USING MACHINE LEARNING AND GRAPH MINING APPROACHES TO IMPROVE SOFTWARE REQUIREMENTS QUALITY: AN EMPIRICAL INVESTIGATION

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Maninder Singh

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Computer Science

May 2019

Fargo, North Dakota

North Dakota State University

Graduate School

Title

USING MACHINE LEARNING AND GRAPH MINING APPROACHES TO IMPROVE SOFTWARE REQUIREMENTS QUALITY: AN EMPIRICAL INVESTIGATION

By

Maninder Singh

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Gursimran Singh Walia

Chair

Dr. Saeed Salem

Dr. Juan Li

Dr. Limin Zhang

Approved:

June 3, 2019 — Date

Dr. Kendall Nygard — Department Chair

ABSTRACT

Software development is prone to software faults due to the involvement of multiple stakeholders especially during the fuzzy phases (requirements and design). Software inspections are commonly used in industry to detect and fix problems in requirements and design artifacts, thereby mitigating the fault propagation to later phases where the same faults are harder to find and fix. The output of an inspection process is list of faults that are present in software requirements specification document (SRS). The artifact author must manually read through the reviews and differentiate between true-faults and false-positives before fixing the faults. The first goal of this research is to automate the detection of useful vs. non-useful reviews. Next, post-inspection, requirements author has to manually extract key problematic topics from useful reviews that can be mapped to individual requirements in an SRS to identify fault-prone requirements. The second goal of this research is to automate this mapping by employing Key phrase extraction (KPE) algorithms and semantic analysis (SA) approaches to identify fault-prone requirements. During fault-fixations, the author has to manually verify the requirements that could have been impacted by a fix. The third goal of my research is to assist the authors post-inspection to handle change impact analysis (CIA) during fault fixation using NL processing with semantic analysis and mining solutions from graph theory. The selection of quality inspectors during inspections is pertinent to be able to carry out post-inspection tasks accurately. The fourth goal of this research is to identify skilled inspectors using various classification and feature selection approaches. The dissertation has led to the development of automated solution that can identify useful reviews, help identify skilled inspectors, extract most prominent topics/keyphrases from fault logs; and help RE author during the fault-fixation post inspection.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Gursimran Singh Walia for his invaluable guidance and support in helping me finish this dissertation work. His enthusiasm and profound knowledge in the discipline of empirical software engineering are a source of my inspiration.

I am very grateful to my committee members Dr. Saeed Salem, Dr. Jen Li, and Dr. Limin Zhang for their unfailing support, advising, and for taking the time to help me finish this dissertation work.

I would also like to thank my lab colleagues Vaibhav Kumar Anu, Anurag Goswami, and Rupinder Kaur for all the assistance that they provided me during my research.

I thank all my family members here in US, Canada and in my home country, India. I would like to acknowledge the support of my wife Gurbinder Kaur, who managed everything all by herself while I was busy in my lab.

Last but not the least, I am grateful to the almighty for blessing me the courage and the strength to overcome all the challenges during my dissertation.

DEDICATION

I dedicate my work to my parents (Gurdeep Singh and Parminder Kaur) who supported me with their well wishes and motivated me throughout my whole educational career. Without their support it would have been impossible for to finish this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AMI	Adjusted Mutual Information
BNB	Bernoulli Naïve Bayes
CIA	Change Impact Analysis
DT	Decision Tree
ET	Extra Trees
KESRI	Keyphrase Extraction in Software Requirement Inspections
LSA	Latent Semantic Analysis
LDA	Latent Dirichlet Allocation
MNB	Multinomial Naïve Bayes
MOKSA	Mapping Of Keyphrases to SRS Approach
NL	Natural Language.
NLP	Natural Language Processing
NB	Naïve Bayes
NMI	Normalized Mutual Information
NuSVC	Nu Support Vector Classification
POS	Part Of Speech
RUS	Random Under Sampling
ROS	Random Over Sampling
RF	Random Forest
SE	Software Engineering
SMOTE	Synthetic Minority Oversampling TEchnique
SGD	Stochastic Gradient Descent
SVC	Support Vector Classification

1. INTRODUCTION

This chapter presents the problem statement, dissertation goals, a brief description of key concepts, and the research framework for the rest of this dissertation document.

1.1. Problem Statement

Software development starts with gathering requirements from various stakeholders (some of whom are non-technical) and producing a natural language (NL) software requirement specification (SRS) document. The SRS document forms the basis for downstream software development activities i.e., design, coding, testing. Due to the inherent nature of NL and involvement of multiple stakeholders, requirements are prone to redundancy, inconsistency, and ambiguity.

To verify requirement recorded in SRS, software companies employ peer-reviews (also referred as inspections or walkthroughs) to ensure that requirements meet certain quality standards (e.g., correctness, completeness). During a peer-review (will be referred to as inspection in this document), skilled inspectors are staffed who review the requirements document and report any potential faults. The faults are then handed back to the requirements author who must manually read through each reported review to identify useful reviews (that report actual fault) and remove non-useful reviews (false positives). Next, when fixing true faults in SRS, the author may need to manually check parts of the SRS that are affected (e.g., may contain similar faults) and need fixing and also avoid reintroducing new faults. This task to manually consolidate faults, manual mapping of useful reviews to SRS requirements, and search for requirements that need similar fixes is tedious, hectic, and time consuming. If information about useful reviews, skilled inspectors, and potential fault-prone requirements can be

automated, then the development time would be better spent towards actual software development.

SRS requirements are highly interrelated and fault fixation post-inspection may produce another fault in the SRS. So, to ensure the validity of the SRS and to verify the impacts of any change occurred during fault fixation, the requirements author has to again manually inspect the whole SRS document, which is very tiresome process. So, this research aims at developing an automated approach using machine learning (ML) to guide the change impact analysis (CIA) during fault fixations to assist requirements author post inspection.

Additionally, prior research [1]–[3], shows that that the quality of inspectors is most vital to inspection output as skilled inspectors can report a large number of faults more accurately and clearly (reducing false positives and redundancy in their performance). Using skilled inspectors also enable time savings because requirements author can validate reviews faster, and a better written review is also better when training the machine-based automated approaches. Therefore, this research aims to automate the following aspects of requirements quality.

1.1.1. Proposed Solutions

A brief description of research solutions for the above listed problems are discussed below:

Validating reported requirement reviews*:* Prior research has utilized different variants of Machine Learning (ML) based techniques (families of Naïve Bayes (NB), Support Vector Machines (SVM), Decision Trees (DT), Regression and Ensemble classification) to validate textual reviews in other domains (e.g., movie reviews, product reviews) [4]–[7]. This dissertation is leveraging existing work and applying it to unstructured requirement reviews to automate the identification of true faults based on fault logs supplied by requirement inspectors.

Identify potential fault prone requirements*:* This step is being automated by automating the key phrase extraction (KPE) from fault logs and then mapping these keyphrases to individual SRS requirements to identify fault prone requirements in an SRS document. Prior research on existing key phrase algorithms [8]–[11], algorithms to identify semantically similar NL text [12]–[14] is being leveraged to ensure that mapping between key phrases from fault logs (extracted using KPE algorithms) to individual SRS requirements (using semantic similarity) is accurate.

Identify interrelated requirements (IRR) in a SRS pre and post-inspection*:* To enable post-inspection fault fixation, this dissertation uses semantic analysis research [12], [15]–[19] to find interrelated requirements (IRR) by implementing algorithms like Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA). The similarities obtained for SRS document with semantic algorithms are then transformed into graph using graph mining algorithms [20]. Pre-inspection, IRR graph can be analyzed for loosely connected requirements (that may be extraneous), redundant requirements, and other fault-prone requirements. Post-inspection, IRR can be used to identify requirements that may need similar fixes when fixing the faults reported during the requirements inspection.

Automating the selection of skilled inspectors: To enable the selection of skilled inspectors, this work tries to identify generalizable characteristics of an effective inspector (e.g., reading patterns (RPs)). Data collected during previous eye tracking studies of requirements inspectors were used to train the ML models that can guide the selection of skilled inspectors prior to the start of inspection process.

1.2. Dissertation Goals

This dissertation leverages existing ML solutions from range of domains (code, design and testing [4], [21]–[25]) to help professional software engineers better manage the requirements inspection process, enable cost-savings and develop a better software product. The primary goal of this dissertation is formally defined as follows:

To identify and analyze various machine learning, natural language processing, semantic analysis and graph mining approaches that can be applied to validate reviews, to select skilled inspectors, to find the fault prone requirements, and evaluating CIA in an SRS document to improve quality of software requirements.

Another goal of this dissertation is to advance research into the use of ML and NLP methods to improve the quality of software requirements. Additionally, this dissertation shows that ML applications developed during this research can be extended to research problems in other domains. As an example, this dissertation proposed and subsequently validated a vertical mining approach that can support distributed processing of very large graph network. The vertical mining approach also supports dynamic update of output without having to re-compute from beginning (details appear in Chapter 8).

The rest of this chapter describes pertinent key terms relevant, challenges, and research framework for this research.

1.3. Key Terms and Key Concepts

There have been few prominent definitions and terms that have been used throughout this dissertation and these are briefly discussed in this section. These key concepts are as follows:

1.3.1. Machine Learning, Classification and Clustering

The idea of machine learning (ML) is to give computer systems the ability to learn i.e. learning some behavior of given tasks and then use that learned behavior to make decisions later on some new/unseen data. The intent of ML is to make predictions with data using statistical, and feature based techniques to alleviate the need to perform predictions based on human judgment [26], [27]. These predictions help analysts, researchers, scientists, and engineers to produce reliable decisions (classification or clustering) and to uncover hidden relationships within the data. The ability to make data-driven predictions, allows ML to overcome the need for strictly static programs by building a model from sample data (input). The ML approach is applied in a wide range of computing problems such as email filtering, text processing, intrusion detection in a network, computer vision, and optical character recognition [28]–[36].

Machine learning can be categorized into two parts 1) supervised learning and 2) unsupervised learning. ML approach that requires to be trained on some training data before it can predict outcome on test data is a supervised learning approach (e.g. classification) and the ML approach that do not requires to be trained on any training data is an unsupervised learning approach (e.g. clustering).

The process of classification entails the assignment of 'class' label (e.g. true-fault or false-positive) to test dataset and is called '*classification'*. A classifier (a defined algorithm that learn over the input data to predict final outcome class) is employed that maps unforeseen data from the learned model to assign an appropriate class. For example, Naive Bayes.

Clustering on the other hand groups data w.r.t their similarities. Clustering does not learn *class-labels,* but instead measures the distance between data (e.g. based on mean, standard deviation etc.) and then divide it systematically to form coherent grouping.

1.3.2. Ensemble Method

Ensemble is a ML algorithm that combines several meta-algorithms to form one predictive model. It (ensemble) is developed in order to improve classification outcome, in which many weak classifiers (i.e. situation in which individual classifiers could not improve prediction accuracy) learns iteratively to improve predictive performance. The ensemble is used in this dissertation is developed from various ML classifiers to generate voting to arrive at a categorization of each review into a useful or non-useful review.

1.3.3. Natural Language Processing and Part of Speech (POS) Tags

Natural language (NL) processing combines computer science, computational linguistics and artificial intelligence in order to find ways for machines to understand, analyze and construct meaning from human language. Using NLP, programmers can structure knowledge to perform many language processing tasks e.g. translation, semantic relation extraction, speech recognition, sentiment analysis and topic segmentation, etc., [4], [6], [12], [18], [37]–[39].

NLP allows the machines to understand human language based on grammar, semantic relations and natural language rules such that machines could parse the input text to extract the queried information. Natural language toolkit (NLTK) provides capabilities required to process NL text. Using NLTK, a given text can be split into various part of speech (POS) components that helps in understanding the semantic constructs of the text. Many studies have shown applications of NLP at providing automated solutions (e.g. summarization of text blocks [40], [41]), and development of Chat bots (a deep learning model that interacts with humans) [42]. NLP has been extensively applied for pre-processing NL text (i.e., removal of stopwords, extracting POS tags, stemming and lemmatization)

The POS in NLP is the word category that have similar grammatical properties. The words that are categorized within same POS tend to have similar behavior for language syntax. The common examples for POS in English are verb, pronoun, noun, adjectives, adverbs, preposition, interjection, conjunction and determiners. The process to analyze POS tags from NL text is called '*part of speech tagging*' (POS tagging). Tagging involves marking of each word in a corpus to its POS. POS tagging is helpful in identifying correct sentence structuring based on grammar syntax i.e. POS tagging can express correct word-form for a word that expresses multiple meanings. For example, consider the following sentence:

"I *left* my wallet on the *left* side of my bed*"*.

The word '*left*' in above example sentence appears in this sentence in two different word-form, though the spellings are similar. So, to differentiate between two different word-forms, part of speech tags are helpful. The outcome of above sentence with part of speech tags is:

I *(personal pronoun)* ***left*** *(verb past tense)* ***my*** *(possessive pronoun)* ***wallet*** *(Noun)* ***on*** *(preposition)* ***the*** *(determiner)* ***left*** *(adjective)* ***side*** *(Noun)* ***of*** *(Preposition)* ***my*** *(Possessive pronoun)* ***chair*** *(Noun)*

The importance of part of speech tagging (or POS tags) in text processing is their ability to provide information on lexical structure of text/sentence. Information about the lexical structure provide the ability to understand a text over its syntax, that help in understanding meaning, structure and type of affixes (a morpheme that is attached to the word stem) that it takes.

1.3.4. Key Phrase Extraction (KPE)

Key Phrase Extraction (KPE) is a process of extracting prominent (i.e., key) information from a NL text but in fewer words (referred to as keyphrases) and is achieved with unsupervised

and supervised approaches [11]. This dissertation work leveraged existing KPE approaches to extract key phrases from fault logs to identify problematic area in software requirements. For example, consider the following fault reported during the review.

"*The initial value of variableR was defined to be 10000. But now it is defined as 1000. Inconsistent value of variableR*."

If an estimate about problem highlighted by this fault log is to be obtained; then it can be achieved by extracting a fewer key phrases automatically, such as "*Initial value variableR*", and "*Inconsistent value variableR*". Our proposed work then maps these keyphrases (using similarity measures) to the actual requirements in an SRS document to find fault prone requirements.

1.3.5. Graph Mining

The graphs are the pictorial representation to display data in an orderly and organized manner. A simple graph consists of nodes and edges, where nodes represent the data and the edge connecting two nodes represents the relationship between those two nodes. For example, a graph can be modelled in which the nodes represent the requirement number and edges represent similarity between two requirements.

Next, the graph mining refers to mining various patterns (of node relationships) in a graph that conveys useful information. For example, mining highly similar requirements from the IRRs in a graph using cliques, k-cliques. These IRRs can help assist requirements author evaluating CIA during fault fixations.

1.3.6. Key Algorithms Used

The algorithms used in this dissertation work are divided around the following major goals and are shown in Table 1.1.

Goal 1: Automation of validation process of requirement reviews.

Table 1.1. Key algorithms used

Goal #	Algorithms
1	Various supervised learning algorithms from Bayesian, Support vector, Ensemble, Trees, and Regression.
2	Three main categories of algorithms namely, supervised feature based, unsupervised statistical based, and unsupervised graph based
3	Semantic similarity approaches, e.g., latent semantic analysis, graph mining algorithms
4	Supervised learning approaches, feature selection algorithms

Goal 2: Identify the fault prone requirements in an SRS.

Goal 3: Identify the technique to guide CIA during fault fixation post inspections.

Goal 4: Identify the characteristics of skilled inspectors.

1.3.7. Evaluation Metrics

To predict classification effectiveness at fault prediction, a metrics called confusion matrix is generally used by ML community (see Table 1.2). This matrix consisted of rows that represent predicted outcome class by a classifier whereas, columns represent actual class (see Table 1.2).

Table 1.2. Evaluation metrics

		Actual Class	
		Fault	**Non-fault**
Predicted Class	**Fault**	True positive	False positive
	Non-fault	False negative	True negative

For a classifier to be efficient, it is highly expected to be able to significantly predict true positives and true negatives accurately. Various evaluation measures that asses classification efficiency are defined in Table 1.3. Apart from this, three other metrics are used to address goal 2, and these metrics are Rand index, adjusted mutual information (AMI), and normalized mutual information (NMI). Their proof lies beyond the scope of this dissertation, so it is not discussed here (more details appear in chapter 6).

Table 1.3. Various metrics and their description

SR #	Name of the Metric	Description	Formulae
1	# of true positives (TP)	This is # of positives samples i.e., total # of true faults	# of actual faults
2	# of false positives (FP)	These are non-faults that got predicted as faults.	# of falsely predicted faults
3	# of false negatives (FN)	These are actual faults predicted as non-faults.	# of falsely predicted non-faults
4	Precision	This metric denoted how many predicted faults are actual faults.	TP/ (TP+FP)
5	Recall	It is the measure that denoted how many actual faults are predicted as faults. In other words, the proportion of true faults that are identified by a classifier.	TP/ (TP+FN)
6	F-measure	This is harmonic mean of precision and recall.	2TP/ (2TP+FP+FN)
7	G-mean	It is a performance metric that consider true negative rate and true positive rate.	√precision x recall

1.3.8. Inspection Artifacts and Inspectors

Inspection studies were conducted at North Dakota State University (NDSU) to generate inspection reviews that were used for validating the proposed approaches. There were four main artifacts that used during the dissertation - Parking Garage Control System (PGCS), Loan Arranger System (LAS), Restaurant Interactive Menu (RIM), and Wonders Of Weather (WOW). LAS and PGCS artifacts were developed by requirement experts and is used at Microsoft to train their employees over fault checklist technique to carry out inspections [15], [43], [44].

RIM and WOW were developed under real project conditions through interaction with real client. All documents (except WOW) contained seeded faults that were available to authors in the form of master fault list. The inspectors' reviews were checked against master fault lists to label each review as fault or non-fault. The WOW SRS was only used to evaluate the CIA approach because of its short length and comparatively small number of requirements than the other SRS documents. Overall, there were in total 82 reviewers involved across all the inspection

studies. Table 1.4 shows more details on participating subjects w.r.t undergraduate, graduate and industry inspectors. Most of these subjects had at least 2 years of experience in software.

