHR domain. The dataset contains the text and location of comments and their labels indicating whether the comment belongs to a certain category of NFR or not.
- We proposed an automated method to find NFRs from unstructured text of comments available on the source code and locate NFRs in the source code.
- We evaluated empirical performance results for machine learning classifiers to identify NFRs from comments.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 reviews the background for this project which briefly provides the basis for software requirements engineering, machine learning techniques for text processing, and classification evaluation. Chapter 3 summarizes the related works in finding NFRs from software documents and also processing the text of source code comments to extract knowledge about software systems. Chapter 4 describes our proposed method for finding NFRs from source code comments and details of our implementation. Chapter 5 presents our evaluation process along with the results and discussion of our study. Finally, chapter 6 summarizes our work, and suggests possible future research directions.

Chapter 2

Background

This chapter provides the required background knowledge for this research. Section 2.1 describes software requirements engineering, explains the idea of NFRs, and describes several different kinds of NFRs. Section 2.2 provides an overview of machine learning and text classification. Finally, section 2.3 describes the evaluation techniques that are commonly used for comparing performance of text classification methods.

2.1 Requirements engineering

Requirements engineering can be defined as a coordinated set of activities for exploring, documenting, and maintaining the objectives, capabilities, qualities, and constraints that a particular software system should meet. Although requirements engineering is traditionally considered as the preliminary phase of software development in waterfall-like process models [52], most modern development processes are iterative and requirements engineering activities are performed repeatedly throughout development [65, 18, 29].

Within the literature, multiple frameworks have been suggested as models for the requirements engineering lifecycle [36]. In the widely-used categorization, Loucopoulos et al. categorized requirements engineering into four main activities: (I) *discovery* (more commonly referred to as *elicitation*), (II) *specification*, (III) *negotiation* and (IV) *validation and verification* [35]. During requirements discovery, analysts explore candidate requirements and assumptions that will shape the desired system, based on domain understanding, consultation with stakeholders, and utilizing other sources [65]. Requirements specification is the process in which the development team acquires, abstracts, and represents the requirements in SRS document. During requirements negotiation, stakeholders are aided by analysts to make informed decision to discuss, select, and prioritize the issues raised in the first two phases, and attempt to predict the behavior of the system before its implementation. The goal of the requirements validation and verification phase is quality assurance; in this phase, analysts ensure that requirements are consistent and reflect actual stakeholder needs [36]. A number of techniques are used in validation and verification phase including reviewing the SRS document, cross-referencing the SRS with other models to check for consistency, interviewing stakeholders, and providing simple scenarios and building prototypes to describe how the system will work once it is in operation [10].

Historically, software system requirements have been categorized into two classes, functional requirements (FRs) and non-functional requirements (NFRs). While the focus of functional requirements is user-observable operational properties of the system, NFRs focus on quality attributes of the system and constraints on the way the desired system should satisfy the functional requirements. However, the distinction between FRs and NFRs is fuzzy, and there are cases in which they overlap [65]. For example, many functional requirements in a firewall management system can be also considered as *security* requirements [65].

Since the number of NFRs categories can be large[1], some previous studies have worked on classifying them in a taxonomy [15, 1, 65]. Figure 2.1 shows a typical taxonomy that classifies NFRs into four main categories: *Quality of Service* (QoS), *compliance*, *architectural constraints*, and *development constraints* [65]. Quality of Service NFRs concern quality requirements of the software; they complement the "what" aspect of software with "how well" it should be met. Security, reliability, and performance are some of the most common NFRs in this category. The compliance category focuses mostly on external standards and organizational regulations, such as HIPAA — Health Insurance Portability and Accountability Act — for the medical domain that sets the standards for protecting sensitive patient data. This category focuses on the effects of software on the environment to conform to standards, national laws, and cultural and political constraints. Architectural constraints are mostly focused on the structural constraints of a developed software to fit its environment. Examples of this type of requirements are distribution constraints that focus on the distribution of data or geographic distribution of host and installation constraints that concerns running software smoothly on the target platform. The last category is development constraints that are mostly concerned with the attributes that regulate development of an anticipated software. This category include requirements on development costs, delivery schedule, variability of features, maintainability, reusability, and portability.

[1] Chung et al. identified 156 NFR categories [15]

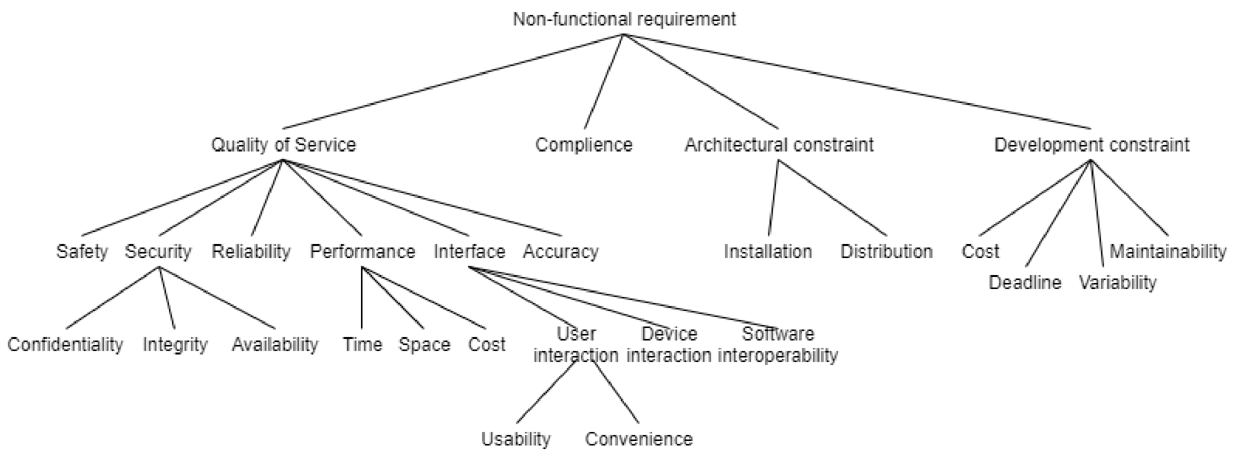

Figure 2.1: A taxonomy of non-functional requirements [65]

2.2 Machine learning for text processing

While there is a wide variety of techniques and algorithms in machine learning, they can be divided into two main categories: *supervised learning* and *unsupervised learning*.

The goal of supervised learning is to infer a function from labeled training data that consists of data instances and the desired output value for each instance. The inferred function can then be used to predict the label for the unseen data instances. In contrast, unsupervised learning infers a function to describe the hidden structure of the unlabeled data. Thus, it searches data for common patterns, and then groups (clusters) of common instances are created. Besides these two common categories, there is another category named *semi-supervised learning*, that falls between supervised learning and unsupervised learning [13]. In semi-supervised learning, the goal is to infer a function based on small amount of labeled training data and a large amount of unlabeled data.

The goal of our work is to extract NFRs from the natural language text of source code comments i.e., to identify whether a comment is about any of the categories of NFRs or not. This can be modeled as a supervised learning task in which the goal is to create an NFR classifier from a provided labeled dataset. Another approach to solve this problem is to assume that all of the sentences about a certain category of NFR share a common pattern, and use NLP techniques to learn and extract these patterns. Thus, unsupervised learning can be utilized to extract these NFR patterns from the comments. Throughout this thesis, we have used Latent Dirichlet Allocation (LDA) algorithm [9], which is an unsupervised topic modeling method, and also text classification, which is a supervised algorithm to model our problem.

To be able to use unstructured text for classification, the first step is to convert the text to a vector of features to feed into the classifiers. In this thesis, other than using a simple bag-of-words model to convert text to vector, we have used *doc2vec*, which is an unsupervised model to convert text to the vector of semantic features and used the generated vectors to train our supervised model [34].

2.2.1 Latent Dirichlet Allocation (LDA)

Topic modeling techniques are a suite of statistical models that analyze the words in the original text of unstructured documents to find any themes that might run through the text, the connection between these themes, and their change over time [8]. The intuition is that if groups of words occur together often, then there is likely an underlying common theme, or *topic*, that they represent [9]. For example, if a text document has the words "player, game, win, lose, stadium" with relatively high frequency, one might guess that the topic of this document is "sport". Topic modeling algorithms do not require prior labeling to find the topics in the text. Thus, they can be categorized among the unsupervised techniques with the goal of finding latent topic structure from a text corpus.

Latent Dirichlet Allocation (LDA) is one of the most popular and simplest topic modeling algorithms [9]. The main intuition behind LDA is that each document can belong to several topics. LDA is most easily described by its generative process, the imaginary process by which it is assumed each text document is generated [8]. In this model, it is assumed that each document is generated in two steps: First, a set of topics is chosen for the document according to the distribution of topics. Then, to generate each word in the document, a topic is selected according to the distribution in the first step, and then a word is randomly selected according to the distribution of words for each topic. In this way, it generates a set of words and put it together as a document. Within this model, each text document is represented as a bag-of-words, that is a simple representation of a text as a multiset of words, disregarding grammar and order of words in the text. Figure 2.2 shows the generation process for a sample text document. The topics (left side of the figure) shows the distribution over words available in the entire text corpus. For each word, first a topic (colored coins) is chosen according to the topic distribution (right side of the figure), and then a word is then chosen from the corresponding topic [8].

The goal of LDA algorithm is to automatically find the topics for each document, given a set of documents. In this problem, the unstructured text documents are observed, while

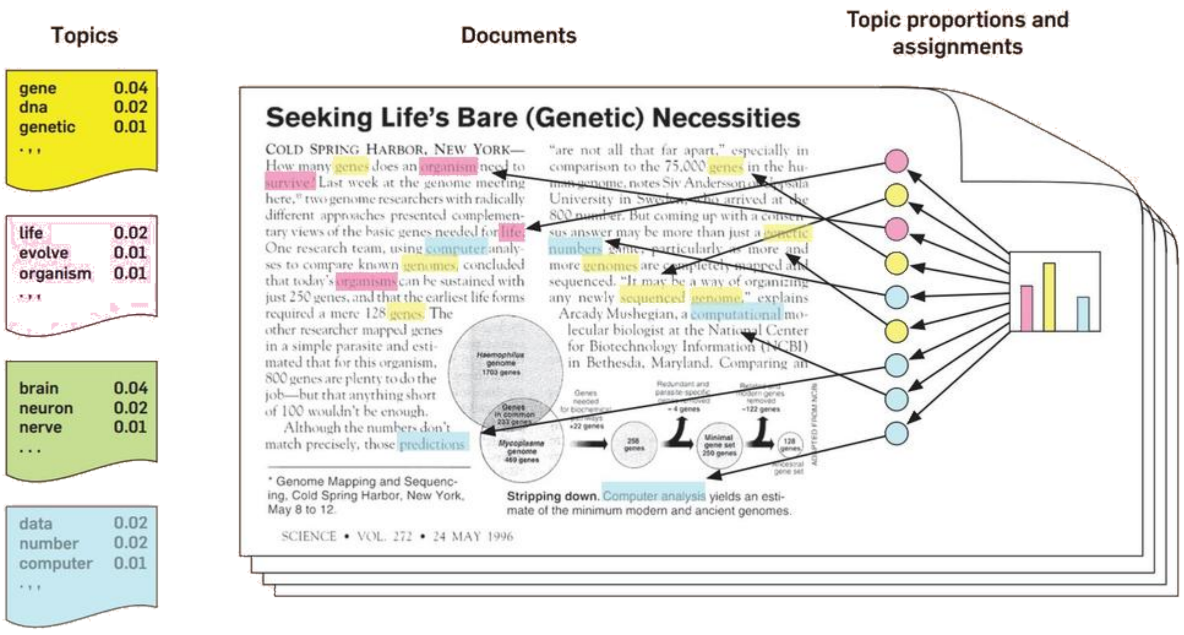
Topics
Documents
Topic proportions and assignments
gene 0.04
dna 0.02
genetic 0.01
life 0.02
evolve 0.01
organism 0.01
brain 0.04
neuron 0.02
nerve 0.01
data 0.02
number 0.02
computer 0.01
Seeking Life's Bare (Genetic) Necessities
Stripping down. Computer analysis yields an estimate of the minimum modern and ancient genomes.
SCIENCE • VOL. 272 • 24 MAY 1996

Figure 2.2: The intuition behind LDA and its generative process for text documents [8]

the topic distribution for each document and the word distribution for each of the topics are hidden structures. The central computation problem for LDA is to infer the hidden topic structure by using the observed text documents, that can be assumed as reversing the imaginary generative process. Therefore, the input needed by LDA is a text corpus and a user-defined number of topics. The model has two outputs that determine the latent statistical structure of the text corpus: (i) the word distribution for each topic, and (ii) the topic distribution for each text document.