Table 1.4. Various SRS artifacts used in this dissertation

SRS	# of Seeded (Known faults)	# of Pages	# of inspectors in this study
LAS	30	11	20 (industry personnel)
PGCS	35	14	41 (27 UG, 14 Grad)
RIM	150	21	21 (Grad)
WOW	NA	6	NA

1.4. Research Framework

To summarize, this subsection provides a high-level description (see Figure 1.1) of research framework and is organized around 4 major goals as described below:

***Goal 1- Automating the validation of requirement reviews*:** Development of a classification method (to classify NL requirement reviews into true faults and false positives) and its testing of effectiveness is achieved in the following steps:

1. Identify the list of classifiers most applicable to inspection reviews from literature.
2. Plan and execute inspection studies to collect NL inspection reviews across multiple SRS documents that can be used as training and testing sets during evaluation.
3. Use NLP toolkit to pre-process reviews (removal of stop words, punctuations, articles, stemming/lemmatization etc. from a review sentence) that outputs review sentences without non-useful terms.
4. Implement various classifiers identified in step 1 over output reviews to perform classification of inspection reviews into true positives vs. false positives.
5. Evaluate the classification results using evaluation metrics (precision, recall, F-measure and G-mean) for each classifier that is used on various training and testing sets.

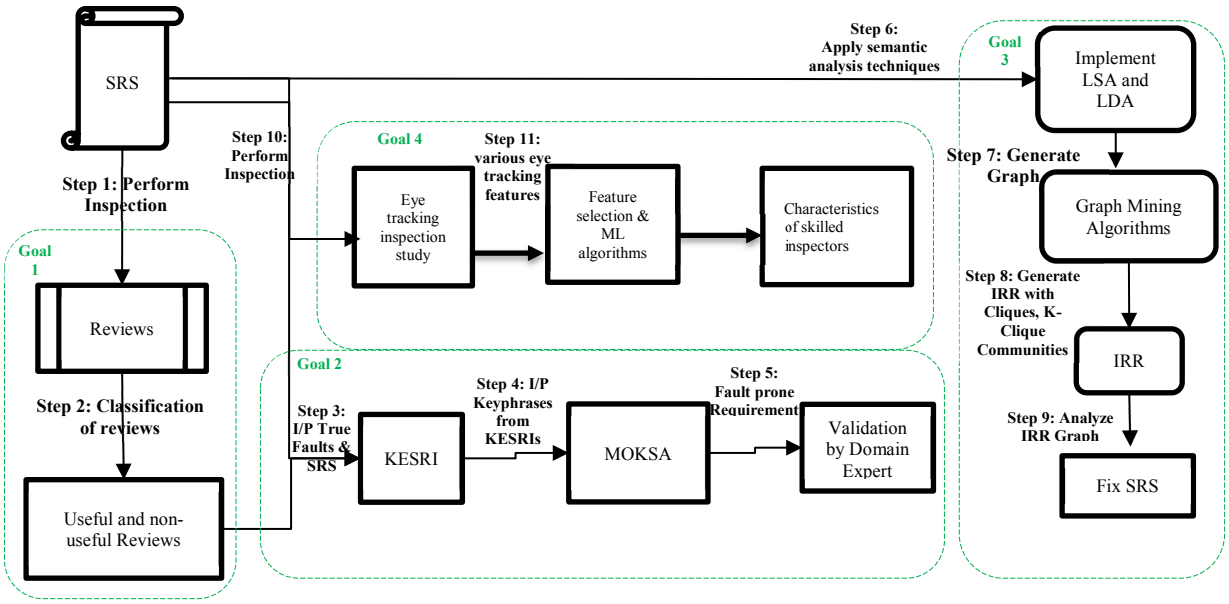

Figure 1.1. Research framework

***Goal 2- Identify the fault prone requirements in an SRS*:** Another dissertation goal is to provide requirement authors (post inspections) the list of fault prone requirements from requirement reviews through the following steps:

1. Implement the KESRI (Keyphrase Extraction for Software Requirements Inspections) approach that input an SRS and true faults validated as result of goal 1 (step 3).
2. The KESRI approach adapts various Key Phrase Extraction (KPE) algorithms to output the set of top ranked prominent phrases extracted from requirement reviews. These phrases provide the best contextual information about the fault (but in fewer words) without restricting the inspectors to adhere to any specific NL guidelines while documenting faults.
3. The output from KESRI approach are the top ranked keyphrases (step 4), which are input to the MOKSA (Mapping Of Keyphrases to SRS Approach). The MOKSA approach maps these keyphrases to SRS using semantic similarity approaches.

4. The output of MOKSA (step 5) is the list of fault prone requirements in an SRS and these fault prone requirements are validated by domain expert. The result from goal 2 would enable requirements author with additional assist to fix all fault prone requirements post inspections and that can save a lot of valuable time.

***Goal 3- Identification of a technique to guide CIA during fault fixation post inspections*:** This goal assists the requirements authors in handling CIA during fault fixation post inspections. This goal helps the author with post inspection analysis of SRS document for strongly interrelated requirements (IRRs) using graph mining that are most likely be impacted by a change in one requirement.

1. Implement a Latent Semantic Analysis (LSA) algorithm to develop similarity score matrix to establish relevance between each requirement in an SRS document (Step 6).
2. Query each requirement against similarity score obtained in previous step, to find similarity of that query with other requirements (indexes) in SRS document. Store these indexes on a file.
3. Generate an undirected graph from the indexes stored on the file (step 7).
4. Implement graph mining algorithms (clique, k-clique community etc.) over the graph generated (in step 7) to find strongly interrelated requirements (IRRs). This is step 8 in goal 3.
5. Analyze interrelated requirements (from step 8) and label highly impacted requirements in an SRS document based on for strongly related requirements (i.e. cliques, K-Cliques). Post-inspection, requirements author can verify CIA based on these related requirements and can fix fault in IRRs (Step 9 in Figure 1.1).

***Goal 4- Selection of skilled inspectors*:** This goal aims at finding generalizable characteristics of skilled inspectors that can find higher number of faults. The selection of skilled inspectors is achieved in the following steps:

1. The inspection study is performed using eye tracking equipment with PGCS SRS document at NDSU (step 10 in Figure 1.1). The output of this inspection is various features e.g., reading patterns, time spent per page, number of linear saccades scanned during inspections, etc., (step 11 in Figure 1.1).
2. The features obtained in step 11 are analyzed using principal component analysis (a technique to evaluate the most informative features).
3. Using the principal features evaluated in previous step are then used to train the ML models. The trained models are next tested using a test data to obtain the prediction. The features that best train the ML approach are labels as characteristic features to select the skilled inspectors.

The results from this goal can help requirement authors determine the skilled inspectors that may have reported higher number of faults. The requirements author can priorities reviews of such a skilled inspector.

2. BACKGROUND

This chapter discusses the existing work on ML and NL techniques most relevant to the dissertation focus areas.

2.1. Machine Learning in Software Engineering

In software engineering (SE), ML has been applied to predict software defects during code, design and testing phases [17], [23], [45]–[48]. Most of the related work has applied ML on code and design reviews (unstructured text) to help software developers fix those faults in order to improve software quality [16], [17], [22], [49]–[53]. These studies have shown that the use of automated tools (e.g. Gerrit for code reviews) [54]–[57] results in improved quality. These tools utilize version histories of a software project (i.e., code/design changes) to recommend problematic classes/design in a software project.

To the best of my knowledge, ML approaches have not yet been applied to automate requirements inspection activities. Therefore, this dissertation is a first attempt at developing an automated approach to validate requirement reviews and find fault-prone areas in an SRS post inspection (refer chapter 1 for more details).

In terms of validating reviews with supervised learning methods; Bosu et al., [4], Chen et al., [5] and Agarwal et al., [6] developed an approach to validate textual data (from code reviews, mobile apps, and tweets from twitter respectively) and categorize them into useful or non-useful reviews. These ML approaches (classification) use the training reviews to predict the outcome of test reviews. However, the results from application of these ML approaches showed a propensity towards misclassification errors (i.e., a false review is labeled as true and vice versa). This research tries to leverage the prior work on ML approaches and extend it on requirement reviews (details appear in chapter 3).

2.2. Machine Learning Algorithms

Some of the major ML algorithms and techniques identified during the literature search are discussed below and are organized around major dissertation goals.

Classifiers for validating requirement reviews: Stefan et al. [58] reported most applicable classification families (Statistical classifiers, Support Vector Machines, Ensemble, Decision Trees etc.) that can be used to predict fault prone modules. This dissertation plan to develop a technique using these prominent classifiers to validate inspection reviews and develop a recommender system (that produces final review categorization outcome). The proposed recommender system, in this dissertation, uses a voting method to predict the final outcome from various selected classifiers. Voting is a process in which the final outcome to a review is assigned based on majority prediction from all selected classifiers. The applicability of the voting method is motivated by Sun et al. [47], that showed that voting predicts the most number of faults, as well as most variant fault-types and that diverse classifiers, make better ensemble than individual classifiers.

***Keyphrase extraction algorithms for identification of fault-prone requirements*:** One of our goal aims at finding fault prone requirements by extracting prominent keyphrases from requirement reviews and then mapping them to individual requirements in an SRS. Keyphrase Extraction (KPE) is a process of extracting prominent phrases from an NL text with unsupervised and supervised learning algorithms [11]. This dissertation leverage existing KPE algorithms to extract prominent phrases from fault logs (requirement reviews) to identify the problematic requirements in an SRS. Most relevant KPE algorithms reported in literature [20], [59], [60], [6], [7], [16], [57], [61], [62], [75]–[77], can be categorized into three categories and are discussed below:

- ***Graph-based algorithms***: Graph-based KPE algorithms (e.g., Mihalcea et al., [63]) models the NL text into a graph, where key phrases are the nodes and the edges are the lexical structuring between two POS tags e.g., Noun-Verbs. These algorithms report that the best key phrases are extracted with Noun-Adjective lexical structuring. Their results guided my research at identifying lexical structuring most applicable to NL text in software requirements. Wan et al. [64], reported a similar approach (*called CollabRank on Single-Document cluster*) that clustered the documents before extracting keyphrases. This approach ranks prominent keyphrases higher because of their higher relevance within a document cluster. Other algorithms that use graph-based models include Multipartite Rank by [65], a supervised approach called PositionRank by [10], and LDA based TopicRank algorithm proposed by [9][66].
- ***Statistical algorithms***: These are unsupervised learning algorithms and in general use TFIDF score (to generate significance) for a term in a given text to identify prominent keyphrases from the NL text. Some approaches such as, KPMiner by El-Beltagy and Rafea [8], developed a KPE (*called KP-Miner*) from multilingual documents that can extract keyphrases without requiring training document for their algorithm. The prominent phrases are extracted through generation of candidate terms (based on TFIDF score) followed by weight assignment to rank the terms. Other similar models that can extract key phrases from NL text include YAKE [67], TOPIA [61], and RAKE [16], [61].
- ***Feature-based algorithms***: Witten et al. [68] developed a supervised learning algorithm called automatic keyphrase extraction (*KEA*) from text using lexical methods. Their approach generates the feature values for candidate terms based on frequencies of occurrence and then employ ML algorithms to identify the best keyphrase set. Similarly,

Nguyen et al. [69], proposed a model known as WINGNUS that generates key phrases using logical structures of the documents during candidate identification.

KPE algorithms have been applied in software requirements domain to automatically extract glossary items [16] and to check conformance to NL requirements templates [57]. Aguilera and Berry [16] extracted phrases from an NL requirements document by identifying important topics in the text and then extracting prominent phrases from those topics. Some researchers have reported KPE using POS (such as nouns, adjectives, etc.,), parsing, and by using heuristic measures such as, Euclidean distance between two terms [16]. Some researchers have explored KPE from NL text using graph models [63], multipartite graphs [65], ranking models [64][10][66], lexical methods [68], building general-purpose KPE system [8], logical structure [69], and collective multilingual KPE model [67]. Aforementioned categories of algorithms were implemented and adapted to support key phrase extraction from software requirements document.

***Identifying CIA technique to support fault-fixations post inspections*:** To assist requirements authors during fault fixation, two competing algorithms namely Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA) were analyzed. LSA is a semantic similarity technique based on a mathematical foundation known as singular value decomposition, whereas, LDA assigns similarity score to two words based on context in which they appear in a document [17], [40], [57], [70]. This decision of including LDA and LSA was guided by the evidence gathered from the literature. Within software engineering domain, semantic analysis has been applied to improve and automate various processes, e.g., extraction of glossaries, software requirements evolutions, finding similarities in bug-reports and software traceability.

Chetan et.al [16], used semantic analysis to extract candidate terms for glossary based on relevance between terms in a requirements document. They reported that using topic modeling in NL requirements resulted in 20% improvement in accuracy over generic term extraction tools. Similarly, the concept of semantic analysis was used on publicly available user reviews [19] to evolve requirements for future releases of the software. Software engineering researchers also applied semantic analysis [17] to automate the software traceability by using LDA to generate traceability links during software development. A similar work by Dit et.al [18] used the semantic similarity of comments in bug reports to detect duplicate bugs and save duplicated work.

Motivated by the prior literature findings, this dissertation aims at using semantic analysis to automatically locate fault-prone areas in software requirements and to extract most similar requirements. The successful application of semantic analysis approaches on SRS and requirement reviews can assist requirements engineers post inspections

***Selection of inspectors*:** This goal aimed at finding characteristics of skilled inspectors that are most likely to report higher 3 of faults during inspection, and these characteristics were identified using supervised learning algorithms (by training them on inspection data).

2.3. Natural Language Processing

Software engineering researchers have applied NLP to process NL text (i.e. code reviews, test reviews) to improve software quality [16], [57], [71], [72]. These studies developed and validated automated approaches that can find valuable features (POS tags, most informative and similar features) from NL text (code reviews, app reviews etc.) to improve their predictions. For example, Guzman et al., [73], used online app reviews to analyze users' feedback to identify app features that need improvement before next release. They showed that features extracted by their

NLP approach resulted in better software evolution task selection. Similarly, NL requirement reviews require a lot of pre-processing (i.e., removal of punctuations, stop words, removing slangs and fixing spelling mistakes), to extract the most valuable features. This research leverages work from studies [16], [57], [71], [72] that described NL preprocessing methods to convert unstructured reviews into more structured reviews.

2.4. Role of POS Tags in Mining NL Text

Natural language is built over part of speech (POS) tags like an adjective, adverbs, nouns, and verbs, etc., and all requirements are written by combining various POS tags [6], [7], [16], [57], [61], [62]. The identification of important POS tags that are pertinent to SRS quality and evaluation based on POS tags is one of the sub-goal of this dissertation.

On that end, this dissertation found some studies in literature like Agarwal et.al [6], in which they used Twitter data (again unstructured text) to classify tweets into categories based on their polarity (positives, negatives and neutral) using Tree-based classification approach and part of speech (POS) tags. They reported improved results by POS extraction while implementing ML algorithm. Another work in the classification of NL text was presented by Gimpel et.al [7] in which they analyzed Twitter tweets using POS tags. They mentioned that the most prominent POS tags that are present in any tweet are Nouns (N), Adjectives (J), Adverbs (R), punctuations, Verbs (V) and Determiners (DT). The above-discussed work related to unstructured text and the use of POS tags gave us insight on implementing our proposed approach to automate inspection reviews to develop guidelines to write high-quality SRS document. The development of high-quality SRS document can ensure comparatively less ambiguous requirements.

2.5. Graph Mining for Requirement Reviews

Graphs have become intriguingly important to model complicated structures e.g. protein structure, network structure, biological structures, webs (with various hyperlinks to different webpages), circuits and social network [20]. All the links within a graph are called edges and an edge connects two nodes. Nodes may represent various locations for a path search problem, two different web pages in hyperlink evaluation in webs and so on. Many algorithms have been developed for mining crucial information such as shortest paths, centrality between-ness (to find out central node in a graph), to find isomorphic patterns (to locate similar patterns in a graph), clique mining (strongly connected component) [20].

As described in chapter 1, goal-3 aims at finding fault-prone areas in SRS document with the help of graph mining algorithms. Tang and Huan [74], [75], discussed the importance of node-centric detection for communities within a graph. A node-centric communities (in a graph) satisfies properties like reachability, mutuality and degrees (details in [75], [74]) for a specific node. Communities in graph include nodes that are closely-knit share some common traits. For example, a strong community with 5 nodes interconnected to each other in a social network graph may indicate that all the 5 nodes (that represent users or persons in this scenario) are mutual friends or family members or go to work together etc. A community contains edges between certain nodes. A community that contains an edge from each node to every other node is a strong community and is called a *clique*. The communities that share maximum number of edges across a given number of nodes (say N= 4) in a set of graph S, is said to be K-plex if every member is connected to N-K other members. K-plex are also known as relaxation on *cliques* i.e. for K=1 and N=4, the other members must be connected to 3 other members. A lot of work [20]

has been done to mine such strongly connected communities in a graph to locate *cliques* and *K-plex*.

On the similar track, requirements in a SRS document also have strong cohesion between other requirements in a document. These interrelated requirements in a SRS document could provide essential information on correlated requirements that are needed to be inspected while fixing for fault. The goal-3 in chapter 1 aims to find highly interconnected areas within a SRS document by mining for *cliques and K-plex*. The major challenge in this goal is to transform an entire SRS document into a graph to apply clique mining approaches.

3. AUTOMATING THE VALIDATION PROCESS OF REQUIREMENT REVIEWS

This chapter discusses the work to automate the validation of requirement reviews into useful vs. non-useful category. To automate the inspection process, few studies have been conducted at North Dakota State University (NDSU), with SRS documents namely RIM, PGCS and LAS (details can be found in chapter 1). This chapter presents extensive details about proposed approach, experiment design, experiment methodologies, results, and discussion of results regarding these three studies to evaluate the review automation process; and these are referred to as study 1 [72], study 2 [76], and study 3 [77] (in chronological order). These three studies are described as sub-goals of goal-1 (see Figure 3.1).