LDA model has several features that make it useful for different domains. One of these features is that LDA enables a low-dimensional representation of text that uncovers the latent semantic relationships and also allows faster analysis on the text [69]. Moreover, LDA is an unsupervised method which means that there is no need for providing the labeled dataset to learn the model. In many of the domains, providing the label for documents is a laborious and time-consuming task. So, using unsupervised methods like LDA can be helpful in those domains.

One drawback of LDA is the topics found by this technique are unlabeled, and human experts are usually needed to decide on an appropriate name for the topics according to the distribution of the words for each topic. Another drawback of this approach is that number of topics must be chosen by the user before the analysis, and may not correspond to the number of distinct topics that a human expert might feel is present in the dataset. Recently, there have been some efforts in automatically suggesting an appropriate value for the number of topics for a given dataset, but the proposed methods are still inaccuracte and the area needs more effort [2].

2.2.2 Text classification

The goal of classification is to infer a function from labeled training data that consists of data instances and the desired output value for each instance. The inferred function can then be used to predict the label for the unseen data instances. In contrast, unsupervised learning infers a function to describe the hidden structure of the unlabeled data. Thus, it searches data for common patterns and groups (clusters) of common instances are created.

For the classification tasks, we used two popular classifiers that are known to perform well on text data: Naïve Bayes classifier [40] and Support Vector Machine (SVM) [60, 30]. Naïve Bayes is a probabilistic classifier based on applying Bayes theorem [31]. In this classifier, it is naïvely assumed that each of the features in data instances is independent of other features, which may not be true in practice. The goal of the model is to find the class that maximizes the conditional probability of belonging to the class, given the data instances. The model calculates this conditional probability for each class with the independence assumption and using Bayes theorem to calculate conditional probability for each class. Then, it assigns the data instance to the class with maximum conditional probability. Although the model seems oversimplified, it is known to work effectively on real-world problems; the reason is that although the probability estimations of Naïve Bayes is not accurate because of the naïve independence assumption, its classification decision is accurate [14]. In SVM classifier, the objective is to find a separator with the maximum margin between the classes. Thus, suppose that we are dealing with data instance of p dimensions. SVM classifier tries to find a (p-1)-dimensional hyperplane in space that represents the highest possible separation (margin) between the classes.

2.2.3 Doc2vec

When it comes to text processing, most approaches use a simple bag-of-words to convert the text to the vector of fixed length. Although the bag-of-words model performs accurately in many cases, it ignores the semantic meaning of the words and their context. With the progress of machine learning, it has become possible to train more complex models that typically outperform simple models [4]. For example, neural network-based language models outperform N-gram models on larger datasets and predictor of the words given the context for neural network-based language models was the closest to true model of the language [43, 42]. However, in practice the training time required to create these models often make their use infeasible.

Mikolov et al. suggested two models focused on distributed representations of words learned by neural networks to overcome the above-mentioned shortcomings: Continuous Bag-of-Words Model (CBOW) and the Continuous Skip-gram Model (Skip-gram) [42]. CBOW model moves through all words and learns to predict the current word based on the surrounding words. The Skip-gram model is very similar to CBOW, but instead of predicting the current word based on the context, it predicts the surrounding words in a sentence or document based on the current word. Figure 2.3 shows the graphical representation of CBOW and Skip-gram models.

It has been shown that representation of words as vectors in this model goes beyond syntactic representation, and the vectors can be used to capture semantic relationships between words, such as the city and the country it belongs to. Semantic analysis is possible using simple algebraic operations on word vectors. As an example, the result of a vector calculation vec(“Madrid”) - vec(“Spain”) + vec(“France”) is closer to vec(“Paris”) than to any other word vector [42, 44].

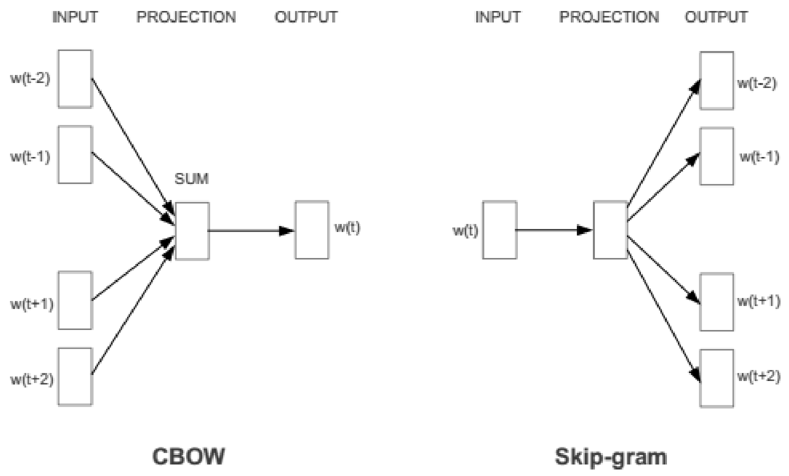

Figure 2.3: Representation of CBOW and Skip-gram models. CBOW predicts the current word based on the context, and Skip-gram predicts surrounding words given the current word [42].

These models, known as *word2vec*, can be used to convert a word to a fixed length vector that takes into account the semantics of the word. Le et al. proposed an extended version of CBOW and Skip-gram models called *Paragraph Vector* or *doc2vec* [34]. Paragraph Vector is an unsupervised algorithm that learns vectors of fixed length for larger blocks of text, such as sentences, paragraphs, and documents. Unlike the bag-of-words represenatation, paragraph vectors have the power of taking into account the ordering of words in text and also semantics of each word. Thus, Le et al. showed that paragraph vector can outperform bag-of-words and N-grams for text representation [34].

2.3 Classification Evaluation

In our work, we have used various types of classification techniques as well as a variety of features passed to these classifiers. To evaluate the performance of each classifier and compare the results, we used precision, recall, and F1 measure. For binary decisions (yes/no), the classifier's prediciton can be divided into one of four categories:

- True Positives (TP) are correct predictions when the classification under evaluation (e.g., NFR) is implied by the sentence and also predicted by the classifer.

- True Negatives (TN) are correct predictions when the classification under evaluation (e.g., NFR) is not implied by the sentence and also not predicted by the classifer.
- False Positives (FP) are incorrect predictions when the classification under evaluation (e.g., NFR) is not implied by the sentence but predicted by the classifer.
- False Negatives (FN) are incorrect predictions when the classification under evaluation (e.g., NFR) is implied by the sentence but not predicted by the classifer.

Having computed these values, we used precision, recall, and F1 measure to evaluate our clasifiers. Precision (P) is the proportion of correctly predicted classifications against all of the predictions under test.

$$P = \frac{TP}{(TP + FP)} \tag{2.1}$$

Recall (R) is the proportion of correctly predicted classifications against all of the instances with “yes” as their actual class.

$$R = \frac{TP}{(TP + FN)} \tag{2.2}$$

In general, maximizing recall and precision simultaneously is not feasible [11]. F1 measure is the harmonic mean of precision and recall, giving equal weight to each of them.

$$F1 = 2 * \frac{P * R}{P + R} \tag{2.3}$$

In the context of NFR extraction, higher recall implies that more of the sentences related to NFRs are being found in the text, while lower recall implies NFR-related sentences are missing from the model. High precision means that from the sentences that the model has chosen as related to NFR, most of them are indeed NFR-related sentences; while low precision implies that from the sentences found as NFR-related by the model, a high ratio were not related. From the NFR extraction perspective, we consider that recall is more important than precision since we wish to extract all of the relevant NFRs mentioned in the software documents and source code. However, we try to maximize precision as much as we can, since low precision will frustrate users.

2.4 Summary

This chapter reviewed some of the background knowledge for our study. Since the problem we are addressing is automatically extracting NFRs from source code comments, during this chapter we have provided the basis for requirements engineering, machine learning techniques used for text processing with the focus on the algorithms we used for this study, and also methods used for evaluating the performance of classification algorithms. In the next chapter, we will review some of the most related previous studies to this thesis.

Chapter 3

Related Work

The research areas most similar to our project can be divided into two main categories: extracting NFRs from software artifacts, such as SRS and RFP, and analyzing the content of source code comments to extract semantic knowledge. In this chapter, we briefly review notable works related to each of these two categories.

3.1 Extracting NFRs from software artifacts

Cleland-Huang et al. performed the first notable exploration of NFR extraction and classification from software development artifacts [17]; in this work, they attempted to detect and classify NFRs for 15 SRS documents developed as term projects by MS students at DePaul University. Their approach is based on the assumption that different types of NFRs can be distinguished by certain keywords known as *indicator terms*. Thus, they trained some NFR-specific indicator terms for each of the NFRs from the training data, and then tried to classify each given document as one or more class of NFR according to the function of the occurrence of indicator terms. The model worked well for detecting the most of the NFRs appearing in the text, with a classification recall of 81.2 %. However, the precision was low (12.4 %) due to a high rate of false positives. They also made their dataset available to the PROMISE repository [41], and it since been used by the other authors to build their models and compare it to the original work.

Using the PROMISE dataset, Casamayor et al. replicated the study of Cleland-Huang et al. using a semi-supervised approach, so as to decrease the need for manually labeled data and to improve the results of the original work [12]. They used a multinomial Naïve

Bayes classifier coupled with an *Expectation-Maximization* (EM) algorithm [46] to boost the performance of the classifier. To this end, they provided labels for each instance of unlabeled data based on the function learned by the small labeled training set. The results provided by this work approach the maximum theoretical performance on the data set with a given algorithm. Empirical evaluation of the approach showed higher classification accuracy compared to the cases in which a fully supervised approach was used.

Zhang et al. repeated the experiment in 2011 using an SVM classifier with a linear kernel as their classifier [70]. They investigated the effect of different features, including original words, N-grams, and phrases in detecting NFRs. Interestingly, they found that using single original words with Boolean weight was the most effective approach for this problem. Compared to the original study performed by Cleland-Huang et al., they reported higher precision but lower recall on the same dataset.

The approaches that used PROMISE NFR dataset are similar to our project since they all used supervised learning approaches to solve the NFR detection problem. They differ because they tried to extract NFRs from various software documents, while we extracted them from the comments in source code. The data they used for the project is not directly related to the EHR domain. However, we included the data in part of our training to be able to compare our approach to the previous studies.

Another approach for finding NFRs from text is to look for semantic similarities in text and try to extract NFRs based on observed semantic patterns. Using this idea, Hindle et al. used topic modeling to find the topics from commit-log comments recovered from source control systems such as CVS and BitKeeper [26]. Their idea was to relate the topics back to the NFR categories, and to this end, they used a cross-project taxonomy of NFRs to assign NFR classes to each extracted topic. More specifically, they used a wordlist of top words for each NFR category and compared the words of the topics with the words of the wordlists to decide whether the topic is about any of the NFR categories or not. With ROC score between 0.6 and 0.8, they claimed that labeled topic extraction is an accurate approach to find NFRs in text. We used their method in a part of our study to examine the performance of topic modeling for comments. However, their application is different since they used commit log messages and mailing list of developers for their research.

The study conducted by Zou et al. [71] used the method proposed by Hindle et al. [26] to find NFRs from posts in StackOverflow and the comments on posts. They tried to find answers to three questions: (I) Which NFRs are discussed most and least frequently? (II) For which NFRs are questions most likely to remain unanswered? (III) What evolutionary trends of NFRs exist with respect to time? By analyzing the topics of posts and comments in StackOverflow, the authors found that developers are mostly concerned about usability and reliability. In contrast, they pay less attention to maintainability and efficiency when they are coding. They also found that more than 80% of the questions about usability remain unresolved, which suggests that developers need more help in this area. Moreover, by analyzing topic tends over time, it was found that usability was always a hot topic among developers, while functionality and reliability became more troublesome over time.

In a recent study, Mahmoud et al. [38] tried to extract NFRs and trace them to their implementation in source code. In the NFR extraction part of their approach, they extracted keywords from requirements documents and then clustered these words according to their semantic similarity calculated by using *Normalized Google Distance* (NGD) of words [16]. Then, they classified each cluster as NFR classes based on the semantic similarity between the functional requirement and the clusters of words representing individual NFRs. They evaluated the proposed method on SRS of three different projects. The average recall reported for each of the projects was at least 74%, with the least precision of 50%.

The work that is most related to our research is the method proposed by Slankas et al. [57] to find NFRs from sentences in unstructured text. To that end, they collected a series of 12 documents of various types — including Data Use Agreements (DUA), install manuals, regulations, RFPs, SRS, and user manuals — from different projects in the EHR domain. They developed a framework called *NFR-Locator*, based on machine learning and NLP techniques to classify each sentence in natural language text documents into their appropriate NFRs; they used a mode with 14 NFR categories, including access control, security, maintainability, and reliability.

NFR-Locator identifies NFR-related sentences in a text document in two steps. In the first step, natural language text is parsed into an internal representation called "Sentence Representation (SR)" based on the Stanford Type Dependency Representation (STDR) [22]. SR is a tree-like representation in which each sentence is represented as a directed graph where vertices are words and edges are relationships between words. In the second step of the process, SRs are used to classify the sentence into specific NFR categories or "not applicable" category if it does not specify any NFRs. They proposed a modified version of the k-nearest neighbor (k-NN) classifier that computes a custom distance function based on similarity of SRs and assigns NFR categories based on similar sentences.

The highest accuracy was achieved by using Sequential Minimum Optimizer (SMO), with 0.60 in F1 measure. The suggested k-NN had an F1 measure of 0.54, outperforming the optimal Naïve Bayes classifier with F1 measure of 0.32. They found that all of the evaluated documents contained NFRs; however, they found that the types of NFRs available in each document can vary. For example, DUA documents contained high frequencies of legal and privacy NFRs compared to other types of documents. Moreover, in analyzing specific

NFR categories, they found particular features unique to each category that make them a suitable candidate to use as a feature in classification and improving the accuracy of the models.

Riaz et al. [51] extended the application of Slankas et al. [57] by focusing on security requirements and by generating more comprehensive and classified set of requirements. The process takes natural language text documents as input and identifies security-relevant sentences in the documents using customized k-NN classifier [57] and classifies them according to the security objectives (such as confidentiality, integrity, availability). By observing similarities and common elements in the classified set of security-relevant sentences, they derive a set of context-specific security requirements templates and suggest them to requirements engineers to generate the security requirements. Moreover, similar studies were performed by the same authors to extract access control NFRs from text documents [56, 58].

We built our proposed framework on top of *NFR-Locator* suggested by Slankas et al. [57] and used their data for part of our training. However, the goal of our study is to find NFRs in the comments developers write in source code of software projects. Although the approaches we used are similar to previous studies, none of the studies so far has attacked the problem for source code comments.

3.2 Analyzing the content of source code comments

Ying et al. [68] analyzed the content of comments with the purpose of exploring the widely varying ways in which comments may be used. The primary focus of the project was "Task" comments that Java programmers use in their project. To this end, they checked out 2,231 files from an IBM internal codebase, the Architects Workbench (AWB), and analyzed 221 Eclipse task comments that were present in those files. As a result, they found that programmers not only use comments for explaining the source code and describing tasks but also for many other purposes, such as communication with their colleagues. Thus, these comments can be considered as useful inputs for mining project information.

Padioleau et al. [47] designed a taxonomy to classify contents of comments programmers write in the source code. They believe that leveraging comments can help in improving software reliability and a better understanding of programmers needs. The study was initially performed on a dataset of 1050 comments randomly sampled from the latest versions of Linux, FreeBSD, and OpenSolaris.

The comment classification was performed from different aspects on the basis of four "Wh" questions:

- *What?* concerns the “Content” of the comment
- *Who?* concerns “Beneficiary” (who benefits from the comment e.g., testers) and “Author” (who wrote the comment e.g., an expert or a beginner)
- *Where?* concerns “Code entity” (the location of comment in the file (e.g., header, before a loop, function) and “Subsystems” (which subsystems the comment is located in e.g., file system)
- *When?* concerns “Time” (when the comment was written) and “Evolution” (how comments evolve over time)

The research mostly focused on the content of the comment, that is the answer to the “What” question. They found that while many comments are simply explanations of the code, 52.6% of the comments can be used by existing tools — such as annotation languages, bug detection tools, editor features, and programming languages — or inspire new tools. The authors labelled these comments as “exploitable”, since they can be useful for enhancing software reliability and detecting bugs. In this study, they found that (1) many comments describe code relationships, code evolution aspects such as cloned code, deprecated code, TODOs, or the usage and meaning of integers and integer macros, (2) a significant number of comments could be expressed by existing annotation languages, and (3) many comments express synchronization-related concerns, although they are not well supported by annotation languages.

In a similar study, Steidl et al. [59] analyzed the content of comments with the purpose of providing a model for comment quality based on different categories. They used a machine learning classification algorithm to categorize each comment according to its inferred type: *Copyright*, *Header*, *Member*, *Inline*, *Section*, *Code*, and *Task*. The classifier was trained on a dataset of 1330 comments from twelve open source projects in Java and C++. The evaluation shows that using a J48 decision tree achieved the best performance for this problem with a weighted average precision and recall of 96%. They assessed the coherence between Member comments — i.e., comments that describe the functionality of a method/field — and the name of the corresponding method. Their coherence evaluation determines whether the Member comment provide a useful description of the method. They also used the length of Inline comments — i.e., comments within a method body — as an indicator of their coherence to the following lines of code. Intuitively, shorter inline comments contain less information compared to the longer ones. However, the role of very short or very long comments has not been investigated. They suggested to use short comments — with at most two words — indicates redundant information and helps developers to find parts of the code that should be refactored. Moreover, they found that very long comments — with more than 30 words — contain significant global information. Although the long comments can lead to better understanding of code, having too many long comments indicates a missing information in other software documents for the corresponding software system.

Tan et al. have proposed several techniques to automatically extract knowledge from source code comments [62, 63, 64]. These studies used comments mainly to improve software quality by detecting bugs — where the source code does not follow the assumptions mentioned in the comments — and inappropriate comments — where the comments are misleading and inconsistent with the source code. These techniques have extracted rules from comments and applied these rules to detect inconsistencies between what is said in the comments and what is implemented in the code. As an example case, the rules concerning lock and synchronization can be extracted from both source code and comments automatically. According to the extracted rules, if the comment says that a lock is needed within a method, but there is no acquisition found in the code, the code is not consistent with the comment [62].

Identifying common topics of the comments can help in extracting related information for each topic. To discover the topics of comment, Tan et al. proposed two kinds of topic miners: *Hot-Word-Miner* and *Hot-Cluster-Miner* [62]. Both of these miners rely on various NLP and statistical techniques to preprocess each comment and break it down to its individual constituent pieces — such as words and phrases — and their corresponding Part of Speech (POS) tags. Hot-Word-Miner uses simple word frequencies to find the most common words in the corpus that can later be used to identify common topics by the user. In Hot-Cluster-Miner, instead of using a simple word count, they clustered the related words together and proposed a more sophisticated model for finding topics.

Having extracted topics and topic-related comments using the two topic miners described above, the study performed by Tan et al. focused on the analysis of C/C++ comments regarding locks and call relationships [62]. In subsequent studies, they focused on analyzing comments related to interrupts and locks [63], and utilized dynamic testing combined with analysis of Javadoc comments concerning null pointers and exceptions [64].

According to Tan et al., most comments can be categorized into one of two classes: comments that descibe the code, and comments that explain programming rules [61]. For the studies performed by Tan et al. [62, 63, 64], the second type is more important since it focuses on specifying rules that programmers are supposed to follow. Although comments like this can be beneficial for bug detection and prevention goals, they are less helpful for our problem. To find NFRs from comments, we focus only on the content of comments describing the code and purpose of implementing it.

3.3 Summary

This chapter highlighted some previous research that is closely related to our work. We divided the similar studies into two main categories: studies performed for extracting NFRs from text of various software artifacts such as SRS, DUA, and RFP, and the previous works related to analyzing the content of source code comments.

Our initial investigation shows that most of the previous studies utilized machine learning and NLP techniques to extract semantic knowledge from the text of software artifacts, especially comments. However, as yet there is no unique solution that solves the problem of NFR extraction considering the characteristics of source code comments and its similarities and differences to other artifacts. In the next chapter, we will present our proposed approach for extracting NFRs from the source code comments.

Chapter 4

Methodolgy

The goal of this work is to locate NFRs in source code by using automated NLP techniques. In order to do that, we find comments that concern NFRs from all of the comments available within the source code of a set of open-source software projects. We narrowed our focus to *security* and *reliability* NFRs, since they are the two most common types of NFRs that developers mentioned in our dataset of comments from the EHR domain. Therefore, given a dataset of comments, we aim to identify which of the comments are about *security* or *reliability* requirements and which are not. By detecting this kind of comments successfully, we would be able to take a first step in the more general problem of locating where any NFRs are implemented within a code base, and to tag those locations for possible future reference. This can help developers and project stakeholders to ensure these requirements are met in the development process.