3.1. Study 1 - Validation of Inspection Reviews Over Variable Feature Set Threshold

This study was conducted to identify the optimal number of features required and to identify prominent POS tags most suited to train model for validation of software reviews. Unlike prior studies, this study included reviews from a semantically similar domain (of movies) in addition to domain specific reviews (i.e., inspection reviews). This was done to understand the quality of features extracted from semantically similar domain, and the effect the large volume of features had on evaluation metrics. Another motivation to include movie reviews was to overcome biased classification due to class-imbalance problem (discussed later) with inspection reviews.

To overcome class-imbalance, certain sampling techniques (random under sampling, random oversampling, synthetic minority oversampling technique, boosting etc.) were implemented for inspection reviews to select the most applicable sampling technique to inspection reviews and movie reviews.

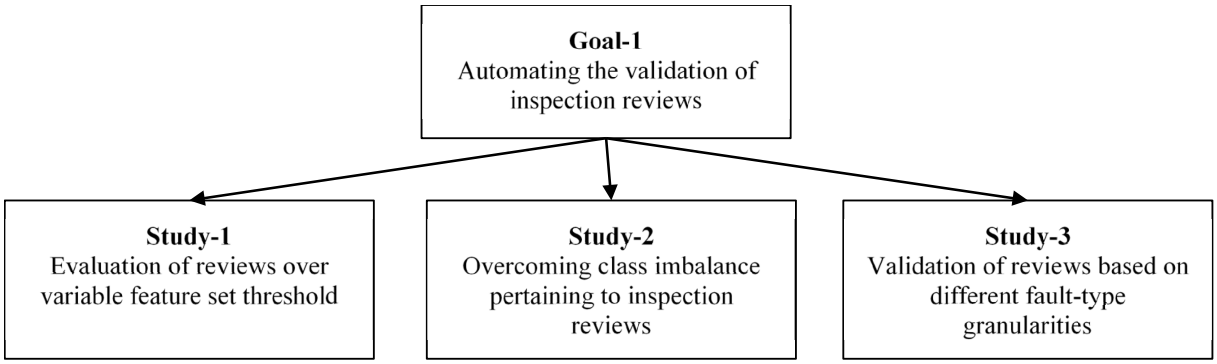

Figure 3.1. Studies conducted to achieve goal-1

Next, the complete feature set from both the domains was ranked separately based on their informative importance and was then divided into 20 equal intervals (each interval containing 5% of features). After that, a training model for each feature set interval and six POS tags (Nouns, Adverbs, Verbs, Adjectives, Determiners, and without-POS) was trained, to plot their classification prediction (for test set) against evaluation metrics (F-measure and G-mean). Finally, the outcomes (F-measure, G-mean etc.,) obtained for training models (inspection and movies) by manipulating number of features and POS tags, were contrasted to observe prominent POS tags and best feature set range. Extensive details can be found in [72] while some prominent results from this study-1 have been described in following subsequent sections.

3.1.1. Proposed Approach

Our proposed mining approach used supervised learning classifiers, reviews (requirements and movies) and features generated during study run to evaluate the accuracy and G-mean (i.e., geometric mean of precision and recall). Two types of review data sets (requirement reviews and online movie reviews) were used in this study to develop two different training models (*inspection trained* vs. *movie trained* respectively).

The reviews generated online for movies were 10 times larger than software requirement reviews, were more readily available online, and were evaluated for addressing class imbalance

problem in mining. The reviews were divided equally into two categories: positive and negative. Positive reviews in context to movies used positive feedback to describe a movie while negative reviews used detrimental feedback to describe a movie. The study-1 used '*positive*' category of movie reviews to train the model to detect false-positives and '*negative*' category to detect true-faults.

The features were extracted from both '*movies*' and '*inspection*' reviews. The whole feature-set for both the reviews ('*movies*' and *'inspection'*) was divided into 20 intervals (5% to 100%) of equal size to perform analysis within each feature-set interval. The model was trained 20 times by gradually increasing the size by 5% each time until all features were used to train the model. The '*movies trained*' and inspection trained model had a total of 11318 and 368 features respectively. The following example explains the idea behind extracting features with the help of NL text.

3.1.1.1. Feature set generation

Feature set generation is explained with an example (see Table 3.1) using three reviews. Two reviews in Table 3.1 are true-faults while one review is a false-positive. Each word in these reviews is a feature and collection of all the features from all the reviews in each category makes feature set.

Table 3.1. Example of feature reviews categories

Reviews	**Category**
The working of system in heavy load is not tested.	Fault
System load is not tested.	Fault
Who opened the gate?	False-positive

The extracted feature set is shown in Table 3.2. The feature "system: 2" is interpreted as "system" being the unique feature and the number after colons in "system: 2" denotes the

frequency of that feature in a fault class. A similar feature set as generated in Table 3.2 was used to train the models using supervised learning classifiers.

Table 3.2. Example of feature set generation

Reviews	**Class**	**Feature set size**
"working:1", "system:2", "heavy:1", "load:2", "not:2", "tested:2"	Fault	6
"Who:1", "opened:1", "gate:1"	False-positive	3

3.1.1.2. Assigning category to test review

The assignment of category (i.e. fault or false-positive) to a test review is explained with the following example sentence which is tested against the feature set generated in Table 3.2. The test review is as follows:

"System variable is not tested".

In this example, total size of the feature set is 9 (6 from fault category and 3 from non-fault); various frequent occurring words in English such as 'the', 'in' and 'is' are removed as part of pre-processing task to create more informative and descriptive feature set. Our proposed approach extracted all unique features from a training sentence and stored them along with their frequency of occurrence. The test sentence was then tested against the feature set developed during training to assign a final category. As seen, the features from an example test sentence, "system" and "tested" have higher frequencies in "fault" of Table 3.2, while none feature occurs in false-positive class. So, this test sentence was assigned fault class because more *informative features* in fault class were matched.

3.1.1.3. Feature set generation using POS tags

POS tags are grammatical tagging or word-category disambiguation to mark up a word in text corpus corresponding to its lexical categories. Study-1 proposed an approach in conjunction

with POS tags to perform preliminary analysis of classification results for inspection reviews and movie reviews. Gimpel et al. [7], reported in their research that in any unstructured text, the most important POS tags are Nouns, Adjectives, Verbs, Adverbs and Determiners. They claimed that "the *extraction of only important POS out of an unstructured sentence delivered apposite sense*". So, the study-1 trained classification model over feature sets obtained for each of the POS tags as guided by Gimpel et al., in [7]. The analysis was performed over 20 intervals (5% to 100%) of feature-set for each POS tag and during the analysis, it was observed that both the models performed poorer when trained over determiners. The reasons for this poor performance was due to use of '*engish stopword*' that removed most of the determiners (like 'the', 'an', 'that' etc.) during preprocessing steps and the determiners did not provide any important information about context.

Additionally, there were very low number of features generated during training on determiners for both the models (less than 50 in inspection trained and less than 100 in movies trained); so the determiners were removed from the analysis. The experiment design, experiment procedure, research questions investigated are discussed in next section.

3.1.2. Experiment Design and Procedure

Some details about major RQs, variables exploited in study-1 are discussed in this section as follows:

3.1.2.1. Major research questions

There were two major research questions that were investigated in study-1 are as follows.

RQ-1: Does training our mining approach on movie reviews and using part of speech (POS) tags overcome class imbalance problem associated with inspection reviews?

RQ-2: What impact does the size of features set included in training model had on evaluation metrics (F-measure and G-mean) of validating reviews?

3.1.2.2. Variables used

There were independent and dependent variables that were included into experiment. These are discussed as follows:

- *Independent Variables***:** these are the variables that were manipulated during study run. The independent variables in this study were nine types of classifiers used (see Table 3.3), training models (inspection trained and movie trained), feature set size (20 equal intervals) and six prominent part of speech (POS) tags (namely Nouns (N), Adjectives (J), Verbs (V), Adverbs (R), Determiners (D), and without-POS) that were used in this study.
- *Dependent Variables***:** These are the variables that were measured to record the effect of independent variables. Evaluation metrics (Geometric mean of precision and recall, F-measure) and Threshold Value (the percentage of features needed to train the model to achieve higher G-mean) were used.

Table 3.3. Various classifiers used in this study

Classification Family	**Name of Classifiers**
Bayesian	Naive Bayes (NB), Multinomial NB, and Bernoulli NB
Support Vector Classification (SVC)	Linear SVC and NuSVC
Ensemble	Random Forest and Extra Trees
Regression	Logistic and Stochastic Gradient Descent (SGD)
Trees	Decision Trees

Next, the details regarding various artifacts and subjects participated in this study are discussed.

3.1.2.3. Artifacts and participating subjects

There were two sets of artifacts that were used in study-1, where, one set consisted of all the reviews belonging to inspection of Parking Garage Control System (PGCS) SRS document and the other set consisted of reviews from closely related domain of movies. A total of 41 inspectors from software engineering (SE) course at NDSU (27 undergraduates and 14 graduates) participated and performed an inspection on software artifact that generated reviews used in this study.

3.1.3. Results and Analysis

This section report results regarding the application of two differently trained models (*'inspection' vs. 'movie'*) and six POS tags (Nouns, Adverbs, Verbs, Adjectives, Determiners, and 'without-POS') at validating requirement reviews. The results compare G-mean and F-measure of test-set that was tested across varying features (full feature set divided into 20 equal intervals) of each training model type. The percentage of features varied (5% to 100%), were plotted along X-axis while G-mean and F-measure was plotted along Y-axis (Figure 3.2 and Figure 3.3). The response variable (evaluation metrics) was observed for various POS explanatory variables over different feature set %age; W-POS, POS-J, POS-N, POS-R and POS-V stands for 'Without POS', 'POS adjective (J)', 'POS noun (N)', 'POS adverb (R)' and 'POS verb (V)' respectively. The results shown in Figure 3.2 and Figure 3.3 were analyzed and discussed around the two RQ's described earlier.

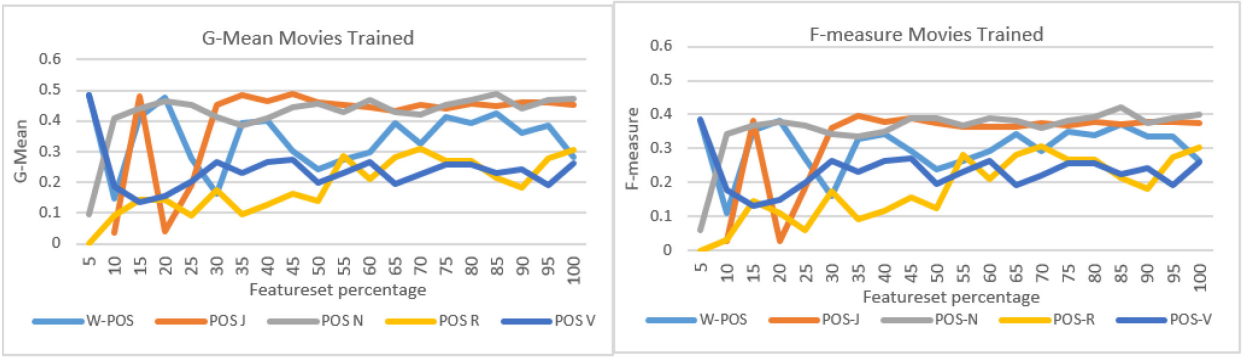

Figure 3.2. Results of movie trained model for G-mean and F-measure

3.1.3.1. Results for RQ-1 treated for class-imbalance

The purpose of this requirement question was to investigate if reviews from a semantically similar domain (movie domain) in NL context can overcome classification prediction of inspection model that was treated for class-imbalance threat. So, five POS tags as shown in Figure 3.2 and Figure 3.3, were analyzed for both the models trained on inspection and movies. Some major outcomes for this RQ are discussed below:

- The model training (for both movie and inspection reviews) using W-POS (Without POS) did not show any stable trend in evaluation metrics.
- Movie trained and Inspection trained model showed a stable trend for two part of speech tags (namely POS-N and POS-J).
- Movie trained performed slightly better than inspection trained w.r.t Stable trend of POS-N and POS-J.

Implications from results of RQ-1: Inspection trained model showed slightly better performance than movies trained and the reason behind this was that the test reviews and training reviews belonged to same domain. The POS tags Nouns and Adjectives were prominent in both the models because reviewers tend to emphasize more on Nouns and Adjectives while reporting a fault. It was also observed that movies trained model yielded better stable trend (for POS-N

and POJ-J) than inspection trained because movie trained were larger corpus and consisted of most naturally occurring instances of reviews compared to inspection trained (where ROS was used to address class imbalance).

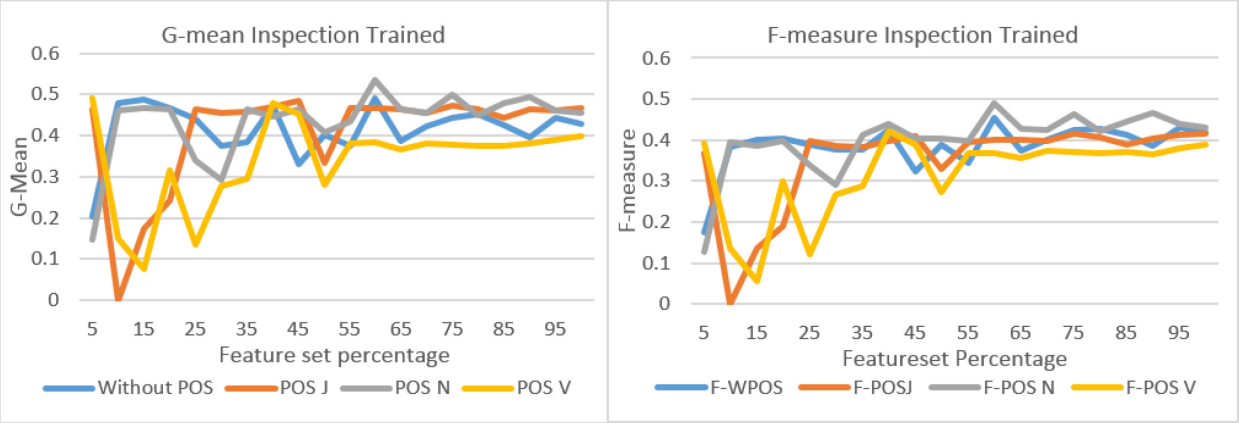

Figure 3.3. Results of G-mean and F-measure of inspection trained model

In conclusion, few inferences drawn from RQ-1 were the performance of both the models can be improved with POS tags and reviews from semantically similar domain (movie trained) could help in generating better stable trends (with specific POS tags) for evaluation metrics as compared to inspections domain (treated for class-imbalance).

3.1.3.2. Results from RQ-2 treated for optimal feature set threshold

This requirement question was investigated to study the threshold for most informative features-to-include in training model to achieve stable results. Each model was analyzed over complete range of feature set (i.e. from 5% to 100% features), where, the complete feature set was divided into intervals of 5% forming 20 equally spaced intervals. The reason to choose 20 equal intervals was to observe the results by gradually adding small but significant number of features in next iteration, to sketch a refined prediction for complete data set.

Next, the G-mean and F-measure was observed for the most balance percentage interval beyond which the gain in performance becomes stable. The other reason towards observing for

such a threshold interval was to minimize the tendency to misclassification error. The analysis was performed for each model using POS tags and variable feature sets against testing set (Figure 3.2 and Figure 3.3). The X-axis shows the percentage of features taken to train the model and Y-axis shows evaluation metrics. Some major outcomes with this RQ's are discussed as below:

- W-POS tag was found under-performing w.r.t other POS tags for both the models over entire range of features.
- For movie-trained model, It was observed that a stable G-mean and F-measure was obtained for POS-N and POS-J beyond feature set percentage range of 45%, whereas for inspection trained, the stable trend for POS-N and POS-J was seen beyond feature set interval range of 65%.

Implications from results of RQ-2: From the observations above, there were few implications about the results. *First*, validation of reviews was better performed with POS tags (POS-N and POS-J), and *second*, the most reliable feature set range with POS tags that gave the best performance was 45% (for movie trained) and 65% (for inspection trained).

3.1.4. Discussion of Results

The result presented in section 3.1.3 showed that POS tags (POS-N and POS-J) were most prominent. In this section, the evaluation regarding the results are presented for POS-N and POS-J with R-square and P-values that predict model's capability. Next, the polynomial regression was used to fit the model's prediction to maximum polynomial degree of three because beyond that the model resulted in over-fitting and complexity, such that it did not perform well on any new/unseen data.

3.1.4.1. Discussion on RQ-1 treated for class-imbalance

The discussion in this section involves the applicability of movie reviews to validate inspection reviews. The R-squared values and P-values are discussed for both the models w.r.t evaluation metrics (discussed in Section 3.1.2). Table 3.4 below shows the R-square values and P-values for the POS tags that showed best stable trend among all POS tags considered for analysis. W-POS tag was also considered to evaluate the difference between the use of POS tags and without POS. The significant P-values are made bold and the discussion is presented below:

- Movies trained model showed the better performance as the P-values for this model were significant with 95% of confidence for both POS-N and POS-J.
- On the other side, W-POS failed to show any significant R-square value for any of the evaluation metric.

Table 3.4. R-square and P-values for both trained models

Trained models	**Inspection Trained**		**Movies Trained**	
	G-mean	**F-measure**	**G-mean**	**F-measure**
POS-J	0.3827	0.4751	0.6135 (2)	0.6431
P-Value	0.095836	**0.034265**	**0.004016**	**0.002223**
POS-N	0.3489 (3)	0.5455 (3)	0.5266 (3)	0.5797 (3)
P-Value	0.13163	**0.012856**	**0.01706**	**0.007384**
W-POS	0.1103 (3)	0.2997 (3)	0.1158 (3)	0.1001 (3)
P-Value	0.6434	0.199229	0.626847	0.674563

The major implication from Table 3.4 was that using movies trained model provided significant results for both the evaluation metrics (G-mean and F-measure). The class imbalance can be improved by using reviews from a balanced review-set taken from semantically similar domain.

3.1.4.2. Discussion on RQ-2 treated for optimal feature set threshold

This section presents the discussion on percentage of feature sets required to include while training a model. The discussion is presented for POS-N and POS-J for movies trained model in Figure 3.3.