Tracking whether the source code meets the requirements or not can be costly and time-consuming, especially in cases that scale of the project grows very fast. Our goal is to use machine learning techniques to extract NFRs in a cost-effective way automatically. To do this, we provided a set of data from comments on the source code, as well as labels for training and evaluation purposes (section 4.1). Before any further analysis, we preprocessed and removed noise from the data (section 4.2). We suggested two approaches to extract NFRs from source code comments. In the first approach, we analyzed the topics that developers talk about in the comments and related those topics back to NFRs to solve the problem in an unsupervised manner (section 4.3). In the second approach, we modeled the problem as a supervised text classification problem and tried to classify each comment as related or not-related to NFRs. Section 4.4 describes details of methodologies we used for this project.

Table 4.1: Type of software documents and projects used for each document type in dataset of software documents provided by Slankas et al. [57].

Document Type	Projects
SRS	CCHIT Ambulatory Requirements, iTrust, PROMISE NFR dataset
Installation manual and user manual	OpenEMR
DUA	Two documents Centers for Medicare & Medicaid Services 9 and the North Carolina Department of Public Health
RFP	Two documents from organizations within the state of California for EHR system
Code of Federal Regulations (CFRs)	Three sections of the United States Code of Federal Regulations related to healthcare

4.1 Dataset

Most of the previous studies in the literature of extracting NFRs from text has been performed in the Electronic Health Record (EHR) domain and provided data in this field. To be able to compare our proposed approach with previous works and also continue the efforts in this area, we focused on the EHR domain in our projects and provided a dataset of comments in this field. The rest of this section describes the details of the datasets that we utilized[1].

4.1.1 Software documents

We used the dataset from the EHR domain that was collected by Slankas et al. [57] for parts of our training and comparing the characteristics to our dataset of comments. The dataset contains five types of software documents from 12 projects in the EHR domain. The type of these documents and projects used for each type is listed in table 4.1. The data is available in original format, `txt` format, `json` and `arff` format[2].

The labels provided for this span 14 NFR categories: *legal*, *look and feel*, *privacy*, *reliability*, *recoverability*, *audit*, *maintainability*, *operational*, *security*, *usability*, *access control*,

[1]The data is available on: https://github.com/Sahba-e/NFRExtaction

[2]https://github.com/RealsearchGroup/NFRLocator

Table 4.2: Statistics of dataset of software documents. For each document, the number of total sentences (Size) and number of sentences related to each NFR category is shown.

Document	Document Type	Size	AC	AU	AV	LG	LF	MT	OP	PR	PS	RC	RL	SC	US	OT	FN	NA
CCHIT Ambulatory Requirements	SRS	306	12	27	1	2	0	10	0	0	1	5	2	28	4	8	228	6
iTrust	SRS, Use Case	1165	439	44	0	2	2	18	2	9	0	9	9	55	2	0	734	376
PromiseData	SRS	792	164	20	36	10	50	26	89	7	75	4	12	71	101	19	340	0
Open EMR Install Manual	Installation Manual	225	3	0	0	0	0	0	5	1	0	6	1	25	0	0	2	184
Open EMR User Manual	User Manual	473	169	0	0	0	14	0	0	0	0	0	0	8	4	0	286	95
NC Public Health DUA	DUA	62	1	0	0	20	0	0	0	4	0	0	0	1	0	0	0	41
US Medicare/Medicaid DUA	DUA	140	1	0	0	26	0	0	0	17	0	0	0	0	0	5	2	108
California Correctional Health Care	RFP	1893	94	120	9	85	0	133	94	52	13	16	13	193	14	38	987	409
Los Angeles County EHR	RFP	1268	58	37	8	3	2	28	19	3	11	8	13	108	21	10	639	380
HIPAA Combined Rule	CFR	2642	28	8	3	0	0	78	0	213	0	9	0	41	1	0	317	2018
Meaningful Use Criteria	CFR	1435	0	0	0	0	0	0	0	0	0	0	0	8	0	0	116	1311
Health IT Standards	CFR	1475	10	20	0	0	0	119	0	1	0	2	2	71	1	2	164	1146
Total		11876	979	276	57	152	68	413	207	300	100	50	43	563	148	82	3568	6076

Notes: AC: access control, AU: audit, AV: availability, LG: legal, LF: look and feel, MT: maintainability, OP: operational, PR: privacy, PS: capacity and performance, RC: recoverability, RL: reliability, SC: security, US: usability, OT: other, FN: functional, NA: Not Applicable

capacity and performance, *availability*, and *other* NFR categories. The *other* category is added to cover NFR sentences that did not readily fall into any other existing categories. As it was claimed by Slankas et al. [57], this category helps in terms of removing noise from the data in each category and improving the performance of machine learning algorithms. Table 4.2 shows the number of sentences per category and in total. From the table, we can see that functional and database design categories have the most number of sentences since they refer to more general NFR concepts.

To use the dataset for our project, we found that the NFR categories defined in this dataset are narrow, providing more details than is needed for our project. In the original work, the authors mentioned that they separated *access control* and *audit* from *security* to meet the future research needs. As we do not need to separate these concepts, we merged *security*, *audit*, *access control*, and *privacy* classes in the training set into one more general class named *security*. Then we removed the labels for all of the classes except for *security* and *reliability*, that is the focus of our study. Table 4.3 shows the statistics for the modified version of the software documents dataset.

4.1.2 iTrust comments

The dataset of comments that we used for this project is from iTrust, which is an open source medical application within the EHR domain [66]. This project involves the development of an application that can help patients track their medical history and contact their selected medical professionals. It also helps medical professionals to obtain and share information about their patients and their medical records.

Table 4.3: Statistics of dataset of software documents after merging security, audit, access control, and privacy requirements into one single category named *security*. For each document, the number of total sentences (Size), number of sentences related to security requirement (SC), and number of sentences related to reliability requirement (RL) is shown.

Document	Document Type	Size	SC	RL
CCHIT Ambulatory Requirements	SRS	306	67	2
iTrust	SRS, Use Case	1165	547	9
PromiseData	SRS	792	262	12
Open EMR Install Manual	Installation Manual	225	29	1
Open EMR User Manual	User Manual	473	177	0
NC Public Health DUA	DUA	62	6	0
US Medicare/Medicaid DUA	DUA	140	18	0
California Correctional Health Care	RFP	1893	459	13
Los Angeles County EHR	RFP	1268	206	13
HIPAA Combined Rule	CFR	2642	290	0
Meaningful Use Criteria	CFR	1435	8	0
Health IT Standards	CFR	1475	102	2
Total		11876	2171	43

Table 4.4: The six categories of NFRs listed in SRS document of the iTrust project and the description for each NFR.

NFR #	NFR	Description
1	HIPAA	Implementation must not violate HIPAA guidelines.
2	Exclusive authentication	The system shall enable multiple simultaneous users, each with his/her exclusive authentication.
3	Form validation	The form validation of the system shall show the errors of all the fields in a format the same time.
4	Reports	A report is a page which opens in a separate window and contains minimal decoration. The format is printer-friendly in that the background is white and the information does not exceed the width of 750 pixels so that upon printing, no information is lost due to the information being too wide.
5	Privacy policy	The system shall have a privacy policy linked off of the homepage. The privacy policy should follow the template provided here.
6	Security of MID	Remove MID from being displayed on all pages and URLs. MIDs should be considered private, sensitive information.

For the iTrust project, we have access to its SRS document and to the source code of the project. The SRS includes diagrams and descriptions of the project's use-cases, functional requirements, NFRs, and constraints on development process such as programming language and coding standards. In particular, six NFRs are stated in the SRS document and listed in table 4.4.

Knowing the NFRs of the system, we collected the comments on the source code of version 21 of the iTrust project. The project contains 432 Java source code files, 444 classes, and 53727 lines of code. By processing all of the files with `.java` extension in the source code of the project, we generated a dataset of 7726 comments from source code. According to the syntax of comments in Java, we extracted any blocks of text in one of the following forms as a single comment:

- A line of text starting with //

Table 4.5: Statistics of the dataset of comments; number of comments related to each category and percentage of the category in the dataset

Category	Number	Percentage
Reliability	1316	17.9
Security	73	1.0
Other NFRs	3	0.04
Total	1392	18.9

- A block of text (containing one or more lines) starting with /* and ending with */

Since we collected the dataset of comments to use for extraction of NFRs, we provided each comment with a label that makes it appropriate to use for training and evaluation purposes. We performed the labeling process manually by reading all of the comments available in the dataset. The NFR categories that we were specifically looking for in the source code was based on the intersection of the six categories mentioned in the SRS document and 14 categories that were defined by Slankas et al. [57]. Before starting the labeling process, we looked through a sample set of comments and found that the iTrust project mostly focuses on fulfilling Quality of Service (QoS) requirements and especially on two of the categories: *reliability* and *security*. Table 4.5 shows the statistics for the labeled dataset. Results show that less than 0.1% of the comments belong to other categories of NFRs in the dataset. Thus, we focused on these two categories during our study and assigned comments to each of these categories as follows:

- *Security:* A comment is labeled with security class if it concerns security, authentication, privacy, and user login of the system.
- *Reliability:* A comment is labeled with reliability if it concerns making a more robust and reliable system, such as form and data validation, verification and also handling exceptions of the system.

As a result of this process, we provide binary labels for the dataset to determine whether the comment is related to NFR concepts or not. In other words, the binary label can have one of the two values for each comment:

- True: The comment relates to at least one of the NFR categories *reliability* and *security*.

Table 4.6: Examples of comments and their corresponding category in the dataset

Category	Comment
NFR	`/*The patient ID is validated by the superclass*/`

not-NFR	`/*Returns a list of all HealthRecords for the given`
	`patient*/`

NFR	`/***`
	`Generate a new more secure hashed and randomly salted`
	`password based on the users`
	`* new desired password passed in as a String.`
	`* @param newpas String, desired new plain text password`
	`* @return`
	`private String genPassword(String newpas){`
	`String pas = "";`
	`SecureRandom rand = new SecureRandom();`
	`byte newbie[] = new byte[32];`
	`return pas;`
	`}`
	`*/`

not-NFR	`/***`
	`*`
	`* Used for chronicDiseaseRisks.jsp.  Passes most of the`
	`logic off to @link ChronicDiseaseMediator, and the`
	`* various subclasses of @link RiskChecker.`
	`*`
	`*/`

- False: The comment is not related to either of *security* or *reliability* categories.

Table 4.6 shows some examples of comments and the corresponding label we picked for them.

4.2 Preprocessing of comments

We preprocessed the data to tokenize comments into their constituent words. The tokenization process was done using NLTK, Natural Language Toolkit, a library written in Python that takes into account white spaces as a delimiter of its tokenizer [6]. We also implemented a customized tokenizer for our problem to tokenize words written in camel case, which is commonly used for the names of methods and variables in Java. Moreover, we removed stop words from each comment and used different techniques of stemming and lemmatization provided by NLTK tool to remove noise from the data.

After creating the dataset of comments, we immediately noticed that many of them clearly did not pertain to NFRs; these included comments about copyright licensing and authorship. We sought to discard these comments as irrelevant before proceeding to the automated classification stage; in so doing, we reduced the number of noisy comments and likely improved the accuracy of the classifiers. Since these irrelevant comments were easy to detect, we felt that any real-world implementation of our ideas would also likely filter them out. We used a comment classifier introduced by Steidl et al. [59] to classify comments into categories according to its natural language semantics by using machine learning. Table 4.7 shows the categories of the comment and description for each category. The approach suggested by the authors of the paper was reported to have more than 0.87 in F1 measure accuracy on each of the classes in a dataset of comments from twelve open source projects written in Java and C++. Similar to the original work, we defined seven categories and modeled the problem as a multi-class text classification problem in that the goal is to classify each comment as belonging to one of the categories listed in table 4.7. Figure 4.1 shows example code with comments illustrating each category[3]. Each comment is thereby annotated with a highlighted preceding note, stating the corresponding category.

We believe that comments related to NFRs are likely to be found only in "Inline", "Member", or "Task" comments, and that comments from the other categories can be considered unlikely to related to NFRs. Specifically, comments in the "Copyright" and

[3] The original code was taken from the book "Java in a Nutshell" by David Flanagan but modified and extended with more comments by Daniela Steidl.

```
Copyright
// This example is from the book -Java in a Nutshell- by David
// Flanagan. Written by David Flanagan. Copyright (c) 1996
// O' Reilly & Associates. For Demonstration purposes, source
// code was modified and comments were added by Daniela Steidl
import java.applet.*;
...
Header
/**
* This applet displays an animaton. It doesn't handle errors
* while loading images.
**/
public class Animator extends Applet implements Runnable {
Interface
/* the current image */
protected int current_image;
Section
/*******************************************
* Methods to start and stop the applet         *
*******************************************/
Interface
// Read the basename and num_images parameters.
// Then read in the images, using the specified base name.
public void init() {...}
public void stop() {
Code
// animator_thread.sleep();
if((animator_thread != null) && animator_thread.isAlive())
animator_thread.stop();
Inline
// We do this so the garbage collector can reclaim the Thread
// object. Otherwise it might sit around in the Web browser
// for a long time.
animator_thread = null;
}
public void run() {
Task
//@TODO: add functionality to launch animator in separate window
while(true) {     ...     }
}
}
```

Figure 4.1: Examples for each comment category in source code

Table 4.7: Different categories of source code comments according to their content and description of each category.

Comment Category	Description
Copyright comments	Include information about the copyright and license of the source code.
Header comments	Give an overview of the functionality of the class and provide information about, e.g., the class author and the revision number.
Member comments	Describe the functionality of a method or field
Inline comments	Describe implementation decisions within a method body
Section comments	Address a group of methods and variables that are similar in functionality, e.g., getters and setters
Code comments	Contain a piece of commented out code
Task comments	Contain note for developers and related to their future tasks or current issue.

“Section” classes cannot be about NFRs according to their description. Also, comments describing a code section can also be ignored, since our ultimate goal is to find NFRs from unstructured text of comments and commented codes cannot be helpful in our case. Furthermore, according to the definition of comment classes and also by investigating samples of source code comments, “Header” comments often contain more general information about large blocks of code — such as classes — and rarely has information about NFRs. Thus, by designing an accurate classifier, we decided to discard comments in the aforementioned classes and process only comments in one of “Inline”, “Member”, and “Task” categories.

We modeled this task as a classification problem and replicated the method used by Steidl et al. [59] with the following settings:

- **Training set:** In order to provide the data for training and also evaluation, we selected a set of 700 comments from Apache httpd web server (version 2.4.18) and manually labeled them with one of the comment categories described above[4].
- **Classification method:** The classification is done using Weka library. Steidl et al.

[4] The data was a part of a dataset containing 3667 .c files from source code of Apache httpd web server, with totally 16163 comments. The original dataset is collected by Ian J. Davis for a line of research that we chose not to explore further.

Table 4.8: The features extracted for each comment to use it as classification features in preprocessing of comments.

Feature	Description
isCopyright	Boolean feature that is true if comment contains the words “copyright” or “license”.
braceCount	Indicates how many braces are open at the position of the comment.
isFrame	Boolean feature that is true if comment contains a separator string multiple times (e.g., ***, - - -, ///)
length	Shows the number of words, separated by white spaces, in the comment.
hasTaskTag	Boolean feature that is true if comment is tagged with “task”, “todo”, “fixme”, or “hack”
followed	Boolean feature that indicates whether the comment is directly followed by another comment
specialCharacters	Indicates the percentage of special characters in a comment (e.g., ;, =, (,))
containsSnippet	Boolean feature that is true if comment contains code snippet.
insideMethod	Boolean feature that is true if the comment is within a method definition

found that optimal results were obtained by using the J48 tree as the classification method. We have also tried SVM and REP tree.

- **Classification features:** We have extracted the set of features that was used in the original work by Steidl et al. [59] from each single comment to use it as classification features. Table 4.8 lists the extracted features and their description.

The best accuracy was obtained by training a J48 classifier using 10-fold cross validation on the set of 700 training data. The trained model had an F1 measure of 80.832% with the above-mentioned setting.

4.3 Topic analysis methodology

The problem of detecting NFRs that occur in text can be considered as an unsupervised learning problem in which the goal is to identify common patterns between all of the sentences in certain NFR categories. If this kind of pattern exists, it can be used for unseen data to decide whether it contains discussion of an NFR or not. We believe that each NFR can determine a distribution over words, and thus a comment that is about a certain NFR will contain certain related words with higher probability. To capture this distribution for each requirement, we built our classifier based on topic modeling, as suggested by Hindle et al. [26]. We believe that there is likely a relation between generated topics and NFRs. So, the method performs the classification in two phases: finding topics and finding labels for each topic.

- Finding topics: We used LDA provided by *Gensim*, a Python library that provides tools to infer semantic information from text [50]. The input to this method is the dataset of comments plus the number of user-defined topics. The output is the model that shows topic distribution per document and word distribution for each topic.

- Finding labels for each topic: To find labels associated with each topic we used a wordlist provided by Hindle et al. [26]. In their study, they used three wordlists and compared the results of them. In this project, we just used one of these datasets that performed better than the others in the original study. Table 4.9 shows the list of top words in the wordlist for each of the *security* and *reliability* category. The goal of this part is to assign a label for each topic, if there is any correlation between the that topic and any of the categories of NFRs. The label for each topic is determined by using the intersection of the wordlist and the top words in a topic. We chose set of 50 common words for each topic and then compared it to the words in each wordlist. If there were at least one common words between them, we labeled that topic with the corresponding NFR category in the wordlist. With this assumption, a topic can be labeled with more than one NFR category or even no categories (in case that the lists have no common words).

The results of using topic modeling for extracting NFRs from the source code comments are discussed in detail in section 5.2.1.

4.4 Text classification methodology

The problem of extracting NFRs from text can be modeled as a text classification problem in which the goal is to find out whether a comment contains any of the categories of NFR or not. The precise model for this classification problem is:

Table 4.9: List of top words in the wordlist for each of the *security* and *reliability* category.

Category	Related Words
Reliability	reliability, dependability, dependableness, reliableness, accountability, answerability, answerableness, fault, reliable, authentic, dependable, honest, failure, error, redundancy, fails, bug, crash, stable, stability, integrity, resilience, responsibility, responsibleness, maturity, recoverability, fault tolerance
Security	security, protection, certificate, security department, security measure, security system, amount, payroll, resistance, risklessness, personnel, return, share, law, bill, expenditure, loss, capital, antenna, resource, authorization, license, plug, permit

- **Document:** Each comment is assumed to be a single document. A single comment can consist of several sentences, the name of variables, a piece of commented out code or description of the functionality of a class, variable, or method.
- **Classes:** Each comment can be either about NFRs or not. Thus, the decision is binary in which a comment belongs to "NFR" category if it belongs to at least one of the *security* and *reliability* classes. If the comment is neither about *security* nor *reliability*, we classify it in "not-NFR" class.
- **Classification model:** In principle, any binary classifier can be used with this problem; we chose to use Support Vector Machine (SVM) and Naïve Bayes as they are simple and commonly use.

4.4.1 Feature extraction

The original features we used to convert each comment to a vector was the original bag-of-words suggested by Slankas et al. [57]. By looking through a small sample of comments, we found out that some of the features in the comments can help in deciding whether a comment is about the NFR or not. For example, the comments describing the "return type" or "return value" of the functions are usually very short and just name one or more variables of the function. Thus, they usually describe a functionality of the method and does not concern about NFRs of the software.

By capturing similar kinds of patterns in the comments, we extracted the following six attributes for each comment and used them as classification features in addition to the bag-of-words features suggested by Slankas et al. [57].

- *isCRUD:* A Boolean attribute that is true if the comment is detected as related to one of the CRUD (Create, Read, Update, Delete) operations on data and false otherwise. So, the attribute is true if the comment contains at least one of the words "create", "read", "update", "remove", and "delete"
- *hasReturn:* A Boolean attribute that is true if the comment contains the word "return" and is false otherwise
- *isImperative:* A Boolean attribute that checks whether the comment contains imperative words or not. The attribute is true if the comment contains at least one of the words "should", "must", "ought to", "have to", "has to", "need", "remember", "make sure", and "be sure" and is false otherwise.
- *length:* A numerical attribute that shows the length of the comment in number of characters

Although bag-of-words features are known to have a high performance in text classification approaches, they have two major drawbacks [34]: (1) they lose ordering of the words and (2) they ignore the semantics of the words, and the concepts that are similar to each other. Thus, to capture the semantics of text, we also used *doc2vec*, a model based on neural networks, to convert each document to a fixed length vector in an unsupervised manner. The model was learned using Gensim package using the dataset of comment with ten epochs and parameters "α" = 0.025 and "min-α" = 0.025. The effect of using different sets of features on the performance of classifiers is discussed in chapter 5.3.

4.4.2 Classification models

To perform the classifications, we converted each comment into its corresponding feature vector. The size of feature vectors depends on the feature extraction approach we use, however for each approach the length of feature is fixed for all of the documents. For classification, we used Gaussian Naïve Bayes and also SVM with polynomial kernel in which we evaluated the model using 10-fold cross-validation on our dataset of comments. We discuss the results of using text classifiers in detail in section 5.2.2.

4.5 Summary

In this chapter, we proposed a framework to extract NFRs from the comments available in the source code of software project in the EHR domain. To this end, we modeled the problem as a text classification problem and classified each comment to determine whether it is related to any categories of NFR or not. For this project, we only focused on *security* and *reliability* categories that are the most common quality attributes of software that are mentioned in the source code. As the first step of the process, we created a labeled dataset of comments to have a ground truth to work with. Next, we preprocessed data to filter out some of the irrelevant comments and remove noise from the data. We suggested two approaches to solve the problem, the first one is based on topic patterns in the comments and the second one is based on supervised text classification models. The details of the experiments and result on our methodology is explained in chapter 5

Chapter 5

Research Results and Discussion

Throughout this chapter, we present the results of running the experiments described in chapter 4 on the dataset of software documents and comments in the EHR domain. As suggested in chapter 4, we analyzed the topics available in the comments to find common patterns of using non-functional categories in the comments, with the idea of relating the topics to categories of NFRs. We have also modeled the problem as a classic text classification problem and performed the experiment on the available data. This chapter presents the performance of suggested models for our problem and compares the results under different settings.