- The R-square value for POS-J (0.6431 in Figure 3.4b) and POS-J (0.6135 in Figure 3.4a) showed that our model was able to predict 64.31% of the variance in response variable.
- The R-square values shown in Figure 3.4 were significant with P-value test at 95% confidence level.
- In Figure 3.4, it was observed that for both POS-N and POS-J, the polynomial regression curve exactly fits the data points. POS-N data was exactly fit by polynomial regression over interval (35% to 60%) while POS-J data was fit over interval (40% to 90%) for both G-mean and F-measure.
- From results in section 3.1.2, it was observed that the stable behavior in G-mean and F-measure was shown beyond feature set interval 45%. This can be seen from Figure 3.4 that beyond the interval value 45%, the polynomial curve exactly fits the data points for POS-N and POS-J.

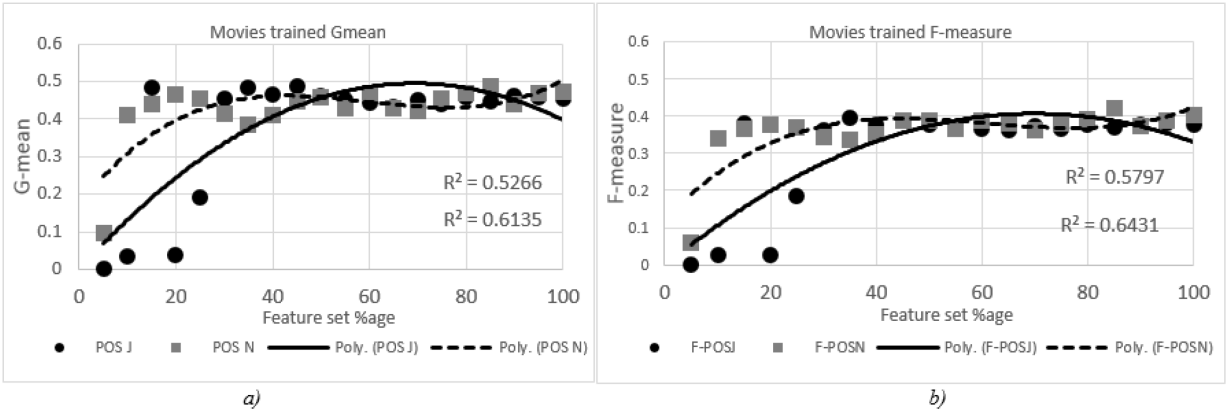

Figure 3.4. R-square and P-value analysis of movies trained data for POS-N and POS-J

3.2. Study 2 - An Empirical Investigation to Overcome Class-imbalance

The focus of study-2 was on validating useful (true-faults) versus false-positives (non-faults) reviews, overcoming class-imbalance (using sampling and various POS tags combination). Study-2 also focused on restricting misclassification error rate due to false classification prediction. The objectives for study-2 are derived from the future work of study-1 and which are discussed as follows:

- The classification results were analyzed over individual POS tags i.e. considering just one POS tag at a time like noun, adjective, adverb etc., and their various combinations can be used to further explore model effectiveness on requirement reviews.
- The classification accuracy (in terms of G-mean and F-measure) was between 50-60%, and it was not significant and contained misclassification errors. So, study-2 explored ways to restrict misclassification.

Study-1 provided some good insights on analyzing the classification problem with POS tags and reviews from semantically similar domain. Study-2 used these important insights from study-1 to train and analyze the models. Again, two sets of reviews were taken i.e. inspection and movies (same as in study-1). The class imbalance problem is handled by introducing some techniques like AdaBoost, SMOTE (Synthetic Minority Over-Sampling TEchnique), Random Over-Sampling (ROS) and Random Under-Sampling (RUS) techniques etc. More details about study-2 can be found in [76] and some details of study-2 about experiment procedure, design, methodology etc. are discussed in following subsequent sections.

3.2.1. Proposed Approach

Study-2 developed an automated mining approach to validate requirement reviews using supervised learning classifiers, NLP over various combinations of POS tags. In this section,

major components of study-2 are discussed at identifying true faults vs. false-positives in requirement reviews.

3.2.1.1. Selection of classifiers, training and testing sets

This section discusses selection of classification model, natural language processing and POS tags and training data set used to develop, and testing set used to test those classification models.

***Supervised learning classifiers (model selection)*:** Nine (9) different classifiers were used to develop a training model that classified each individual review into a fault or a false-positive category (Same as in study 1 and shown in Table 3.3). The classifiers were selected based on prior literature studies [6][20], [21] and based on the results during cross-validation (performed as part of model selection). Each review was classified based on the majority outcome from all nine classifiers (i.e. voting). To avoid situations of equal prediction conflict, an odd number of classifiers were used in this research to be able to classify each review either as fault or false-positive.

***Natural language and POS tags*:** NL requirement reviews were analyzed using same POS tags as used in study 1 (selected based on literature search [22]). Briand et al. [22], applied POS tags over twitter data and found that most valuable information can be extracted from POS tags like nouns (N), pronouns (PR), verbs (V), adjectives (J), determiner (D) and adverb (R). For example, the POS tagging is explained through the following inspection review:

"This requirement does not provide error handling."

The dis-integration of this review over POS tags is shown below:

"This/D requirement/N does/V not/R provide/V error/N handling/V"

Determiners were removed as part of stop words in NL text. So, the final POS tags analyzed were Adverbs (R), Verbs (V), Nouns (N), and Adjectives (J) along with their all their 16 combinations (e.g., RV, JRV).

***Training and testing data sets*:** Study-2 was conducted with same reviews (used in study 1) generated from PGCS inspection document. Total number of reported faults (reviews) were 857 (201 true faults and 656 false positives). The 10-fold cross-validation was used and repeated 10 times over nine different classifiers. Next, the cross-validation score was tested for all these nine selected classifiers using AdaBoost, RUS, ROS, SMOTE [17], [18], [20], and without sampling to address class imbalance problem (refer Table 3.5). Table 3.5 shows the mean score of all 10 repetition of cross-validation.

Table 3.5. Cross validation results for model selection

Classifiers	**Without Sampling**	**Ada Boost**	**RUS**	**ROS**	**SMOTE**
Naïve Bayes	0.721	0.717	0.622	0.737	0.746
Multinomial Naïve Bayes	0.798	0.723	0.676	0.825	0.831
Bernoulli Naïve Bayes	0.776	0.750	0.660	0.831	0.880
Decision Tree	0.761	0.77	0.637	0.883	0.840
Linear SVC	0.813	0.816	0.704	0.895	0.905
Random Forest	0.805	0.808	0.670	0.925	0.919
Extra Tree	0.818	0.819	0.682	0.951	0.938
Logistic Regression	0.794	0.751	0.691	0.852	0.857
SGD	0.775	0.732	0.684	0.871	0.890

RUS, AdaBoost and without sampling methods were discarded because they resulted in low cross-validation score as compared to ROS and SMOTE (Table 3.5). It was observed that ROS and SMOTE performed almost identical in cross-validation, but ROS was chosen for study-2 to address class imbalance because it out-performed SMOTE for decision-trees, Random Forest, extra trees (main ensemble classifiers). Also, it was easy to implement and less time

consuming to deliver results. The percentage split for training and testing data for both the models was 70% train and 30% test data (shown in Figure 3.5).

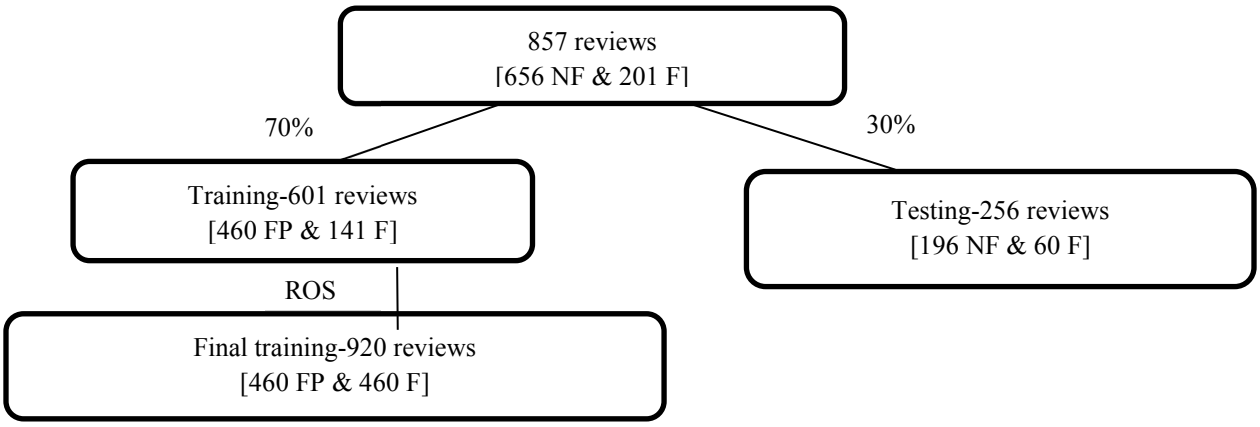

Figure 3.5. Training and testing set split for study 2

Models trained over requirement reviews are also referred to as 'inspection trained' and contained 920 requirement reviews (after ROS sampling). The ROS of test set is not performed to avoid redundant test reviews and to avoid excessive misclassification error. The model trained over movies reviews (that are publicly available at [59]) was referred to as '*movie trained*'.

The next sub-section discusses the working of proposed approach of study-2.

3.2.1.2. Working of proposed approach

This section describes the process of training and testing our mining approach to classify a review into a true-fault or a false-positive.

Training models and testing sets: As mentioned earlier, there were a total of 4 different POS tags (without considering determiners and W-POS) along with their all possible combinations ("J and R", "J and V", "R and V" and "J, R and V" etc.) to train our models. This resulted in a total of 15 different combinations of POS tags. Additionally, one model was trained without using POS (a conventional approach that acted as a control group). This resulted in 16

different ways to train each model (separately for '*inspection trained*' and '*movie trained*'). The following example explains the process to train '*movies review*' model on Adjectives ("J"):

***Training*:** Each movie review was read by the model; and Adjectives ('J') as described earlier, were extracted from it. The extracted part from a review was called a *feature* and all extracted features from all the reviews formed a '*feature set*' for Adjectives (J). This collection of feature set (for 'J') represented the trained model over POS tag 'J'. This process was repeated for all 16 combinations of J, R, N and V including none.

***Testing*:** Each review from test set was tested against the trained model (developed during training for POS 'J'). When a review was tested against model trained on Adjective ('J'), then same POS tag ('J' in this case) was extracted from the test review. The extracted POS tag from test review was tested against feature set with all nine different classifiers (Table 3.3) to classify into a fault or false-positive (based on the majority outcome).

Priority classes and confidence analysis: The priority analysis was performed (similar analysis that has been used successfully in the domain of movie reviews [24][25]) to determine the priority of requirement review. Priority classes described extent of review's confidence in each class based on the priority value (higher the value more the confidence). The models were trained using 16 different model instances but used top 10 POS combinations out of 16. The formation of priority classes and confidence analysis is explained below:

***Confidence analysis*:** Every review in a test set was tracked when tested against the training model to develop its classification outcome. The analysis was performed for both trained models and the classification outcome for each review was combined from all the model instances (16 POS in total) to obtain confidence value. For example, if out of 16 POS model instances, 12 categorized a review into as true-fault, then that review is assigned a fault-

confidence value of 0.75 (12/16) and a false-positive confidence value of 0.25 (4/16). This analysis was replicated for all reviews.

***Priority classes*:** The priority classes were used to analyze the dispersal of review confidence i.e. based on confidence-value. Priority classes assign unique interval to each review within each category (fault or false positives) based on extent of confidence; which defines the value with which a review was being classified to a category. For example, based on the total number of trained model instances (e.g., assumed 10), there were 10 different priority classes i.e. class-1 through class-10 for each category. The review assigned to class-1 represented the least confidence i.e. that review has least value of 0.1 with confidence of 10%. Each priority class had a unique confidence interval that defined the level of confidence a review was assigned to that category e.g. a review with fault confidence value of 0.7 (7/10) was assigned to priority class-7 (confidence 70%) as a true-fault. This implied that the same review is classified as a false-positive with a confidence of 30%. The advantage of assigning priority class to each review was that it helped to make efficient post-inspection decision i.e. reviews from class with high confidence most likely ensured accurate classification of those reviews.

Table 3.6. Common POS tags across both models for interval-1 to interval-3

Models	**Interval 1**				**Interval 2**				**Interval 3**				**Interval 4**			
Movies	**JNR**	**JN**	**N**	**RV**	**J**	**JR**	**JNV**	JRVN	**R**	JV	**JRV**	**NRV**	NV	WPOS	NR	V
Inspection	**JNR**	**JN**	**JR**	**JNV**	NV	**NRV**	NR	**J**	**R**	**N**	**JRV**	**RV**	JRVN	JV	V	WPOS

Selection of POS tags for priority classes: One motive to include movie reviews was to find the generalized POS tags that could be applied to a similar domain in NL context. To find such generalized POS tags, the G-mean values obtained during training of model instances were analyzed for all 16 POS tags. The generalized POS tags were selected out of total 16 in a way that could assure the selection of most effective POS tags. Total 16 POS tags (sorted in

descending order of G-mean) from both trained models were divided into 4 equal consecutive intervals (see Table 3.6) and each interval contained 4 POS tags i.e. interval-1 contained top 4 tags for both inspection and movies trained model; interval-2 contained next 4 top POS tags and so on up to interval-4 and together all sums to 16. The evaluation of priority classes was performed separately for four cases: *Firstly*, evaluation over interval-1 that output only two common POS tags (JNR and JN); *Secondly,* evaluation over consecutive interval-1 and 2 that output 5 common POS tags (JNR, JN, JR, J and JNV); *Thirdly,* evaluation of interval 1 to 3 that output 10 common POS tags among both models (see bold and underlined POS tags in Table 3.6); *Fourthly,* over all intervals that output all POS tags. Top three intervals were chosen based on their most precise G-mean values that output 10 common/generalized POS tags, and the results were discussed based on these 10 POS tags that resulted in the formation of 10 priority classes.

3.2.2. Experiment Methodology

This section discusses major RQs, variables explored, and artifacts used in this study.

***Major research questions*:** There were two major research questions that were investigated in study-2 and these are as follows:

RQ-1: How accurately does models trained on inspection reviews vs. movie reviews classify useful vs. non-useful reviews (i.e. true faults vs. false positives)? (Using combinations of POS tags).

RQ-2: How does accuracy of true faults and false-positives spread across priority classes over part of speech (POS) tags?

3.2.2.1. Variables used

There were independent and dependent variables that were included into experiment. These are discussed as follows.

Independent variables: These variables were manipulated during study run (see [76] for more details) and these were the # of classifiers (i.e., 9), training models (i.e. 2), and the POS tags (N, J, R, and V) considered based on the results from study-1.

***Dependent variables*:** Evaluation metrics (Geometric mean of precision and recall, F-measure) and Threshold Value (the percentage of features needed to train the model to achieve higher G-mean) are used.

Artifacts and participating subjects: The same set of artifacts were used in this study that were used in study-1 (PGCS and movie reviews).

3.2.3. Results and Analysis

The analysis compared the results w.r.t evaluation metrics when used two different training models (inspection vs movie trained). Study-2 used the results from study-1 in determining the optimal percentage of features sets to train the models. Also, the prominent POS tags were chosen from study-1 to use their various combinations in developing priority classes. The results and important finding are discussed around the two RQ's are discussed in section 3.2.2.

3.2.3.1. Results from RQ-1 treated for validation of reviews

This research question was focused on investigating whether G-mean at classifying true-fault (using various combinations of POS tags) varies when using different training data sets (i.e., when trained on inspection reviews vs. online movie reviews). To answer this RQ, the G-mean values were compared for both training models among conventional approach (without

POS), approach using POS tags and our proposed approach with priority classes. The analysis of G-mean for these three approaches is discussed in subsections below.

***Conventional approach*:** This approach used supervised learning classifiers (listed in Table 3.3) without applying POS to classify reviews into either a true-fault or a false-positive (depending upon the majority outcome). The results regarding the number, percentage of correctly (and incorrectly) classified true faults (and false-positives) and G-mean for both training models are shown in Figure 3.6. The results in Figure 3.6 are presented in the form X (Y %) where 'X' being the number of correctly/incorrectly classified reviews and Y% represents corresponding percentage of correctly/incorrectly review. The results and major observations in Figure 3.6 are discussed below:

- Inspection trained models were more accurate at classifying non-useful reviews (i.e., false positives) whereas movie trained models didn't show a clear demarcation.
- While using conventional approach can help save time that is otherwise spent removing the false-positives, it was not very useful due to large amount of reviews that were incorrectly classified (especially into a true-fault category). This information lead project managers to make incorrect post-inspection decisions (e.g., estimating the number of faults remaining post inspection? and deciding whether to re-inspect the artifact?).

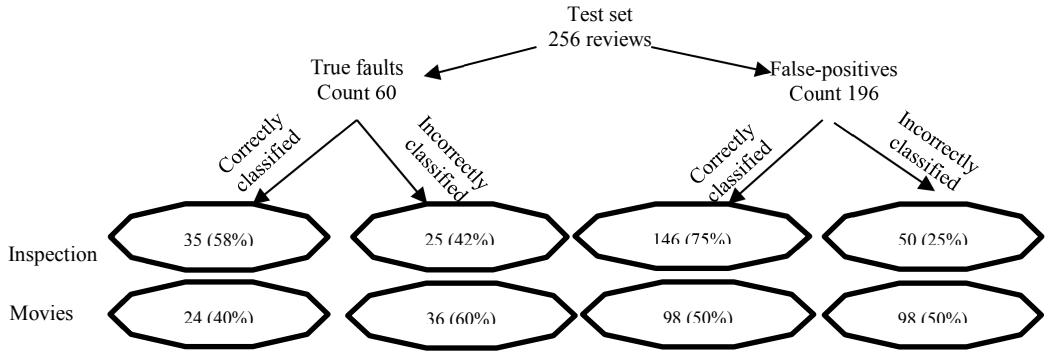

Figure 3.6. Classification results using conventional approach

***Implications*:** The conventional approach was prone to misclassification and offered a considerable room for improvement. So, it was not recommended to make post-inspection decisions.