In this project, we address the following research questions:

RQ1: *How often are NFRs mentioned explicitly or implicitly in source code comments?*

RQ2: *How effectively can NFRs be extracted from source code comments?*

RQ3: *Which sentence characteristics are the most useful for extracting NFRs from source code comments?*

RQ4: *What are the differences and similarities between extracting NFRs from source code comments and other software documents?*

This chapter provides the answer to these research questions along with the results of our studies.

Table 5.1: Statistics of the dataset of comments; number of comments related to each category and percentage of the category in the dataset

Category	Number	Percentage
Reliability	1316	17.9
Security	73	1.0
Other NFRs	3	0.04
Total	1392	18.9

5.1 Breakdown of NFRs in the source code

RQ1: *How often are NFRs mentioned explicitly or implicitly in source code comments?*

Based on the labels we manually provided for the dataset of comments in the EHR domain, we found that 1392 comments out of 7337 total comments explicitly or implicitly relate to non-functional properties of the system (18.9% of the total which is a noticeable portion of the comments). Table 5.1 shows the breakdown of each category of NFRs among comments.

As it is indicated in table 5.1, we found that 1389 of the comments belong to one of the categories of *reliability* or *security* (18.9 % of comments) which is 99.7% of the total comments that are related to the NFRs. Having only 0.04% of the comments about other categories, convinced us to discard those categories in the training and evaluation process and focus on the two most common categories: *reliability* and *security.*

From the comments that are related to *security*, we found that only 26% (19 comments) explicitly mentioned the words "security" or "secure", while the other 74% implied this requirement. However, the results for *reliability* category were somewhat surprising since none of the comments in this category explicitly mentioned words "reliability" or "reliable" in the comments and they mostly talked about other concepts like "validation" that implies the reliability of the system.

Although the breakdown of categories of NFRs mentioned in the comments can be different for other domains and other projects, the results for the iTrust project shows that in that project, most of the developers are concerned about Quality of Service (QoS) type of requirements during the implementation phase of the project. This category can contain any of the requirements related to safety, security, reliability, performance, interface, and accuracy [65].

5.2 Performance of Machine learning techniques

RQ2: *How effectively can NFRs be extracted from source code comments?*

As described in chapter 4, we modeled the problem as a classification problem of extracting NFRs from text and proposed two approaches to deal with the problem. The first approach used topic modeling to verify the existence of NFRs in text of comments and also infer the common patterns for each requirement if such patterns exist. The other approach uses supervised text classification to decide whether the comment contains any NFRs or not. The rest of this section presents our results from using these two approaches.

5.2.1 Topic modeling

We believe that if there exist comments about NFRs in the source code, each NFR category could be considered as a "topic" that developers are talking about. In other words, each NFR category can define a distribution of words according to its type, and in this case, the problem can be modeled as a topic modeling problem in which the goal is to identify latent topics in text according to the word distribution for each topic.

Our method for finding NFRs from the topic mentioned in text works in two phases:

- Finding topics: We used LDA implementation in the Gensim package to find the topics for the corpus of comments. The input for this phase is the text of dataset in which each comment is considered as a separate document and the number of topics that is user-defined. For our study, we used 50 as the number of topics, since we believe it showed better word distribution according to the actual topics in text. The output of this phase is a model that shows the word probability for each individual topic and also topic probabilities for each comment.

- Assigning label for each topic: We used the labeling process suggested by Hindle et al. [26]. To this end, we selected the N top words (the words with the highest probability) for each topic as representatives of that topic. The top words are intersected with a pre-defined list of words for *security* and *reliability* NFRs to determine whether the topic is about any of these categories or not. If there exists at least one common word, the topic is labeled with that NFR tag.

Table 5.2 shows the effect of the number of topics on the performance of our approach. We tried a range of topics numbers, between 10 and 60. At the low end, we found that

Table 5.2: Effect of number of topics on the performance of LDA-based topic labeling

Number of topics	Precision	Recall	F-measure
10	0.209	0.849	0.336
20	0.203	0.654	0.310
35	0.193	0.447	0.269
50	0.210	0.561	0.305
60	0.157	0.339	0.215

with a smaller number of topics, the model appears to be accurate in terms of precision, recall, and F1 measure. However, in those models, the extracted topics are too general and it can be hard to decide on a specific, concrete unifying theme. On the other hand, at the high end with more than 50 topics, we found that the topics become much narrower, and cannot capture statistical co-occurrence behavior of the comments. In the end and using our judgment, we found that 50 topics provided the best trade-off between high performance of the proposed model and level of semantic granularity.

Table 5.3 shows the effect of number of selected word for each topic on performance of our proposed approach. For the experiments in this part, the number of extracted topics is 50 for our corpus. The results shows that when the number of selected words are very limited (for example 5 or 10 words per topic), the model does not perform well. In these cases, the labels assigned to each topic is decided based on a small set of words, that might not represent the distribution of words in the topic very well. Thus, when the number of selected top words are low (fewer than 20), both the precision and recall of the model are relatively low. On the other side, in cases where the number of selected words for each topic is high (more than 35), the performance drops. For these cases, the words with lower conditional topic-term probability are also selected as a representative of the topics and are involved in deciding whether the topic is about NFRs or not. In these models, the recall is high compared to other settings. However, low precision is likely to frustrate users in a real-world situation. For the number of words between 20 and 35, the classifier has the highest F1 score (with 0.305 F1 score in best case) and the top words can be accurate representative of the distribution of words within the topic.

5.2.2 Text classification

We modeled the problem as a classification problem in which the goal is to find whether given code comment concerns an NFR or not. To perform the classification, we used two

Table 5.3: Effect of number of top selected words for each of 50 topics on the performance of LDA

Number of words	Precision	Recall	F-measure
5	0.153	0.195	0.172
10	0.157	0.296	0.195
20	0.210	0.561	0.305
35	0.204	0.606	0.305
50	0.183	0.730	0.293

approaches — Gaussian Naïve Bayes and Support Vector Machine (SVM) with Polynomial kernel — from *Scikit-Learn*, a toolkit for machine learning and data analysis written in Python [48]. We chose these two classification approaches over other classification approaches, since they are known to perform well on text data. Moreover, both approaches were used in the original work by Slankas et al. [57], which enables us to compare our approach to theirs in a similar setting.

For the experiments in this part, we used our extended feature set explained in 4.4.1. The features include the words in original form, length of the comment, whether comment sentence is imperative or not, whether the comment is about CRUD operations or not, and whether it contains "return" statement.

Table 5.4 presents the precision, recall, and F1 measure for the average for five runs of each binary classification algorithm in which the goal is to determine whether the comments are related to NFR or not. For each experiment, the classifier is trained on the dataset of comments using 10-fold cross-validation. We also repeated the experiment for each of the *security* and *reliability* categories separately and presented the results in tables 5.5 and 5.6 respectively.

According to the results for all of the cases, SVM classifier had an F1 score of 0.860 and performed better compared to Naïve Bayes classifier with an F1 score of 0.741. This was unsurprising since SVM models usually work well with large dimensional problems with relatively few instances because of being well-regularized that makes it tolerate misclassification and generalize to new data points [55]. This property of SVM justifies the huge gap between the results in table 5.5, which is the case of determining whether a comment is about *security* or not. The number of sentences with *security* label is very low (only 2.2% of the comments) that makes Naïve Bayes perform poor in this case. However, the gap is less for *reliability* category since it contains relatively more comments in the dataset.

Table 5.4: Performance of classifiers to determine whether the comment is "NFR" or "non-NFR". The method is evaluated using 10-fold cross-validation on the dataset of comments.

Classifier	Precision	Recall	F-measure
Naïve Bayes	0.765	0.718	0.741
SVM	0.875	0.845	0.860

Table 5.5: Performance of classifiers to determine whether the comment is about *security* or not. The method is evaluated using 10-fold cross-validation on the dataset of comments.

Classifier	Precision	Recall	F-measure
Naïve Bayes	0.634	0.261	0.369
SVM	0.870	0.643	0.739

5.3 Performance of different sets of classification features

RQ3: *Which sentence characteristics are the most useful for extracting NFRs from source code comments?*

Classifiers can use any number of features to decide whether a comment belongs to any of the NFR categories or not. In this study, we examined different sets of features to evaluate the performance of classifiers and determine the best set of features for our problem. The feature sets that we used is listed as follows.

1. The first feature set is the same as the features used by Slankas et al. [57]. This feature set includes the words in the original form and a modified version of dependency tree defined in the original work, that is called Sentence Representation (SR).

2. The second feature set is an extended version of feature set #1. Thus, in addition to the original form of the words, it takes into account factors such as length

Table 5.6: Performance of classifiers to determine whether the comment is about *reliability* or not. The method is evaluated using 10-fold cross-validation on the dataset of comments.

Classifier	Precision	Recall	F-measure
Naïve Bayes	0.767	0.784	0.775
SVM	0.876	0.876	0.876

Table 5.7: Performance of difference set of features with Naïve Bayes and SVM classifiers. The method is evaluated using 10-fold cross-validation on the dataset of comments.

	Naïve Bayes			SVM		
Feature set	Precision	Recall	F1	Precision	Recall	F1
Feature set #1	0.711	0.755	0.733	0.872	0.838	0.854
Feature set #2	0.765	0.718	0.741	0.875	0.845	0.860
Feature set #3	0.269	0.316	0.290	0.580	0.583	0.581

of the comment, whether comment is imperative or not, whether the comment is about CRUD operations or not, and whether it contains "return" statement. These extended features are described in more details in section 4.4.1.

3. The third set of features is the vector representation of each sentence in multi-dimensional space. The vectors are created using *doc2vec* that is an extended version of *word2vec*, and provides continuous representation for larger blocks of text such as one or more sentences in a comment.

Table 5.7 shows the results of each set of features evaluated using 10-fold cross-validation on each of the two classification models used throughout this study. The results show that second set of features has the best performance compared to other features (F1 score of 0.860 with SVM classifier and 0.741 with Naïve Bayes classifier). Since *doc2vec* is a statistical model and needs a very large amount of data to be trained on, it is highly unlikely to perform as well as other approaches when the amount of data is limited. In those cases, using words as they appear in the sentence and classic feature extraction approaches will have better performance. To validate this claim, we repeated the experiment, but this time instead of learning the feature vectors using merely the dataset of comments, we used the dataset of software documents to train our model. The learned model is evaluated on the dataset of comments. The results in table 5.8 show that in this case, the *doc2vec* has the highest performance compared to other feature settings. Although the gap between F1 scores is not large, this suggests that *doc2vec* method can perform better in cases that there is enough data to learn semantics from.

For the feature sets #1 and #2, we used the exact feature setting suggested in the original work [57]. As the authors suggested, the best performance was achieved using original word form and also by filtering out determiners as stop words. Thus we used the same feature setting for our experiments and replicated the work for our dataset of comments. For feature set #3, we examined the effect of the size of vectors on the performance of *doc2vec* models. Table 5.9 presents the results of using different vector size in each of

Table 5.8: Performance of difference set of features with Naïve Bayes and SVM classifiers. The classifers are trained on the dataset of documents and evaluated on the comments

	Naïve Bayes			SVM		
Feature set	Precision	Recall	F1	Precision	Recall	F1
Feature set #1	0.07	0.40	0.13	0.07	0.10	0.08
Feature set #2	0.031	0.059	0.039	0.031	0.052	0.039
Feature set #3	0.220	0.094	0.132	0.246	0.098	0.140

Table 5.9: Effect of vector size on performance of classifiers for NFR detection. The method is evaluated using 10-fold cross-validation on the dataset of comments with *doc2vec* feature vectors.

	Naïve Bayes			SVM		
Vector size	Precision	Recall	F1	Precision	Recall	F1
50	0.255	0.244	0.249	0.523	0.548	0.535
100	0.271	0.278	0.274	0.563	0.560	0.561
200	0.263	0.294	0.277	0.560	0.552	0.555
300	0.260	0.295	0.276	0.561	0.569	0.564
400	0.269	0.316	0.290	0.580	0.583	0.581

the classification methods we used in our study. Vector size shows the dimensionality of feature vectors used in the classification. As the results suggest, all of the models perform better in high dimentionality, i.e., vector size bigger than 200. The F1 measure is almost the same for vector sizes 200, 300, and 400. However, since recall is a more critical score for our problem, we chose vector size of 400 for our experiments with *doc2vec* features.

Table 5.10 presents the results of using different word forms and stop words as the classification features. To compare the effect of different word forms and different sets of stop words to remove from data, we used SVM with polynomial kernel. The size of feature vectors is 400 for each of the comments, and the results are obtained by 10-fold cross-validation on the dataset of comments. The "Original" word form represents the words as they appear in the document. "Lemma" is the lemma of the original word, that was obtained using WordNet Lemmatizer in Natural Language Toolkit (NLTK) library. "Stemmed" form is the stem of the original word produced by Porter stemming algorithm [49]. Stopwords are a list of words that do not carry semantic significance due to the common appearance in text. These words can be filtered out from text before further processing. "Determiners" are "a", "an" and "the", and "English stopwords" is a list of 153 common terms in English provided by NLTK.

Table 5.10: Effect of text preprocessing on the performance of classifiers in NFR detection. The method is evaluated using 10-fold cross-validation on the dataset of comments with *doc2vec* feature vectors.

Word Form	Stop Words	Precision	Recalll	F-measure
Original	-	0.511	0.524	0.517
	Determiners	0.540	0.566	0.552
	English Stopwords	0.563	0.545	0.551
Lemma	-	0.527	0.564	0.541
	Determiners	0.538	0.559	0.545
	English Stopwords	0.553	0.552	0.552
Stemmed	-	0.527	0.530	0.528
	Determiners	0.532	0.541	0.535
	English Stopwords	0.580	0.583	0.581

As the results suggest, for the *doc2vec* model the best performance was achieved using a stemmed version of the word in processing and training the model. Lemmatization does not help much in enhancing the performance and f-measure is almost the same in cases of using original word and lemma of the word. For all of the classifiers, removing stop words improved the accuracy of the model, specially in case of using the stemmed version of the word that removing stop words improved the accuracy of the model from 0.528 in F1 score to 0.581. Thus, we used this setting in all of the results presented in this thesis.

5.4 Comparison

RQ4: *What are the differences and similarities between extracting NFRs from source code comments and other software documents?*

To compare the characteristics of comments and other software documents, we repeated the classification experiment under three different settings:

- **Experiment 1:** Learning the model on the sentences from the comments and evaluating on the same dataset using 10-fold cross-validation.
- **Experiment 2:** Learning the model on the sentences from dataset of software documents and evaluating on the same dataset using 10-fold cross-validation
- **Experiment 3:** Learning the model on the sentences from the dataset of software documents and evaluating on the sentences from comments.

Table 5.11: Performance of SVM classifier in different experiment settings

Experiment #	Precision	Recall	F-measure
Experiment 1	0.875	0.845	0.860
Experiment 2	0.743	0.693	0.717
Experiment 3	0.031	0.052	0.039

Table 5.12: Performance of Naïve Bayes classifier in different experiment settings

Experiment #	Precision	Recall	F-measure
Experiment 1	0.765	0.718	0.741
Experiment 2	0.596	0.747	0.663
Experiment 3	0.031	0.059	0.039

We evaluated each of the experiments with the two classifiers described at the beginning of this chapter. Tables 5.11 and 5.12 show the evaluation results using SVM and Naïve Bayes as the classification model respectively. As it is shown in these tables, in both of the models the highest performance is achieved by using the dataset of comments as both training and test set. Although characteristics of data might be slightly different for software documents (compared to the dataset of comments), we believe that higher accuracy in the first experiment is mostly because of the consistency of data in comments and choosing the data from one project.

Experiment 3 — where the classifier is trained on the dataset of software documents and evaluated on the dataset of comments — has the lowest accuracy among other experiments. Since the algorithm we are using is the same for all of the experiments, we can conclude that the difference in performance relates back to the difference in characteristics of our datasets, for example:

- The number of words in the sentences used in the comments is slightly fewer than the number of words in the sentences for other software artifacts. For the dataset that we have, the average length of a comment is 9.74 that indicates developers use shorter sentences compared to other documents (with average 18.45 in length). Thus, comment sentences on average probably convey less information and this will affect the performance of machine learning methods.

- Due to the nature of the code comments, developers mostly tend to talk about functional requirements rather than NFRs in the source code. Moreover, even among NFRs they mostly talk about certain categories and ignore other types. Since other

Table 5.13: Performance of our NFR extraction approach compared to the previous work.

Classifier	Precision	Recall	F-measure
Proposed topic modeling	0.210	0.561	0.305
Proposed classification	0.875	0.845	0.860
NFR-Locator	0.728	0.544	0.523

types of software artifacts have more data in each category and on average cover more types of NFRs, we believe that this will result in a difference in patterns of using NFRs in the two types of data.

- In supervised learning tasks, the performance of the models depends heavily on the labels provided for training and evaluation data. Since the labeling process and the set of labels are slightly different for the two datasets, inconsistency in labeling process might lead to differences in the results.

- Word distributions and category definitions are different for the two datasets. For example, the number of sentences with *reliability* label in the dataset of software artifacts is small — 43 sentences in total — that made the classifier for this category relatively inaccurate. By carefully looking through these 43 sentences, we found that most of the sentences labeled with *reliability* in our own dataset of comments are either about validation or verification of the system and data. In contrast, in the dataset of software artifacts, there were a few sentences about these two concepts. We believe that the difference in definition of the categories can lead to difference in the distribution of words used to describe a certain NFR category (e.g., *reliability*) in the two datasets.

Table 5.13 shows the results of our proposed approach compared to the others. Since our work is the first study in finding NFRs from source code comments, there is no previous method to compare our methods with it. Thus, we discuss the results for *NFR-Locator* [57] in the results, which was the inspiration for our study. However, even the results for *NFR-Locator* is obtained for classifying dataset of software documents in 14 NFR categories, that is different from our experiment in data and the number of categories.

As the results show, modeling the problem of extracting NFRs from the comments as supervised classification has better performance compared to the suggested unsupervised approach. The main reason for poor performance of topic modeling for our problem is that the number of comments is limited and also the comments are usually short (9.74 words on average). These two characteristics make finding patterns in text and relating them back to NFRs a problematic task for statistical models like topic modeling. Thus, we believe that supervised methods are more appropriate models for extracting knowledge from comments.

In general, our proposed approach performed better compared to the original model. Choosing data from one type of software artifact (comments in the source code) and limiting the task to identifying two categories of NFRs instead of 14 categories may have impacted the accuracy of our model. However, we believe that the main reason for the difference in performance relates back to the characteristics of the data in our method and *NFR-Locator*. As it was mentioned in section 4.2, a large proportion of the comments are clearly not about NFRs and can be filtered out in early stages of NFR detection to make the detection task easier.

5.5 Summary

In this chapter, we present the results of running our proposed methods for extracting NFRs from source code comments under different settings. The results show that modeling the problem of extracting NFRs from the comments as supervised classification has better performance compared to the proposed unsupervised approach. We believe that the difference can be due to the small size of available data and characteristics of the dataset of comments (e.g., shorter sentences). We have also evaluated the classifiers with multiple sets of features to identify NFRs from the comments. The results show that in our case, using bag-of-words with added customized features — that help in capturing semantic and structural patterns in the comments — outperforms the more complicated feature sets (e.g., *doc2vec* features). In the next chapter, we discuss the threats to the validity of the experiments that we have performed throughout this research and also the possible future directions to pursue this work.

Chapter 6

Conclusions

In this dissertation, we have presented a method to assist analysts in extracting relevant NFRs from comments available in the source code of software systems using automated NLP. To evaluate the approach, we collected a set of comments from iTrust, an open-source software in the EHR domain and demonstrated that developers are concerned about quality attributes of this software by finding source code comments that are related to NFRs.

We modeled the problem of extracting NFRs from source code as a classification problem in which the model decides whether a sentence contains any NFRs or not. According to the domain of the project and the requirement document available for the iTrust project, we focused on two common types of NFRs, *security* and *reliability*.

We evaluated multiple classifiers with multiple sets of features to identify NFRs from the comments. For the features, we found that for smaller data size, using bag-of-words with some customized features added to capture some patterns in the comments outperforms the other feature sets. However, for significantly larger datasets, the *doc2vec* method can be an excellent candidate to convert the text of each comment to a fixed-sized vector that encompasses the semantics in the comment.

As far as we are aware, this study can be considered as the first attempt to extract NFRs from source code comments. Thus by designing different experiments on comments and also other software documents, we found that although approaches to extract NFRs from comments are similar to extracting it from other kinds of documents, there are several characteristics inherent to comments that can impact the studies. Considering these differences can open future research directions for extracting comments from the source code and its comments.

6.1 Threats to validity

There are several threats that may affect the validity of our study. We have divided them into three main categories: Construct Validity, Internal Validity, and External Validity.

The main construct validity threat for this project is the bias in labeling process. All of the labels for training and evaluation purpose is provided by the author. Although the labels are reviewed several times to avoid possible mistakes, there is still a chance of missing requirements due to inherent ambiguity and complexity of natural language. Moreover, the person performed the labeling process for the dataset of comments was different from the one who provided labels for the dataset of software documents. This might introduce inconsistency between labels for some sentences.

One of the threats to internal validity of this project is the size of data. The dataset of comments that we used in our experiments is of size fewer than 8000 sentences which is considered small for statistical analysis of data. We tried to reduce the effect of lack of data by training the classifiers on a bigger size dataset of other software documents. However, the models that we used for this study are among statistical models, in which larger size of data will provide more information and enhance the performance of the models. Using statistical models like *doc2vec* for small dataset available for our study could invalidate some of our results.

For the experiments in this project, we focused on data from the EHR domain that provided us with some extent of labeled data and previous studies to compare to our results. Although the approaches we used in this study are independent of the domain and can be easily used for other domains, our results should not be generalized for other domains. For example, it is possible that developers have more concerns about NFRs in the EHR domain compared to other domains. Using data from one domain can be an external threat to the validity of the results.