***Classification with POS tags*:** Similar analysis of G-mean (for true faults and false-positives) across two training models was conducted using 10 (out of 16) most generalized POS tags. The results from this analysis are presented in Table 3.7 and discussed here within.

Table 3.7. Classification accuracy using POS tags

Variables	**POS TAGS**									
Movies Trained	**JNR**	**JN**	**N**	**J**	**RV**	**JR**	**JNV**	**R**	**JRV**	**NVR**
TN	14	19	19	37	65	42	41	48	96	101
TP	59	56	54	50	47	46	44	43	29	25
Precision	0.25	0.24	0.24	0.24	0.26	0.23	0.22	0.23	0.23	0.21
Recall	0.98	0.93	0.90	0.83	0.78	0.77	0.73	0.72	0.48	0.42
G-Mean	**0.49**	**0.47**	**0.46**	**0.45**	**0.46**	**0.42**	**0.40**	**0.40**	**0.33**	**0.29**
Inspection Trained										
TN	130	138	122	90	135	94	137	31	132	138
TP	42	40	37	44	30	46	36	52	34	35
Precision	0.39	0.41	0.33	0.29	0.33	0.31	0.38	0.24	0.35	0.38
Recall	0.7	0.67	0.62	0.73	0.5	0.77	0.6	0.87	0.57	0.58
G-Mean	**0.52**	**0.52**	**0.45**	**0.46**	**0.41**	**0.49**	**0.48**	**0.46**	**0.44**	**0.47**

Table 3.7 displays values of the following five metrics namely TN, TP, precision, recall and G-mean. Major observations are discussed below.

- The maximum number of non-useful (TN) reviews were detected by POS: 'NVR' for both the models. On the other hand, there couldn't be any POS tag that was able to detect significant faults for both movies trained and inspection trained model. The maximum G-mean value (√precision x recall) obtained at detecting true-faults is around 0.5 for both the models and this value corresponds to POS: JNR.

- The result (G-mean) was prominent for POS tag ‘JNR’ and ‘JN’ among both the models but still, this approach was prone to misclassification. In conclusion, using combinations of POS tags, the misclassification error rate was still present and is not recommended either for post-inspection decisions or over conventional approach (without using POS).

***Implications*:** Based on these results, the prominent POS tag combinations that showed the maximum G-mean value was for ‘JN’ and ‘JNR’. Referring to study-1, POS-J and POS-N were the prominent and so is there combination (‘JN’ in study-2) but interestingly in study-2, POS-R (adverbs) in combination with POS-JN has appeared to be important. Models with combination of POS tags to validate reviews showed improved prediction as compared to models that didn’t consider POS tags, but they still did not improve misclassification rate.

Additionally, a training models with combinations of POS tags, seemed good at classifying either true-faults or false-positives but not both. For example, POS tag ‘NVR’ resulted better at classifying non-faults for both the models but performed poorly in detecting faults. Therefore, it is very unlikely that using either model with POS technique could aid project managers or requirement engineers at accurately identifying both true-faults and fault-positives.

Results analysis using POS tag combinations: The classification accuracy was analyzed on training models (inspection trained vs movies trained) over priority classes and collective POS tags (in order to improve upon results obtained from approaches discussed in section 3.2.3). The results are described in Table 3.8 w.r.t number of true-faults/fault-positives accurately identified by our proposed approach when applied across 10 combination of POS tags (i.e. 10 most prominent POS tags among all combinations).

Table 3.8. Classification accuracy of fault & non-fault reviews using collective POS

Movies trained	**POS tags**										**Collective POS**
	JNR	**JN**	**N**	**J**	**RV**	**JNV**	**JR**	**JRV**	**NVR**	**R**	**True faults located**
Fault reviews	241	233	231	209	178	199	200	128	119	191	60 (100%)
Non-fault reviews	15	23	25	47	78	57	56	128	137	65	176 (90%)
Inspection trained											
Fault reviews	109	99	112	151	91	95	149	98	93	218	57 (95%)
Non-fault reviews	147	157	144	105	165	161	107	158	163	38	182 (93%)

The columns '*Collective POS*' collectively analyzes true-faults/fault-positives obtained from selected POS tags (Table 3.8). Each review in this approach was tracked and counted towards either fault or non-fault if it was found at least once in any POS tag combination i.e., if a review was identified as a true-fault by at least once out of 10 POS instances, then it was classified as a true-fault with 10% confidence. The idea for implementation of such an approach was to minimize misclassification error rate by implementing priority classes that adds confidence label to each review. The priority classes enabled requirement analyzer to make effective decisions based on confidence value rather than to ignore a review if true fault was incorrectly classified as non-fault. The major outcomes of this approach (Collective POS) are discussed below:

- ***Classification accuracy of faults***: Using collective POS approach, all true-faults were labelled (up to 95%) at least once into fault category in inspection trained and 100% true-faults into fault category in movies trained. Moreover, movies trained was more biased towards classification of a review into fault category.
- ***Classification accuracy of false-positives***: Using collective POS approach, inspection trained model was able to label 93% of non-faults at least once into false-positive

category and movies trained model could label 90% of non-faults at least once as non-fault.

***Implications*:** Collective POS approach resulted into more concrete solution to misclassification error rate. It guides the requirement analyzer/inspectors more efficient in making post-inspection decisions (e.g. when to stop re-inspection process).

3.2.3.2. Results from RQ-2 treated over priority classes

This section presents the analysis of true-faults vs false-positives accuracy for each priority class across each training model. The Acc% values signify the accuracy of a true-fault or false-positive within certain number of reviews in each priority class (Table 3.9).

Table 3.9. Results on categorization into priority classes

Priority Class/ Confidence	**Test set**			
	True Fault Reviews		**False-positive Reviews**	
	Movie trained (Acc %)	**Inspection trained (Acc %)**	**Movie trained (Acc %)**	**Inspection trained (Acc %)**
10% (1/10)	0 (0) =0	6 (46) =13	31 (36) =86	5 (13) =39
20% (2/10)	0 (1) =0	2 (14) =14	55 (78) =71	16 (22) =73
30% (3/10)	0 (0) =0	4 (45) =9	44 (54) =82	13 (15) =87
40% (4/10)	1 (6) =17	4 (18) =22	27 (37) =73	17 (21) =81
50% (5/10)	4 (17) =24	2 (14) =14	13 (17) =77	12 (14) =86
60% (6/10)	10 (37) =27	4 (21) =19	5 (6) =83	14 (18) =78
70% (7/10)	10 (54) =19	2 (15) =13	0 (0) =0	41 (45) =91
80% (8/10)	23 (78) =30	6 (22) =27	1 (1) =100	12 (14) =86
90% (9/10)	5 (36) =14	8 (13) =62	0 (0) =0	40 (46) =87
100% (10/10)	7 (27) =26	19 (33) =57	0 (0) =0	12 (15) =80

The higher Acc% values denote high density of true-fault/ false-positive within that priority class whereas the smaller value of Acc% signifies very sparse occurrence of true-faults or false-positives within that priority class. The outcome was combined from each POS model-instances to assign a priority class to the review.

Table 3.9 shows our results for true-faults and false-positives for both models (movie trained, and inspection trained). The values in the form X(Y) =Z in cells means X is the number of true-faults (or actual false-positives), Y is total number of reviews that were classified as faults (or false-positive) in a particular confidence class and Z is the accuracy obtained by dividing X by Y. The major observations from Table 3.9 are:

- ***Most reliable priority classes for identifying true-faults*:** Priority classes-8 and above for inspection trained model showed greater fault detection accuracy. Movies trained also performed reasonable in priority classes-8 and above.
- ***Most reliable priority classes for identifying false-positives*:** Priority classes-3 to class-10 were most reliable for identifying false-positive reviews for inspection trained model whereas priority class-1 to class-6 for movies trained model performed better.

***Implications*:** The major implications from this study were that the misclassification (TN) was curbed as each review was labelled with confidence percentage. Non-faults were better predicted by both the models, and the true faults were unevenly distributed across all priority classes.

3.2.4. Discussion of Results

The discussion is performed around the results shown in Table 3.8 and Table 3.9 that shows # of true faults/false positives across all priority classes for both models. The discussion is formulated below around the two research questions listed in section 3.2.2.

3.2.4.1. Discussion on results from RQ-1

From our results (Table 3.8), it was seen that there were improvements with 'movies trained' when compared with 'inspection trained' model. The improvements are listed below:

- One of the major improvements was that using 'movie reviews' to test reviews, all actual faults i.e., 60 (100%) were labelled as actual faults (see Table 3.8 for collective POS column).
- It was observed that movies trained model was more inclined towards classifying each review as fault for most of the POS tags whereas inspection trained was more inclined towards non-faults.
- The detection of false positives was significant for both the models with more than 90% labelled at least once into correct category (see collected POS column).
- The non-faults that were misclassified by movies trained (196-176=20 in number; Table 3.8) into faults (see movies trained fault priority class-10; Table 3.9) with 100% confidence were not problematic because that did not account for any true fault slippage. Similarly, this happened for inspection trained model with 14 non-faults misclassified into faults. On the contrary, any fault slipped through priority class (i.e. fault-misclassified into non-faults and were not categorized at least once as fault) was more problematic because that slippage would propagate to later stages.

It was observed that both the models could label a significant number of faults/non-faults into correct category. Considering the minority class (faults) prediction, movies trained addressed 100% faults. The requirement analyzer was thus provided with additional class label and assurance that misclassification is restricted; to assist him during post-inspection decisions.

3.2.4.2. Discussion on results from RQ-2

The purpose of this RQ was to find the accuracy of true faults and accuracy of false positives distributed within each priority class to investigate if there was any similarity for

accuracy across both trained models. For this purpose, each priority class was analyzed for both the models. Some major observations that were found interesting are discussed below:

- An improvement that was shown by 'movies trained' model was the capability to report all true fault within least number of priority classes. As seen in Table 3.9, movie trained listed all faults and non-faults within seven priority classes and showed that the fault and non-fault concentration was more within seven priority classes as compared to ten priority classes by inspection trained.
- It was observed that accuracy of true-fault detection in improved beyond 40% confidence value for both the models and was even greater beyond confidence level of 80%.
- For false-positive category, 'movies trained' model showed improvement for priority class-1 to class-6 (again within least number of priority classes) as compared to inspection trained. Both models performed significantly well within non-fault classes (Acc% is large).
- From Table 3.9, movies trained model for fault category in class-9, there were only 5 true faults out of 36 and the Acc% value is low (14%). Which implied that misclassification chances were more in this class as compared to other. The priority class-1 to class-3 have Acc% value equals to zero (0) implied that none or very few reviews were categorized into these classes and hence misclassification chances within these classes were also none or least.

The requirement analyzer is benefitted more of high fault/non-fault concentration that could provide better post-inspection decision (i.e. probability of true-faults/false-positives is more in each priority class). Moreover, movies trained model could report all true faults that

could ensure requirement analyzer that none out of total reported faults has slipped during classification.

3.3. Study 3 - Validating Requirements Reviews with Fault-Type Level Granularity

Based on the insights from study-1 and 2, where independent variables were manipulated like 'artifacts involved in the study', 'POS tags', and 'features sets' while keeping 'faults types', and 'various classifiers' constant. In study-3, the intent was to manipulate previously considered 'dependent variables' i.e. 'fault-types' and 'classifiers', to observe classification results. For this purpose, the effect of following variables was evaluated in this study-3:

- The concept of validation of reviews over various '*fault types*' was used to consider inspection reviews further into Ambiguous (A), Omission (O), Incorrect Facts (IF), Inconsistent Information (II), Miscellaneous (M) and Extraneous (E) categories instead of just labelling a review as true-fault vs non-fault.
- In study-3, variable ('type of classifiers') was manipulated to validate inspection reviews into its correct fault-type. This study focused on evaluating classification results when *firstly*, individual classifiers were evaluated for each fault-type, and *secondly*, ensembles formed from most efficient individual classifiers were developed to validate fault-types.
- The main contribution of study-3 was based on fault-type granularity (i.e. O, A, IF, II, E and M) to address some issues like finding most occurring fault-types, and fault-types that are most likely to be correctly classified by supervised learning classifiers.

3.3.1. Proposed Approach

Study 3 was executed at NDSU over the inspection data that was collected in previous three inspections studies. The data in the form of reviews was generated by inspection of LAS, RIM, and PGCS document in which a total of 82 inspectors participated that consisted of 35

graduate students, 27 under-graduate students and 20 experienced industry people. More details regarding our proposed approach is discussed in this section below.

3.3.1.1. Experiment design

This section discusses various RQs, variables, and data collection measures applied in this study. There were two major research questions that were investigated in study-3. The research questions are as follows:

*RQ-1***:** Which supervised learning classifiers, when applied to reviews collected during requirements inspections, can categorize individual fault-types into faults vs non-faults with higher accuracy?

RQ-2: Do ensembles created from the best performing individual classifiers provide improved categorization of reviews into faults vs non-faults?

Variables used in study-3: There were independent and dependent variables that were exploited in this experiment. These are discussed as follows:

- *Independent Variables***:** The independent variables in this study were the type of classifiers (nine classifiers are used), and fault types.
- *Dependent Variables:* Evaluation metrics (Geometric mean of precision and recall, F-measure) and Accuracy (It is a measure of true-faults and false-positives correctly detected by the classifiers) were used.

Artifacts and participating subjects for study-3: There were three sets of artifacts that were used in the experiment. The artifacts used were PGCS, RIM, and LAS. The artifacts and the number of inspectors are shown in Table 3.10. The details about breakdown of each fault-type and artifact type in Table 3.10 can be found in [77].

3.3.1.2. Preprocessing, model selection, testing and training sets

The details regarding the proposed approach are discussed in this section that includes information about model selection, formation of training and testing sets, and working of proposed approach.

Model selection: During our study run, each classifier (Table 3.3) was analyzed individually over each fault-type and the best classifiers were combined to form ensembles that performed voting to obtain majority outcome. The majority outcome was assigned based on voting of 9 classifiers. The odd number of classifiers was intentionally chosen for ensemble to avoid equal prediction conflict.

Pre-processing and formation of training/test sets: In this section, the details regarding pre-processing of reviews are presented to enable division of data into training and testing set.

Pre-processing: The inconsequential words were removed from the review corpus through NLTK's *stop words* method. Next, there were multiple reviews that contained words that appears in several inflicted forms (e.g. walk, walks, walking, walked conveys same meaning but are treated differently during text classification because of grammar rules). So, pre-processing of such words was taken care through *lemmatization* in NLTK package. Lemmatization returns the base form of the word i.e. in above example walk, walks, walking and walked are changed to walk. Sampling (ROS) was performed to address class imbalance problem.

Training model: The reviews were split into 70%-30% ratio for training and testing purpose. Training of reviews was performed on 70% of total reviews obtained from all three inspection studies. The inspection studies used three different requirements document (i.e. LAS, RIM and PGCS). Training data resulted in number of fault types as listed in Table 3.10.

Table 3.10. Fault distribution across fault-types and inspection documents

Fault type	LAS (Industry 20 sub)	PGCS		RIM (Grad 21 sub)	Total # of fault types	Train (LAS+ PGCS +RIM)	Test (LAS+ PGCS +RIM)
		Grad (14 sub)	UG (27 sub)				
A	7	12	24	14	57	41	16
O	10	16	29	21	76	53	23
IF	3	11	30	43	87	60	27
II	17	20	40	32	109	77	32
E	0	6	10	6	22	15	7
M	2	2	2	3	9	5	4
Total	**39**	**67**	**135**	**119**	**360**	**251**	**109**

This distribution (70% for training and 30% for testing) of faults across training and testing set was performed to ensure representation of each fault type (with similar distribution) in both training and test sets. Next, 10-fold cross validation was repeated 10 times to estimate the most effective classifiers to build training models.

***Test set*:** There was one test set that consisted of 457 number of reviews collected from all 3 studies (refer Figure 3.7). Our test set consisted of at least 30% of reviews from each inspection document. The selection of reviews in test set was random and through automation, it was ensured that it contains desired (70%-30%) instances of each fault type in both training and testing (see Figure 3.7). Moreover, the reviews in test set were not over-sampled in order to avoid duplicate misclassification error rate.

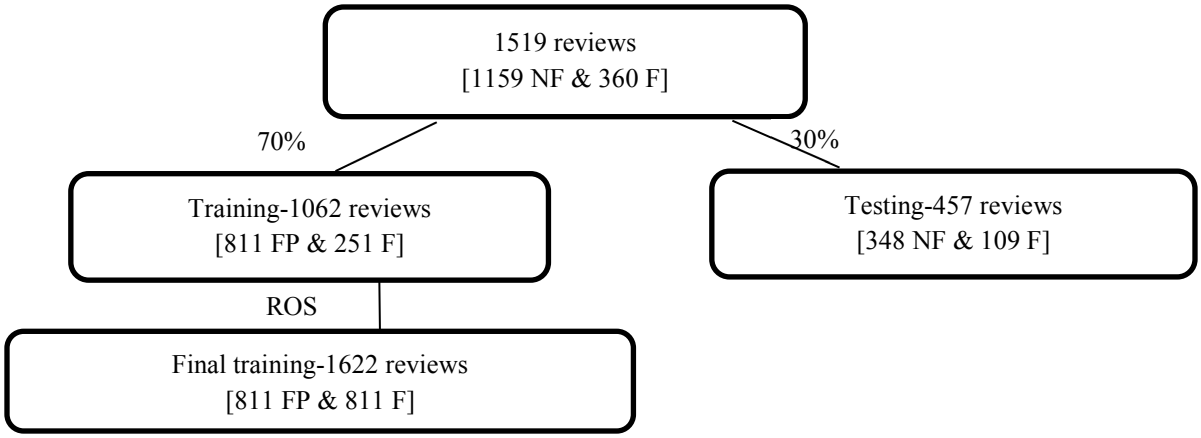

Figure 3.7. Training and testing set split of reviews for study 3

3.3.1.3. Experiment procedure

The working of our approach is discussed in this section that discusses the approach through which various fault-types are analyzed after classification.

Supervised learning classification: In this section, various supervised learning approaches are discussed that were used to train each classifier. The selection of these classifiers was based on their successful usage in previous studies (study 1 and study 2) to mine reviews. The ROS sampling was performed because it yielded best cross validation results (Table 3.11). Each review in test set belonged to at most one of the fault types (A, O, IF, II, M and E) and each review was tracked against each classifier to count correctly classified fault with its fault type. Next, each classification outcome was collected and analyzed to answer our research questions.