We have used a variety of software documents and software projects within the EHR domain. However, we only had access to the source code of one of those projects — the iTrust project — to create our dataset of comments. We believe that the accuracy of the methods can be directly dependent on how well the code is commented by developers. To mitigate this risk, we investigated a subset of comments from the iTrust project before any further processing, and we made sure that it is a well-documented project and the comments adequately reflect the code, which might not be the case for all of the projects even within the EHR domain.

6.2 Future work

The work performed in this project can be considered as the first attempt to extract NFRs from comments on the source code. Thus, it can open several research directions to be pursued in future works. Some of the possible future directions are discussed below.

In this study, we focused on the EHR domain for training and evaluating our work. One possible direction for future studies is to expand the domain both within the EHR and also into other domains. By performing the same analysis on the comments from other domains, we can examine the generalizability of our approach and also investigate similarities and differences of comments in various areas.

The major roadblock for this study was limited access to data to train our model. This issue constrained the models that we could use for the project and also might have affected the results. Neural network-based approach such as *doc2vec* usually have the best performance when trained on a large set of data. By training the model on a large data, we would be able to capture the semantic information hidden in the comment sentences as much as we can and train more accurate models.

The idea behind extracting NFRs from comments available in the source code was to track the focus of developers on implementing this kind of requirements. To this end, we used a well-documented codebase in which the comments reflect the description of functionalities along with the purpose of implementing various pieces of code. In this study, our goal was to extract NFRs from text of comments. The next step for finding NFRs from source code is to use the requirements found by our approach and trace it to their implementation in the source code.

The main drawback of the approach proposed in this study is the need for the labeled dataset to train the models in a supervised manner. Although there is a very large amount of data from comments in open source projects in both EHR and other domains, none of them has the labels to indicate whether they are about any of the categories of NFRs or not. Providing labels for each sentence is a costly process and might be inaccurate due to human bias. Thus, using unsupervised or semi-supervised approaches with comparable performance and less need for labeled data can be one of the possible directions for this study.

References

[1] Nida Afreen, Asma Khatoon, and Mohd Sadiq. A taxonomy of softwares non-functional requirements. In *Proceedings of the Second International Conference on Computer and Communication Technologies*, pages 47–53. Springer, 2016.

[2] R Arun, Venkatasubramaniyan Suresh, CE Veni Madhavan, and MN Narasimha Murthy. On finding the natural number of topics with latent dirichlet allocation: Some observations. In *Advances in Knowledge Discovery and Data Mining*, pages 391–402. Springer, 2010.

[3] Vikas Bajpai and Ravi Prakash Gorthi. On non-functional requirements: A survey. In *Electrical, Electronics and Computer Science (SCEECS), 2012 IEEE Students' Conference on*, pages 1–4. IEEE, 2012.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[5] Christian Bird, Tim Menzies, and Thomas Zimmermann. *The Art and Science of Analyzing Software Data*. Elsevier, 2015.

[6] Steven Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 69–72. Association for Computational Linguistics, 2006.

[7] Gary Blake and Robert W Bly. *The elements of technical writing*. Macmillan New York, NY, 1993.

[8] David M Blei. Probabilistic topic models. *Communications of the ACM*, 55(4):77–84, 2012.

[9] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[10] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE software*, 1(1):75, 1984.

[11] Michael Buckland and Fredric Gey. The relationship between recall and precision. *Journal of the American society for information science*, 45(1):12, 1994.

[12] Agustin Casamayor, Daniela Godoy, and Marcelo Campo. Identification of non-functional requirements in textual specifications: A semi-supervised learning approach. *Information and Software Technology*, 52(4):436–445, 2010.

[13] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542, 2009.

[14] D Manning Christopher, Raghavan Prabhakar, and SCHÜTZE Hinrich. Introduction to information retrieval. *An Introduction To Information Retrieval*, 151:177, 2008.

[15] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[16] Rudi L Cilibrasi and Paul MB Vitanyi. The google similarity distance. *IEEE Transactions on knowledge and data engineering*, 19(3), 2007.

[17] Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. Automated classification of non-functional requirements. *Requirements Engineering*, 12(2):103–120, 2007.

[18] Alistair Cockburn. *Agile software development*, volume 177. Addison-Wesley Boston, 2002.

[19] International Organization For Standardization/International Electrotechnical Commission et al. Software engineering–product quality–part 1: Quality model. *ISO/IEC*, 9126:2001, 2001.

[20] IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board. IEEE recommended practice for software requirements specifications. Institute of Electrical and Electronics Engineers, 1998.

[21] Darren Dalcher. Disaster in london. the las case study. In *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS'99. IEEE Conference and Workshop on*, pages 41–52. IEEE, 1999.

[22] Marie-Catherine De Marneffe, Bill MacCartney, Christopher D Manning, et al. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, volume 6, pages 449–454. Genoa Italy, 2006.

[23] Neil A Ernst and John Mylopoulos. On the perception of software quality requirements during the project lifecycle. In *Requirements Engineering: Foundation for Software Quality*, pages 143–157. Springer, 2010.

[24] Anthony Finkelstein and John Dowell. A comedy of errors: the london ambulance service case study. In *Proceedings of the 8th International Workshop on Software Specification and Design*, page 2. IEEE Computer Society, 1996.

[25] M Mahmudul Hasan, Pericles Loucopoulos, and Mara Nikolaidou. Classification and qualitative analysis of non-functional requirements approaches. In *Enterprise, Business-Process and Information Systems Modeling*, pages 348–362. Springer, 2014.

[26] Abram Hindle, Neil A Ernst, Michael W Godfrey, Richard C Holt, and John Mylopoulos. Automated topic naming to support analysis of software maintenance activities. In *The 33rd International Conference on Software Engineering, ICSE*, 2011.

[27] Abram Hindle, Michael W Godfrey, and Richard C Holt. What's hot and what's not: Windowed developer topic analysis. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 339–348. IEEE, 2009.

[28] Thomas Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 50–57. ACM, 1999.

[29] Ivar Jacobson, Grady Booch, James Rumbaugh, James Rumbaugh, and Grady Booch. *The unified software development process*, volume 1. Addison-wesley Reading, 1999.

[30] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. *Machine learning: ECML-98*, pages 137–142, 1998.

[31] James Joyce. Bayes' theorem. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2016 edition, 2016.

[32] Hyo Taeg Jung and Gil-Haeng Lee. A systematic software development process for non-functional requirements. In *Information and Communication Technology Convergence (ICTC), 2010 International Conference on*, pages 431–436. IEEE, 2010.

[33] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques.* Wiley Publishing, 1998.

[34] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196, 2014.

[35] Pericles Loucopoulos and Vassilios Karakostas. *System requirements engineering.* McGraw-Hill, Inc., 1995.

[36] Pericles Loucopoulos, Jie Sun, Liping Zhao, and Farideh Heidari. A systematic classification and analysis of NFRs. 2013.

[37] K Mahalakshmi and R Prabhakar. Hybrid optimization of svm for improved non-functional requirements classification. *International Journal of Applied Engineering Research*, 10(20):2015, 2015.

[38] Anas Mahmoud. An information theoretic approach for extracting and tracing non-functional requirements. In *Requirements Engineering Conference (RE), 2015 IEEE 23rd International*, pages 36–45. IEEE, 2015.

[39] Richard R Maiti and Frank J Mitropoulos. Capturing, eliciting, predicting and prioritizing (cepp) non-functional requirements metadata during the early stages of agile software development. In *SoutheastCon 2015*, pages 1–8. IEEE, 2015.

[40] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Madison, WI, 1998.

[41] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The promise repository of empirical software engineering data, 2012.

[42] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[43] Tomáš Mikolov, Anoop Deoras, Stefan Kombrink, Lukáš Burget, and Jan Černockỳ. Empirical evaluation and combination of advanced language modeling techniques. In

Twelfth Annual Conference of the International Speech Communication Association, 2011.

[44] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[45] George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[46] Kamal Nigam, Andrew Kachites McCallum, Sebastian Thrun, and Tom Mitchell. Text classification from labeled and unlabeled documents using em. *Machine learning*, 39(2):103–134, 2000.

[47] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*, pages 331–341. IEEE Computer Society, 2009.

[48] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[49] Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[50] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[51] Maria Riaz, Jason King, John Slankas, and Laurie Williams. Hidden in plain sight: Automatically identifying security requirements from natural language artifacts. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 183–192. IEEE, 2014.

[52] Winston W Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, 1987.

[53] David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations.* MIT Press, Cambridge, MA, USA, 1986.

[54] Vibhu Saujanya Sharma, Roshni R Ramnani, and Shubhashis Sengupta. A framework for identifying and analyzing non-functional requirements from text. In *Proceedings of the 4th International Workshop on Twin Peaks of Requirements and Architecture*, pages 1–8. ACM, 2014.

[55] SN Sivanandam and SN Deepa. *Introduction to neural networks using Matlab 6.0.* Tata McGraw-Hill Education, 2006.

[56] John Slankas and Laurie Williams. Access control policy extraction from unconstrained natural language text. In *Social Computing (SocialCom), 2013 International Conference on*, pages 435–440. IEEE, 2013.

[57] John Slankas and Laurie Williams. Automated extraction of non-functional requirements in available documentation. In *Natural Language Analysis in Software Engineering (NaturaLiSE), 2013 1st International Workshop on*, pages 9–16. IEEE, 2013.

[58] John Slankas, Xusheng Xiao, Laurie Williams, and Tao Xie. Relation extraction for inferring access control rules from natural language artifacts. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 366–375. ACM, 2014.

[59] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 83–92. IEEE, 2013.

[60] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[61] Lin Tan. Chapter 17 - code comment analysis for improving software quality*. In Christian Bird, Tim Menzies, and Thomas Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 493 – 517. Morgan Kaufmann, Boston, 2015.

[62] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.

[63] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 11–20. IEEE, 2011.

[64] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. @ tComment: Testing javadoc comments to detect comment-code inconsistencies. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 260–269. IEEE, 2012.

[65] Axel Van Lamsweerde. *Requirements engineering: From system goals to UML models to software*, volume 10. Chichester, UK: John Wiley & Sons, 2009.

[66] Laurie Williams, Tao Xie, Andy Meneely, Lauren Hayward, and A Massey. itrust medical care requirements specification. *Versions of September 3rd 2010*, 2008.

[67] Bin Yin and Zhi Jin. Extending the problem frames approach for capturing non-functional requirements. In *Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on*, pages 432–437. IEEE, 2012.

[68] Annie TT Ying, James L Wright, and Steven Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *ACM SIGSOFT software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[69] ChengXiang Zhai. Statistical language models for information retrieval. *Synthesis Lectures on Human Language Technologies*, 1(1):1–141, 2008.

[70] Wen Zhang, Ye Yang, Qing Wang, and Fengdi Shu. An empirical study on classification of non-functional requirements. In *The Twenty-Third International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, pages 190–195, 2011.

[71] Jie Zou, Ling Xu, Weikang Guo, Meng Yan, Dan Yang, and Xiaohong Zhang. Which non-functional requirements do developers focus on? an empirical study on stack overflow using topic analysis. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 446–449. IEEE, 2015.