Voting through ensemble: Ensemble selected those classifiers that performed best against each fault-type and their combined prediction is used to generate voting to assign final class to each review. Our motive was to build ensemble based on most accurate classifiers for a fault-type. The aim of this analysis was to put more focus on certain fault-types that have not yet found in a requirement document. More details about study-3 can be found at [77] while some key details about this study are presented in following subsections.

Table 3.11. Cross validation results for model selection

Classifiers	Without Sampling	Ada Boost	RUS	ROS
Naïve Bayes (NB)	0.773	0.754	0.657	0.785
Multinomial NB	0.819	0.763	0.626	0.796
Bernoulli NB	0.817	0.822	0.604	0.809
Decision Tree	0.770	0.826	0.546	0.837
Linear SVC	0.824	0.850	0.576	0.852
Random Forest	0.818	0.849	0.584	0.857
Extra Tree	0.835	0.858	0.649	0.975
Logistic Regression	0.825	0.775	0.608	0.835
SGD	0.783	0.783	0.561	0.855
NuSVC	0.669	0.669	0.520	0.563

3.3.2. Results and Analysis

This section report results regarding classification accuracy of individual classifiers that are taken under consideration (Table 3.3). The analysis was performed around two research questions discussed earlier. The analysis discusses classifiers accuracy in predicting each fault-type as well as analysis of ensemble using most accurate classifiers to test each fault-types. The results are organized around the following two research questions:

3.3.2.1. Results from RQ-1

This research question was focused on investigating the performance of individual classifier against each fault-type described in this paper. The fault distribution around three documents is shown in Table 3.10. The evaluation was performed using Accuracy percentage and G-mean. The results are shown in Figure 3.8, Y-axis (vertical axis) represent Accuracy in percentage and X-axis (horizontal axis) shows various classifiers that are used to perform the analysis. The results for each fault-type is discussed below:

Fault-type A (Ambiguity): From Figure 3.8 (1), the classifiers were arranged in descending order of their accuracy results. There was a total of 16 faults in test set that represented fault-type ambiguity (Table 3.10, Test column). The accuracy varied from 81.25% (13 out of 16) to 37.5% (6 out of 16) for different classifiers. It was seen that only three (out of ten) classifiers performed well (for fault-type A) compared to the other classifiers and were Multinomial naïve Bayes (MNB), Naïve Bayes (NB) and Bernoulli Naïve Bayes (BNB), all belonging to Bayesian family of classifiers.

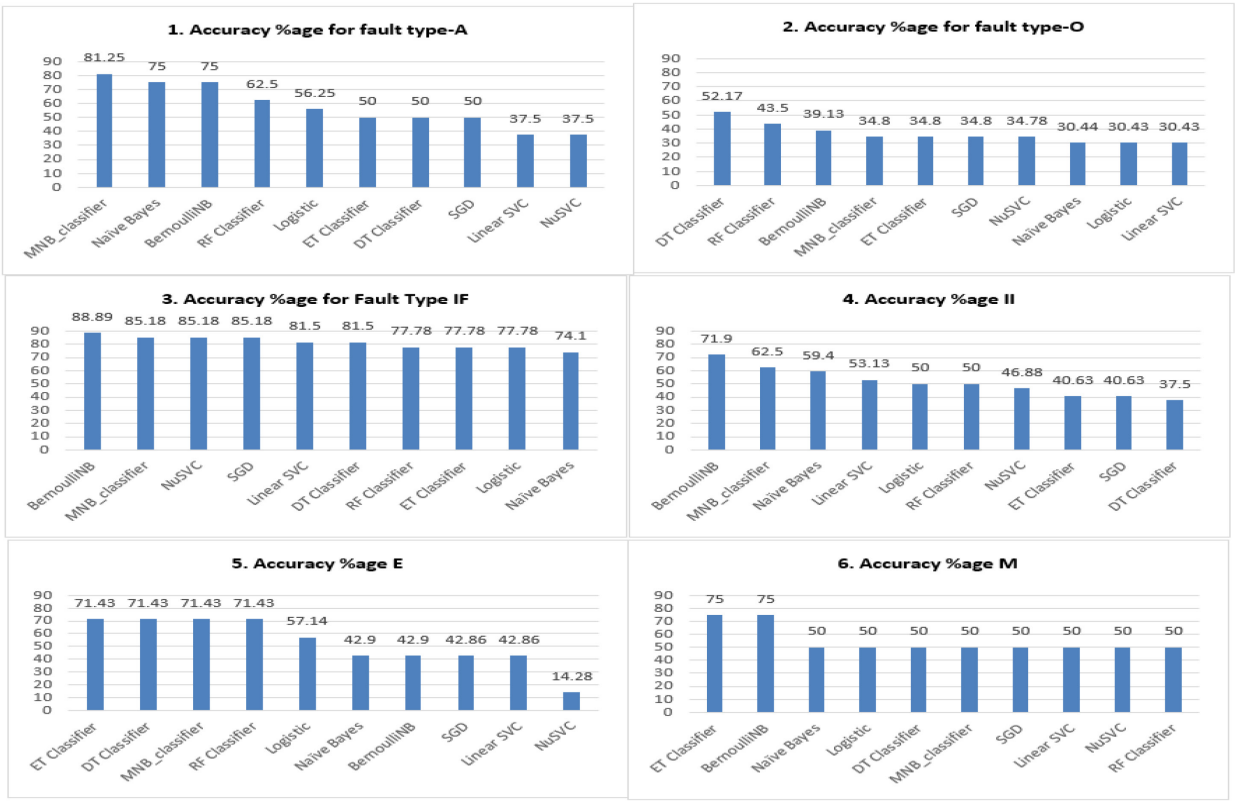

Figure 3.8. Results of fault type versus individual classifiers

Fault-type O (Omission): Figure 3.8 (2) showed omission fault-types that were 23 in total present in test set (Table 3.10). The accuracy for omission fault type varied from 52.17% (12 out of 23) to 30.43% (7 out of 23). The only classifier that showed highest prediction

accuracy against fault-type O was shown by Decision Tree (DT) classifier that belonged to Trees family of classifiers. The fault-type O was the least accurately predicted fault-type.

Fault-type IF (Incorrect Facts): Figure 3.8 (3) shows fault-type IF that were 27 in number. The classification accuracy was observed highest by Bernoulli NB classifier (90%) and went down to 74.1% (by Naïve Bayes). Overall, all classifiers performed well to predict fault-type IF as three classifiers (MNB, Nu Support Vector Classifier (NuSVC), and Stochastic Gradient Descent (SGD) showed 85.18% accuracy and five classifiers showed accuracy between 77% to 82%. So, almost all classifiers predicted IF fault with a significantly high accuracy.

Fault-type II (Inconsistent Information): Figure 3.8 (4) in shows that the accuracy varied from 71.9% (highest) to 37.5% (lowest) for fault-type II. There were 32 faults in total that belonged to fault-type II. Results showed that all three classifiers from Bayesian family are the top performers.

Fault-type E (Extraneous): There were 7 total Fault-type E in test set. There were four classifiers that equally performed and ranked top with 71.43% accuracy (Figure 3.8 (5)). These four classifiers were Extra Tree (ET) Classifier, Decision Tree (DT) Classifier, MNB and Random Forest (RF) Classifier. Interestingly, every classifier from Trees and Ensemble family performed well against fault-type E.

Fault-type M (Miscellaneous): Figure 3.8 (6), showed prediction results corresponding to fault-type miscellaneous (M). There were 4 faults in total and two classifiers Extra Tree (ET) and Bernoulli NB (BNB) were able to predict 75% (3 out of 4) accurately. Rest all classifiers were able to predict at most 50% of true faults.

3.3.2.2. Results for RQ-2

This research question was focused on investigating improvement in each fault-type when the *best-performing* individual classifiers were combined to create an ensemble that classified a test review through voting. The selection of classifiers for ensemble was done based on results shown for RQ-1. For each fault-type, the selection of classifiers for ensemble was almost different and was based on merit (see Figure 3.8). For our binary classification problem; it was required that the outcome be classified into either fault or non-fault. The merit of classifiers was based on accuracy percentage (i.e. include those classifiers that are above accuracy mean of all classifiers for each fault-type); for example, fault-type A (Figure 3.8 (1)) had a mean of 57.5% for all the classifiers (refer Table 3.12) and only first three classifiers were above this mean value.

Table 3.12. Classifiers that qualified for ensemble

Fault type	**Mean of 10 classifiers**	**# and Name of classifiers that qualified for ensemble**	**Mean Acc. % of qualified classifiers for ensemble**
Type-A	57.5	3 (NB, MNB and BNB)	77.1
Type-O	36.53	3 (DT, RF and BNB)	44.9
Type-IF	81.48	6 (BNB, MNB, SGD, NuSVC, Linear SVC and DT)	84.6
Type-II	51.26	4 (BNB, NB, MNB and Linear SVC)	61.7
Type-E	52.87	4 (ET, DT, RF and MNB)	71.43
Type-M	55	2 (BNB and ET)	75

In fault-type-A example, the number of classifiers were odd but if it were even then this even number tie was resolved based on overall G-mean (Figure 3.8) score of classifiers (classifier with greater G-mean score was included for each even tie). G-mean was also considered to evaluate improvement (in accuracy if any) of ensemble over individual classifier

(Figure 3.9). The following observations were noted for each ensemble that was created for each fault-type:

Ensemble for fault-type A: From Table 3.12, three classifiers that qualify for ensemble were NB, MNB and BNB (notably, all were from Bayesian family). As the number of classifiers that qualified is odd, there did not exist equal prediction conflict, so all three of these classifiers were chosen to build ensemble for fault type-A. The improvement with ensemble was seen w.r.t G-mean score and classification accuracy (refer Figure 3.9). The results in Figure 3.9 showed classification accuracy with targeted ensemble along with comparison of improvement in G-mean for each fault-type (Figure 3.10).

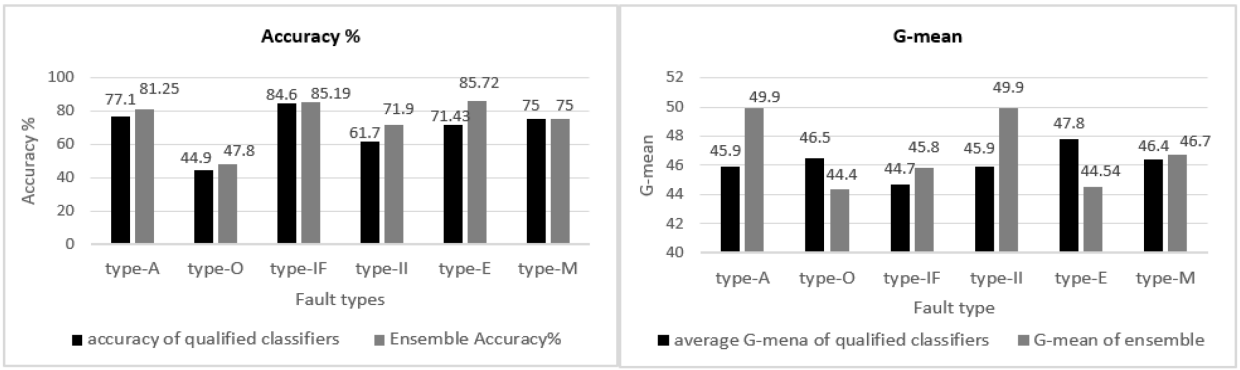

Figure 3.9. Performance comparison of individual classification versus ensemble

Ensemble for fault-type O: Fault-type O ensemble was developed from three classifiers that qualified and namely DT, RF and BNB. Almost all classifiers performed poor against fault-type O. Decision Trees (DT) that performed best among all could only perform with an accuracy of around 52% (refer Figure 3.8 (2)). The ensemble for fault type O had average accuracy of 44.9% for classifiers that qualified, and this accuracy improved to 47.8% using ensemble but G-mean for ensemble type-O did not improve (Figure 3.9) making this ensemble unfit for fault type-O.

Classifiers	MNB	NB	BNB	RF	ET	DT	Logistic	SGD	Linear SVC	NuSVC
G-mean	47.7	43.3	46.7	47.8	46	45.1	43.1	42.4	41.4	41.8

Figure 3.10. G-mean values of individual classifiers

Ensemble for fault-type IF: The ensemble for fault-type IF was made from 5 out of 6 classifiers that qualified. Linear SVC was removed from inclusion into ensemble because of its lower G-mean value among 6 qualified classifiers making the total number of classifiers odd i.e. 5. There was good improvement in G-mean as well as in accuracy using ensemble. This result showed that use of ensemble for fault-type IF was helpful.

Ensemble for fault-type II: The ensemble consisted of 3 out of 4 qualified classifiers and again for this fault-type, linear SVC was removed because of its lower G-mean value. The 3 classifiers for ensemble were again all surprisingly from Bayesian family. This ensemble for fault-type II showed good improvement; as seen in Figure 3.9 that average accuracy (of qualified classifiers) versus ensemble accuracy was improved significantly (62% to 72%) along with improvement in G-mean score from 45.9% to 49.9%.

Ensemble for fault-type E and M: The ensemble for fault-type E was formed with the selection of 3 classifiers out of 4 qualified; namely DT, MNB and RF leaving ET because of its lower G-mean value among all. Although ensemble for fault type-E showed improvement in accuracy but fails to show improvement for G-mean (Figure 3.9). On the other hand, with fault type-M, there were 2 qualified classifiers namely ET and BNB. As the number being even, one classifier had to be removed from consideration leaving behind only one for ensemble. One classifier for ensemble classifies the same way as individual and thus there was no improvement either in accuracy or G-mean.

3.3.3. Discussion of Results

The major goals of this study are to analyze the prediction accuracy of various classifiers against specific fault-types and to analyze the improvement in accuracy/G-mean using ensemble. There were total 109 faults (A-16, O-23, IF-27, II-32, E-7 and M-4).

3.3.3.1. Discussion for RQ-1

This research question was focused on determining performance for each fault-type against the ten chosen classifiers belonging to five different classification families. Our belief was that each classification family will perform differently on reviews as each family executes classification problem through different mathematical functions. For example, Bayesian family includes prior and posterior probabilities and Decision Trees (DT) uses entropy/information-gain theory to classify test data. In rest of this section, major observations are discussed initially for each fault-type followed by discussion on implication of results. So, the major observations that are collected for individual classifiers against each fault-type are listed below:

***Fault-type A*:** Some major observations from results in Section 3.3.3 for ambiguous (A) fault-type are discussed below:

- It was seen that all classifiers from Bayesian family outperformed other classifiers used in this study and classification results of all Bayesian classifiers were very close to each other making them first preference to be used against locating fault-type A.
- Support vector classification family (Linear SVC and NuSVC) performed poorest making them unfit to use against ambiguous fault-types.

***Implications*:** It was observed that the reason for Bayesian classifiers to outperform other classifiers was because of description of ambiguous fault types. Bayesian classifiers works with the use of mathematical functions on probability that output most likely classification class for

the review under consideration. It was also observed that various specific feature terms like 'what', 'when', 'meant by' etc. occurs frequently for ambiguous reviews that generated a high probability for a test review to be classified as fault.

***Fault–type O*:** This was the least correctly-identified fault-type by any classifier used in this study. Automated classification technique could find at most 12 omission (O) faults out of twenty-three (23). Major observations for this fault-type are as follows:

- Only DT classifier could predict highest number of omission faults (12 out of 23).
- Mean accuracy of all the classifiers against omission fault-type was 36.5% (this was the worst performance of classifiers among all fault-types in this study).

***Implications*:** There was fair possibility that omission type faults were responsible for overall degradation of automated classification results. Omission faults (23 in count) for this study that included 109 total true faults were responsible for decrease of more than 10% overall accuracy. I observed our test and training data set for omission type and found that the features that mostly described omission (e.g. 'not' and 'no') are usually removed from analysis using NLTK's stop words.

***Fault-type IF*:** Incorrect Facts (IF) were the only fault-types that were predicted with good accuracy by all classifiers under consideration in this study. Major observations for fault-type IF are as follows:

- Highest accuracy was 89% (24 correctly classified out of 27) while least accuracy was 74%.
- Almost every classification family performed well in predicting correct classification class for test review. The average accuracy of all classifiers fort fault-type IF was found to be 81.5%.

***Implication*:** Incorrect Facts (IF) were the easiest ones to locate because these consisted of many mismatched facts/variable-values across requirements document that were reported by almost every inspector. So, there were mostly similar description of IF fault-type present in both training and test reviews. So, all classifiers performed well during testing because they learned the same mathematical values and features during training.

***Fault-type II*:** Inconsistent Information (II) fault-type were highest in count in true-faults (82 in training and 32 in test set; for more details refer Table 3.10 for fault distribution). The major observations are discussed below:

- Bayesian classifiers outperformed for fault-type II with Bernoulli NB at top with highest accuracy of 72%.
- Mean accuracy (51.3%) of fault-type II was not as high as IF (81.5%) although they report semantically similar faults.

***Implications*:** The behavior of classifiers was cross-checked with inspection reviews collected during study and it was found that majority of reviews that reported fault-type II contained some one-character variable names specific to each requirements document that were important to distinguish fault vs. non-fault but were removed as part of stop words. There was strong implication to use some specific terminology while describing a fault review in NL e.g. 'do not occur' can be written as 'missing' or one-character keywords could be changed to some more meaningful variables such that these are not removed from analysis.

***Fault-type E and M*:** These fault-types being least in number were discussed together in this subsection. There were in total eleven (7-E and 4-M) reviews in test set that represented these fault-types. The best performance for fault-type E was shown by ET, DT, MNB and RF classifiers while for fault-type M, ET and BNB performed highest.

***Implications*:** Fault-type extraneous (E) was reported using few common features across training and testing set. From the results it was seen that DT, ET and RF classifiers performed well and could report significant count. Fault-type M contained very few reviews (4 in total) and it was very hard to perform analysis on few counts. So, some more concrete classification confidence is required to explain classification outcome.

3.3.3.2. Discussion for RQ-2

The purpose of this RQ was to investigate the performance of ensemble classifiers in categorizing a review into fault vs non-fault. The ensemble was created based on top individual classifiers that performed better than other classifiers for each fault-type. The discussion of ensemble approach in this section is presented individually over each fault-types followed by implications of result.

***Ensemble for fault-type A*:** Ensemble for fault-type A consisted of three classifiers namely NB, MNB and BNB, all from Bayesian family (see Table 3.12). Some major observations found (shown in Figure 3.11) are discussed as follows:

- All ensemble classifiers that qualified belonged to Bayesian family.
- Ensemble showed improvement in G-mean.
- The faults were categorized only if 2 out of 3 ensemble classifiers labelled it fault; making the classification confidence of 67% (2 out of 3).

Fault Type	A (16)	O (23)	IF (27)	II (32)	E (7)	M (4)
# of classified faults	13	9	24	23	1	2
Accuracy %	81.25	39.13	88.9	71.9	14.3	50
Ensemble g-mean	49.9					

Figure 3.11. Ensemble for fault type A (NB + MNB + BNB)

***Implications**:* For fault-type A, ensemble with three classifiers was better than one. The classification results (faults/non-faults) were more reliable than individual classifier. Ensemble for fault-type A also performed better in predicting other fault-types i.e. IF and II.

***Ensemble for Fault-type O*:** Ensemble for fault-type O was built over three classifiers namely DT, RF and BNB (Figure 3.12). There were total 23 omission faults and following are some major findings:

- The ensemble performed even worse than the individual classifiers.
- G-mean value did not show any improvement.
- The ensemble could not perform better for any fault-type.

Fault Type	A (16)	O (23)	IF (27)	II (32)	E (7)	M (4)
# of classified faults	9	11	20	13	4	2
Accuracy %	56.25	47.8	74.1	40.63	57.14	50
Ensemble g-mean	44.4					

Figure 3.12. Ensemble for fault type O (DT+RF+BNB)

***Implications**:* Fault-type O were hard to locate through individual classifiers as well as through ensemble.

***Ensemble for Fault-type IF*:** Incorrect Fact (IF) type fault was most accurately and easily located in inspection reviews. There were 27 total faults of type-IF and ensemble using five classifiers was able to locate 23 faults (see Figure 3.13)

- The ensemble predicted good number (85%) of IF faults.
- There had been improvement in G-mean score of ensembles.

Fault Type	A (16)	O (23)	IF (27)	II (32)	E (7)	M (4)
# of classified faults	11	8	23	17	4	2
Accuracy %	68.75	34.8	85.19	53.125	57.14	50
Ensemble g-mean	45.8					

Figure 3.13. Ensemble for fault type IF (DT+MNB+BNB+SGD+NuSVC)

***Implications**:* It was observed that ensemble for fault-type A performed better with one more accurately classified fault than ensemble for fault-type IF. The G-mean value of ensemble A was also better than ensemble IF. Lower G-mean for ensemble IF was reflected because of its under-performance over fault-type A, O and II than ensemble for fault-type A.

***Ensemble for fault-type II*:** It was seen that for fault-type II, Bayesian classifiers qualified for ensemble and performed best among all other classifier (refer Figure 3.14). Ensemble showed improvement in G-mean while accuracy remained same as that of individual classifier (BNB).

Fault Type	A (16)	O (23)	IF (27)	II (32)	E (7)	M (4)
# of classified faults	13	9	24	23	1	2
Accuracy %	81.25	39.13	88.9	71.9	14.3	50
Ensemble g-mean	49.9					

Figure 3.14. Ensemble for fault type II (NB + MNB + BNB)

***Ensemble for fault-type E*:** This fault-type consisted of very few count and ensemble with DT, MNB and RF could label these fault-types more accurately (85.72%). The ensemble for E also performed good over fault-type IF (74%) but was not as accurate as Bayesian family classifiers that gives accuracy of 89% for fault-type IF (see Figure 3.15).

Fault Type	A (16)	O (23)	IF (27)	II (32)	E (7)	M (4)
# of classified faults	9	10	20	12	6	2
Accuracy %	56.25	43.5	74.1	37.5	85.72	50
Ensemble g-mean	44.54					

Figure 3.15. Ensemble for fault type E (DT+MNB+RF)

***Ensemble for Fault-type M*:** Ensemble for M consisted of just one classifier i.e. Bernoulli Naïve Bayes (BNB).

Fault Type	A (16)	O (23)	IF (27)	II (32)	E (7)	M (4)
# of classified faults	12	9	24	23	3	3
Accuracy %	75	39.13	88.89	71.9	42.9	75
Ensemble g-mean	46.7					

Figure 3.16. Ensemble for fault type M (BNB)

The accuracy% of this ensemble was no different than accuracy of individual BNB classifier (see Figure 3.16). The only concern in using one classifier for ensemble M was; *one*, it was not ensemble because to make an ensemble there should be more than one classifier; *two*, it cannot be predicted with classification confidence i.e. no voting is possible. So, it is believed that the fault-type M is better found by individual BNB classifier.

4. SELECTION OF SKILLED INSPECTORS

Although the inspections are effective to find faults early during the software development life cycle, but their effectiveness depends upon the selection of skilled inspectors that can find faults more number of faults. So, this chapter presents details about the selection of skilled inspectors to perform inspection of NL requirements document. This study attempted to find the generalizable characteristics of an inspector using an eye tracker (who can find a greater number of faults).

4.1. Introduction

Leading software companies employ inspections (defined by Fagan [78]) to find and fix faults early and avoid costly re-work later. Previous research estimated that the costs saved by performing early inspections to find faults (especially requirements where they are cheapest to find and fix) vs. testing can vary up to 17:1 work hours [3]. While inspections are useful, its effectiveness is reliant on the selection of skilled inspectors. Researchers and practitioners have tried to understand the background information (experience, education, personality etc.) to predict the performance of individual inspectors but have not been successful [79]. In fact, results at Microsoft and other major software companies showed that most skilled software inspectors had less experience and had non-technical background [79].

Motivated by these findings, Goswami et al., [15] leveraged the research from Psychology to show that *Learning Styles* (LS) can be used to select a team of inspectors. Goswami et al., conducted an industrial empirical study and reported that selecting inspectors with most dissimilar LSs can result in improved fault coverage. This finding was consistent with the results from a large-scale study at Microsoft [79] that managers tend to include inspectors from varied background, especially those with non-computer backgrounds. While LSs seemed

useful in selecting inspection teams, Goswami et al. [2], [3], [15], were not able to find common LSs that were positively correlated to individual inspection performance across studies. One of the reasons for this was that SRS documents are generally developed in NL and are not tailored to LSs of specific readers. One of the major results from these studies was that individual inspector (even within same LS category) exhibit different reading patterns (RPs) depending on the type of SRS being inspected which in turn impacts their ability to report faults. While LS is an abstract model of capturing the RPs of inspectors, more objective means of capturing RPs would help project managers identify skilled inspectors. It is believed that inspectors' RPs are generalizable across SRS documents.

Additionally, past research [77] has identified that software organizations need inspectors that can detect specific fault types (e.g., Ambiguous - A, Inconsistent Information - II, Omission - O, Incorrect Fact - IF) at a higher rate. Therefore, this research tried to characterize the RPs of inspectors and their capability at detecting various requirement fault types.

To characterize RPs, an eye-tracking apparatus was used in a controlled environment wherein software engineers (with industry experience) reviewed an industrial strength NL requirement document and reported faults. This study collected several metrics to examine the RPs of inspectors (e.g. eye movements), their cognitive processing and their fault detection abilities across different areas of SRS (see Figure 4.1). Following are eye-tracking metrics that were collected and are being analyzed in this research:

- *Fixation*: is a point where eyes are relatively stationary, and an individual is taking in the information.
- *Saccade*: Quick eye movement between fixations.

- *Scan paths*: are a complete saccade-fixation-saccade sequence and interconnecting saccades.
- *Gaze*: is the sum of fixations' duration in an area. They are also known as "dwell", "fixation cluster", or "fixation cycle".
- *The region of interest (ROI)*: is an analysis method where eye movements that fall under certain area are evaluated.

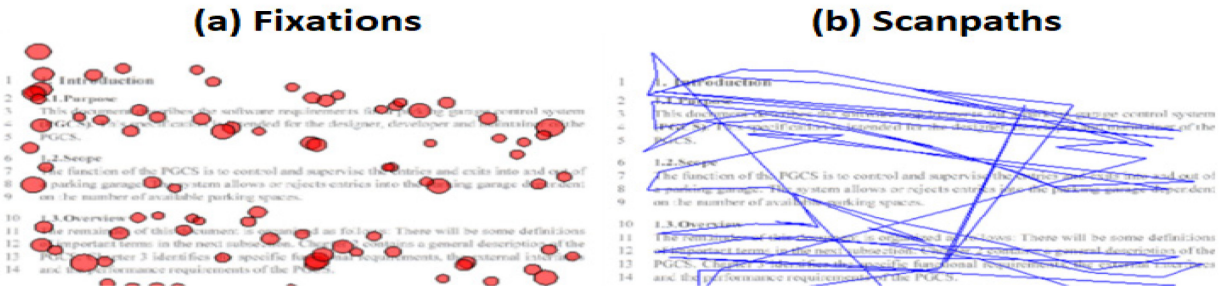

Figure 4.1. Sample reading patterns showing fixations and scan paths

Next, to better analyze the RPs of inspectors and predict their inspection effectiveness, ML algorithms (principal components and classification) were employed. The choice of ML algorithms was motivated by previous research [77], where it was found that ML algorithms have varying prediction accuracy for various fault-types. This study validated ML algorithms w.r.t RPs for each fault-type and applied principal components analysis to be able to best predict the capability of an individual inspector to report various faults-types. Open source commercial ML tool (WEKA - Waikato Environment for Knowledge Analysis) was used for implementing ML algorithms. This study reported results regarding the common RPs and most effective ML algorithm that can assist project managers select the most effective inspectors.

4.2. Proposed Approach and Experiment Design

There were two RQs identified for this study regarding identification of RPs of a skilled inspector and these RQs are as follows.

RQ-1: What reading patterns (RPs) of an inspector are most effective at reporting various fault-types?

RQ-2: What type of ML algorithms can best predict inspectors' effectiveness for various fault-types?

4.2.1. Variables Used

Various independent and dependent variables used in this study are discussed below.

Type of classifiers: The classifiers from 5 different classification families (i.e. Bayesian, Support Vector, Ensemble, Trees and Lazy Learners) were chosen based on their applicability and performance as reported by prior inspection studies in literature [77].

Fault types: The reviews were divided into 6 fault-categories (i.e. ambiguous (A), inconsistent information (II), Incorrect fact (IF), Omission (O) and Extraneous (E) and Miscellaneous (M)). The fault concentration per fault category out of 35 seeded faults was A:4, II:12, IF:2, O:13, E:3 and M:1. The fault-types E and M were excluded from the study because these fault-types were only detected by at most 2 inspectors making this fault-type extremely imbalanced (i.e., likelihood of imbalanced distribution of instances in a binary classification problem [77]). Sampling E and M fault-types did not produce good samples and were not included for analysis.

Attribute evaluators: Eye-tracking equipment recorded 21 RP attributes for each inspector. To select prominent attributes (features or principal components) that could accurately characterize the RPs of inspectors for all fault-types, three algorithms were selected and evaluated (Subset evaluation, information gain and wrapper method) based on literature [26], [80]. Table 4.1 provides a final list of fault-types, classifiers and attribute evaluators for all fault types. More details about classifiers and attribute selection appear later in this section.

Table 4.1. List of various variables exploited for each fault-type

Fault types	**Classifiers**	**Attribute Evaluators**
Type A, Type IF, Type II, Type O	NB, RF, Lazy Learner, MNB, Ensemble (AdaBoost, Voting, Bagging)	Classifier subset evaluator, Information Gain, Wrapper Subset Evaluation

Training and test set: The eye tracking data used in this paper was generated from 39 participants (majority of whom had around 2 years of SE experience) using PGCS document. This data was split into 70% for training and 30% for testing. Post inspection, final class attribute was labelled with binary class labels ('yes' or 'no' to denote the capability of an inspector to report a fault or not). If an inspector was able to report faults at least greater than or equal to mean of all the faults found within each fault-type, then that inspector was labeled with 'yes' in final class category. Final class label was required to evaluate prediction results of ML algorithms. For example, the mean number of faults found by each inspector for fault-type II was 1.4; any inspector reporting 2 or more true II faults are labeled capable of reporting fault-type II (i.e. has a value 'yes' in final class attribute).

Validation method: Throughout our experiment, 10-folds cross-validation method was used as it is most commonly used method to measure model performance. All the classifiers and ML approaches evaluated/used in this study were tested with default parameters, unless specified otherwise.

Dependent variables: The following variables were collected to measure the effect of independent variables and acted as evaluation metrics.

- *Recall*: It is the proportion of true positives that are correctly identified (i.e. sensitivity).

- *Precision*: It is the fraction of relevant instances among the retrieved instances w.r.t true positives.
- *F-measure*: It is a measure of test's accuracy and it considers both precision and recall.
- *ROC (Receiver Operating Characteristics)*: It is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. It is created by plotting true-positive rate and false-positive rate. The observations and results have been derived from F-measure and ROC curve.

4.3. Experiment Procedure

The experiment procedure consisted of following six steps as shown in Figure 4.2. The description of each step is presented briefly in this section.

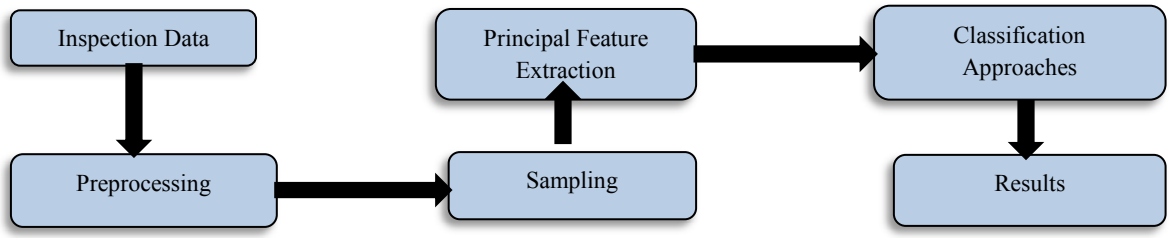

Figure 4.2. Overall experiment procedure

Inspection data: The inspection data generated from the eye-tracking study [1] consisted of 21 different attributes (Features). The SRS was divided into three sections (introduction, general description and functional description) and the time was evaluated on these sections separately to better understand the impact of RPs. The description about 21 attributes is presented in Table 4.2.

Preprocessing: Some features like '*linear saccade per page and time taken*' etc. consisted of suffixes (e.g. % symbol, minutes) that required removal before being processed. The binary class attribute '*actual faults*' contained total number of true-faults found by an inspector.

This attribute was manually processed to represents the final class label (*'yes'* or *'no'*) for all the instances. The attributes *id, total fixation at ROI, total time duration at ROI, total faults, efficiency and false positives* were removed from the analysis because this experiment aimed at providing automated and generalizable attributes for determining the capability of an inspector post-inspection. These six attributes were manually calculated post-inspection, so these did not contribute towards general adaptability of features to other SRS documents where information about seeded faults (i.e., ROI or regions with seeded faults) was not known. The final dataset had 15 total features including final class attribute.

Table 4.2. Various eye tracking attributes recorded

Categories	**Attributes (total 21)**
General	id (# assigned as identifier)
Fixation data per page	Average fixation time, fixations per page, time spent per page, Linear saccades per page
Fixation data at ROI (region of interest)	Total fixation at ROI, total time duration at ROI
# of time inspector went back to search an information	Total lookups in introduction, total lookups in general description, total lookups in functional, total # of searches
Time taken to search an information	Time spent on reading introduction, time spent on reading description, time spent on reading functional requirements, total search time
Inspection performance	Total faults in SRS, false-positives by inspector, faults reported by inspector, actual faults (effectiveness), total time taken, efficiency (fault rate)

Sampling: The data collected for PGCS document consisted of an uneven number of true-positive and false-positive instances; showing the class-imbalance problem. So, one of the sampling techniques i.e. SMOTE (Synthetic Minority Oversampling Technique) was applied with WEKA over the data to artificially generate minority class instances [76]. The data was then shuffled randomly (using Randomize filter in WEKA) to select unbiased training and test sets during validation stage of the experiment.

Principal feature extraction: Three types of techniques were used to evaluate best performing features for the given set of data. These techniques were based on 'Classifier s*ubset evaluation', 'Information gain*', and '*Wrapper methods*'. More details on these techniques can be found in [26], [80]. The selection of these techniques was based on their performance to rank principal attributes over 14 well-known benchmark datasets for classification and these selectors are well applicable to binary class problems.

Classification approaches: The classifiers from five different classification families (discussed earlier in chapter 3) were trained and these classifiers were *naïve Bayes* (NB), *Multinomial NB (MNB)*, *Decision Trees (DT)*, *Random Forest (RF), Lazy Network (Locally weighted learning), Stochastic Gradient Descent (SGD), and Ensemble (AdaBoost, Bagging, and Voting)*.

Results: The performance evaluation was performed by collecting various metrics as described above. The results are shown w.r.t F-measure and ROC; because these were few metrics that were considered prominent among standard benchmarks to measure classification performance. The next section presents the results and discussion regarding this experiment.

4.4. Results and Discussion

This section presents the result on best classification approaches and prominent reading patterns that can predict inspector's ability to report a specific fault-type. The prominent results are shown in Table 4.3 for all fault-type using the area under ROC curve as a prominent metric to evaluate classification performance based on features selected by varying classifiers over three attribute evaluation methods (see Table 4.1). The best performing classifiers, prominent features and evaluation results (%age of ROC) are highlighted with bold/underline in Table 4.3. The

percentage of ROC was used as performance evaluator for all fault-types. The results organized around the two RQs are as follows:

4.4.1. Results for RQ-1

This research question was aimed at finding which RPs (collected during the eye-tracking study) can help determine inspector's ability at finding a specific fault-type. The experiment was evaluated using prominent features extracted in four ways to train a classifier; out of which, three attribute evaluator methods were used to extract features (See Table 4.1 for attribute evaluators) and the fourth method considered all available features. The key result findings (from Table 4.3) are shown below:

- *Most prominent feature set*: The result showed that there were few prominent reading pattern features (out of 15) that were commonly ranked higher across all attribute evaluators used to predict inspection effectiveness for all fault-types. These features included *average fixation time, linear saccades per page, fixations per page, time spent per page, and total lookups in functional*. Using these subsets of features resulted in an improved prediction accuracy.
- *Other prominent features*: In addition to above features, metrics related to time spent fixating or searching on different parts of SRS (or fixating or searching/lookups) strengthened the prediction results. Specifically, *average fixation time, and total number of search time* were most informative features. This was important result because companies rely on selecting inspectors that find faults faster to enable maximum cost savings. Evaluating the reading patterns with respect to the time spent can help characterize inspectors' performance better as demonstrated in this research.

- *Subset evaluators*: Out of four different evaluator methods used in this research, Wrapper Subset Evaluation resulted in largest improvement in prediction accuracy. The percentage of AUC-ROC gain (shown in last column in Table 4.3) was a measure of improvement in prediction accuracy for fault-types. Based on these gains, accuracy at predicting inspectors for fault-type A is 81%, IF is 94%, II is 88%, O is 79%. These prediction results were noteworthy especially when comparing against similar research in other domains (e.g., 80% in dyslexia study and 74% in image-feature study described earlier).

Implications: Using the findings from this study, most prominent features either belonged to inspectors' reading patterns (*average fixation time, linear saccades per page, fixations per page, total lookups in functional*) or using the timing information (i.e. *time spent per page, total search time).* These features provided insights into inspectors' ability to comprehend, analyze, and detect problems in SRS. These prominent features, when used by ML algorithms predict inspectors' abilities to find different fault types in an SRS.

4.4.2. Results for RQ-2

This question aimed at finding the most suitable classifiers that could predict the effectiveness of an inspector at reporting a fault-type. The result and discussion are based on the data presented in Table 4.3. The evaluation metrics Precision, Recall, and F-measure are represented using P, R, and F respectively in Table 4.3. The ROC curve is only sketched (Figure 4.3) for the most prominent fault-type (IF) and most applicable evaluation method (i.e., wrapper method using RF classifier). The result in Figure 4.3 shows that the largest gain (94% when using wrapper method vs. 80% without) for AUC-ROC metric for fault-type-IF. Readers can reference Table 4.3 that shows accuracy gains across all classifiers (i.e. random forest, Lazy Learner, Naive Bayes, Voting, AdaBoost, and Bagging); all evaluation methods (information

gain, wrapper subset, and classifier subset) and for all fault-type. Some of the major observations from Table 4.3 are discussed below:

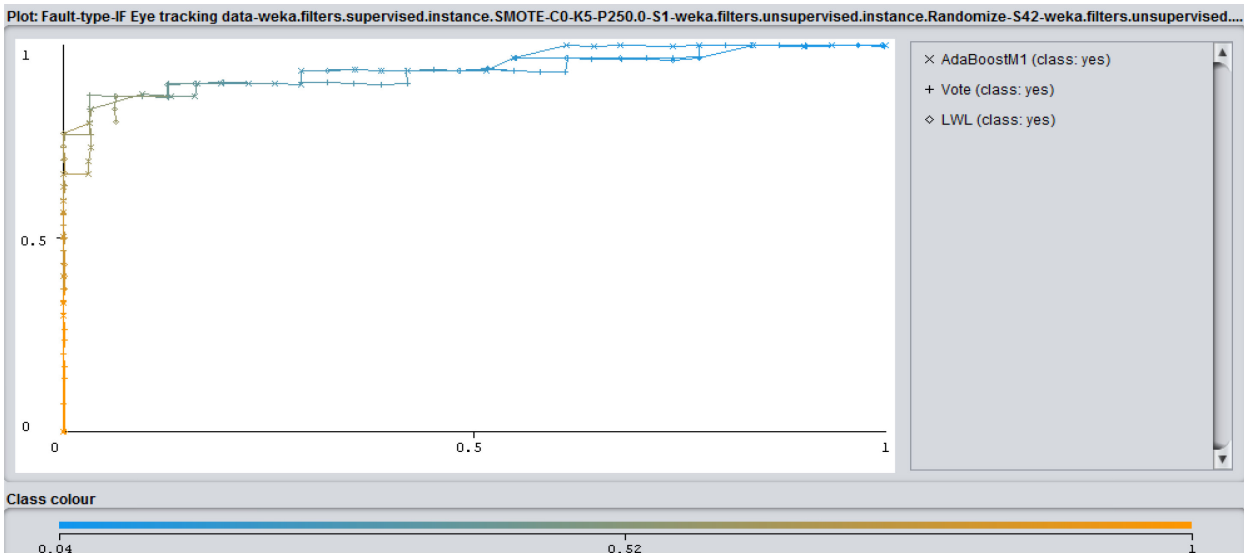

Figure 4.3. ROC curve of fault type-IF for wrapper subset method using RF

- Selection of best performing inspectors were most accurately predicted by *Random Forest (RF)* classifier when used to create ensemble or voting method for almost every fault-type. RF when used with ensemble methods resulted in accuracy between 80% and 94%. The companies can rely on these results to guide the selection of most skilled inspectors.
- In terms of accuracy for fault-types, the largest prediction accuracy values ranged from 94% for fault-type IF, 88% for II, 81% for A, and 79% for O. Mostly, this accuracy was obtained when selected features were related to RPs (i.e., fixations and saccades). Figure 4.3 shows the ROC curve of various classification types (AdaBoost, Voting and Ensemble) using RF as classifier (Table 4.3 shows more details on performance of classification types when used with various classifiers for all fault-types).

Table 4.3. Experiment results for all considered fault-types

Evaluator Method – Fault type	Classification Type	Selected Features	TP rate	FP rate	P	R	F	AUC ROC %
None	RF	All	80.8	34.6	70	80.8	75	75.5
Classifier Subset - A	Ensemble (Bagging with RF)	**Avg. fixation time, Linear saccades per page, Total lookups in general description, Total lookups in functional**, **Total # of searches, Total search time**, Faults reported	84.6	34.6	71	84.6	77.2	78.7
	Lazy Learner with RF	*-Same as immediate row above-*	80.8	30.8	72.4	80.8	76.4	**80.7**
Information Gain - A	**Lazy Learner with RF**	**Total time taken**, **Time spent per page, Fixation per page, Total lookups in intro, Linear saccade per page**, Faults reported, **Total lookups in general description**	80.8	30.8	72.4	80.8	76.4	**80.7**
	Ensemble (AdaBoost with RF)	*-Same as immediate row above-*	80.8	34.6	70	80.8	75	77.6
Wrapper Subset - A	**Ensemble (Bagging with RF)**	**Linear saccades per page**, **Total lookups in general description**, Faults reported	92.3	42.3	68.6	92.3	78.7	**80.2**
	Lazy Learner with NB	*-Same as immediate row above-*	80.8	38.5	67.7	80.8	73.7	77.4
None	RF	All	46.7	3.2	93.3	46.7	62.2	79.4
Classifier Subset - IF	Ensemble (AdaBoost with RF)	**Average fixation time**, **Total lookups in introduction**	70	9.7	87.5	70	77.8	83.4
Information Gain - IF	RF	**Total lookups in intro, Total lookups in functional**, **Time per page**, **Time reading intro**, **avg. fixation time, fixation per page**	83.3	6.5	92.6	83.3	87.7	92.6
	Ensemble (Bagging with RF)	*-Same as immediate row above-*	83.3	3.2	96.2	83.3	89.3	92.0
	Voting (RF, Bagging with RF, and AdaBoost with RF)	*-Same as immediate row above-*	66.7	3.2	95.2	66.7	78.4	91.9
Wrapper Subset - IF	**Ensemble (AdaBoost with RF)**	**Average fixation time, Time spent per page**, **Total lookups in intro, Total lookups in functional, time reading description, total time taken**	83.3	3.2	96.2	83.3	89.3	**94**
	Voting (RF, Bagging with RF, and AdaBoost with RF)	*-Same as immediate row above-*	93.2	3.2	96	80	87.3	**93.2**
	Lazy Learner with RF	*-Same as immediate row above-*	86.7	6.5	92.9	86.7	89.7	**93.5**
None	RF	All	70.8	34.8	68	70.8	69.4	79.3
Classifier Subset - II	Voting (RF, Bagging with RF, and AdaBoost with RF)	**Average fixation time, Fixation per page, Time spent per page, Linear saccade per page**, **Total lookups in functional, Total # of searches, Time spent reading description**, **Total search Time**	79.2	34.8	70.4	79.2	74.5	79.7
	Ensemble (AdaBoost with RF)	*-Same as immediate row above-*	79.2	34.8	70.4	79.2	74.5	79.4
Information Gain- II	Ensemble (AdaBoost with RF)	**Total lookups in functional**, **Time spent per page**, **Total search time**, **Fixation per page, Linear saccade per page**, **Time spent reading description**, **Total # of searches, Avg. fixation time**	79.2	34.8	70.4	79.2	74.5	79.4
	Voting (RF, Bagging with RF, and AdaBoost with RF)	*-Same as immediate row above-*	79.2	34.8	70.4	79.2	74.5	79.7
Wrapper Subset - II	**RF**	**Total lookups in Functional**, **Time spent reading intro**	66.7	13	84.2	66.7	74.4	**86.7**
	Voting (RF, Bagging with RF, and AdaBoost with RF)	*-Same as immediate row above-*	70.8	13	85	70.8	77.3	**85.5**
	Lazy Learner with RF	*-Same as immediate row above-*	75	17.4	81.8	75	78.3	**87.6**
None	RF	All	69.6	29.2	69.6	69.6	69.6	69.7
Classifier Subset - O	Lazy Learner with RF	**Average Fixation Time, Linear Saccade Per Page**, **Time spent reading intro**	52.2	45.8	52.2	52.2	52.2	55.9
Information Gain - O	Lazy Learner with RF	**Total time taken, Linear saccade per page**, **Total lookups in intro, Time spent per page**, Faults reported, **Fixations per page**	73.9	37.5	65.4	73.9	69.4	66.3
Wrapper Subset - O	**Voting (RF, Bagging with RF, and AdaBoost with RF)**	**Average fixation time, Time spent per page**, **Total lookups in functional, Total search time, Total time taken**	78.3	16.7	81.8	78.3	80	**79.3**
	Ensemble (AdaBoost with RF)	*-Same as immediate row above-*	73.9	20.8	77.3	73.9	75.6	**78.5**

Implications: These results also showed that random forest (either alone) or when used with ensemble or voting methods can strengthen the characteristics for selecting inspectors that would identify larger number of faults. RF classifier uses multiple decision trees to fit the data (i.e. train) to classify test data. The majority features from eye-tracking data being continuous in nature were best split in intervals by the underlying decision trees. This resulted in strongly learned RF classifier that outperformed other classifiers over test data. It was also observed that the prediction is accurate and generalizable when features contains RPs of an inspector. Inspectors that tend to fixate more, exhibit linear saccades, performed more searches were more likely to identify more faults.

5. KEY PHRASE EXTRACTION FROM FAULT LOGS

This chapter concentrates on proposing an automated approach to identify key phrases from fault logs (that points to faults) and then mapping those keyphrases to SRS document (chapter 6) to identify fault-prone requirements that needs a fix (goal-2). Automation using our proposed approach saves time and provide additional decision-making support (based on mathematical foundations) to requirements authors during post-inspection fixation process.

5.1. Introduction

Our previous work (see Figure 5.1) was focused at automated validation of requirement fault logs (i.e., identifying true-positives (TPs) vs false-positives (FPs)) [72], [76], [77]. The proposed work in this chapter is motivated by positive results from our prior work and is focused on automating the identification of "problematic areas" in an SRS based on the fault logs. To achieve this automation, several ML *KPE* algorithms were employed to develop KESRI (Keyphrase Extraction in Software Requirement Inspections) approach that would automatically extract keyphrases from fault logs and would reduce manual analysis of each fault log.

The KESRI approach was validated against the keyphrases extracted by manual analysis (details appear later). To implement and validate KESRI, an inspection study at NDSU was executed that collected fault logs which were validated for true faults (using prior work). The true faults were next fed to the KESRI approach to extract keyphrases (corresponding to problematic parts of SRS) that can then be prioritized during fault fixation.

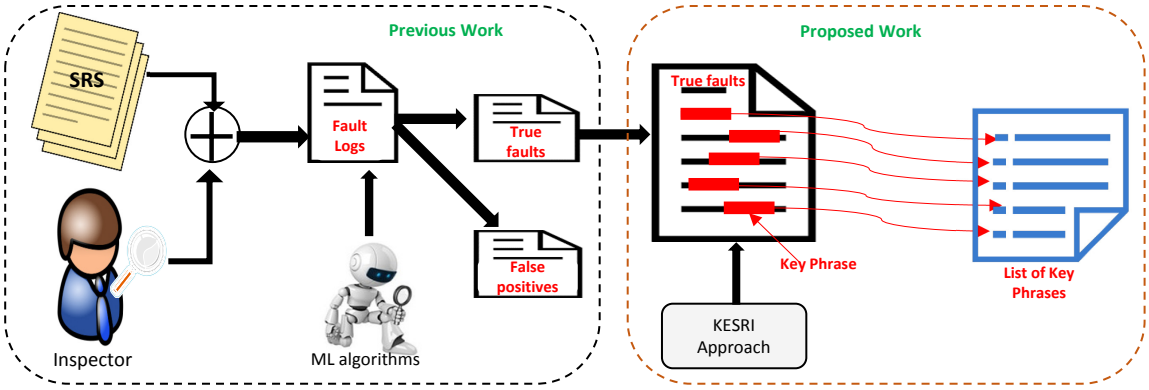

Figure 5.1. The KESRI approach

5.1.1. Proposed Approach

Our proposed approach adapted existing KPE algorithms (that have been applied on NL text) to NL software requirements. KESRI can be used in unsupervised (without SRS) and supervised learning (with SRS) mode and would extract key information from NL fault logs. This section presents the working example of how KESRI would extracts keyphrases from fault logs (reported during an inspection) to highlight problematic areas in an SRS (to assist post-inspection activities). Various steps involved are shown in Figure 5.2 and a sample fault log (representative of nature of fault logs) listed below is used to explain the working of our approach.

"The initial value of r was defined to be 10000. But now it is defined as 1000. Inconsistent value of r."

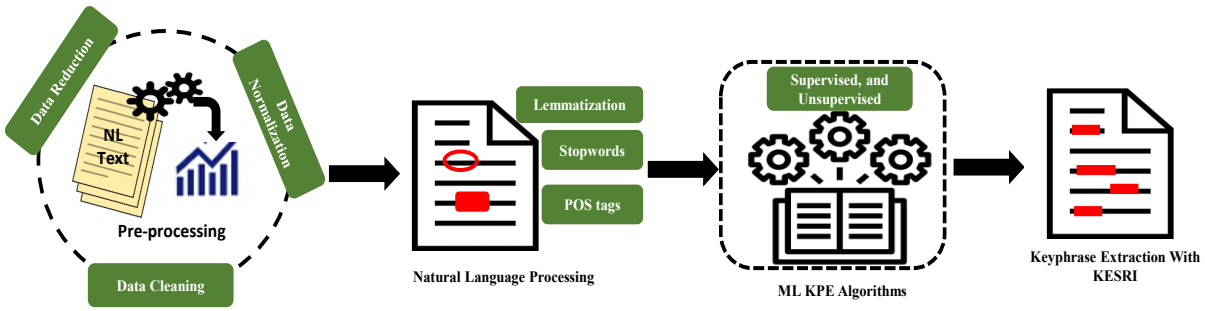

Figure 5.2. Steps involved in KESRI approach

5.1.1.1. Preprocessing

NL fault log text inputted to KESRI approach needs to be normalized to maintain consistency with SRS text. For this purpose, both inputs (SRS and fault logs) used by KESRI are preprocessed to filter non-useful text (e.g., removal of stop words, lemmatization, spelling corrections). All the NL text was converted to lowercase and all the acronyms in a given NL text were converted to their full abbreviations to maintain consistency. Any variable or expression listed as a single character was converted to a standard format to avoid them from being removed as part of stop words removal or as part of any parameter setting (e.g., removal of words that are less than 3 characters in length). The NL text can have hyphenated words and the preprocessing step would de-hyphenate these words to avoid miscalculation of frequency counts. As an example, at the end of this step, the sample fault log considered to explain working of KESRI is preprocessed and is shown below.

"the initial value rvariable defined 10000 but defined 1000 inconsistent value rvariable"

Preprocessing step on original sample fault removed stop words (e.g., 'of', 'was', '.'), the text is converted to lowercase, and a single character variable 'r' is converted to a standard variable form i.e. rvariable. Similarly, the complete SRS document was preprocessed (in supervised learning) to normalize its text to reduce any potential bias during KPE from fault logs as well as during model training with the SRS.

5.1.1.2. Natural language processing and parameter tuning

Application of KESRI required certain ML algorithm parameters to be tuned (e.g., POS tag combinations, window of words to consider, inclusion or exclusion of stop words, # of N-grams to consider etc.) to ensure that keyphrases extracted from fault logs were consistent with software requirements text. The parameter tuning was guided based on our previous findings and

literature on application of ML on software requirements. For example, our prior work [72], [76], [77] reported that POS tag combination of nouns and adjectives are most applicable when validating requirement fault logs using ML algorithms. Similarly, based on literature findings, most keyphrase extraction algorithms used N-grams between length 3 and 5. After stop words removal (during preprocessing), the length of fault logs was reduced, and N-grams ($1 \leq N \leq 3$) were more likely to generate prominent keyphrases. More details about additional parameters appears in Table 5.1.

Table 5.1. Parameter tuning of KESRI approach

Algorithm	**Stop words removal**	**POS reqd.**	**Document frequency weighting**	**N-gram**	**Other/Remarks**
TFIDF	Yes	No	SRS doc. Freq.	3	3-gram is used because of Stopwords removal
KPMiner	Yes	No	SRS doc. Freq.	1-5	Least allowable seen freq. is set to 3 and cutoff is set to 200, parameter boosting is set to 3.
YAKE	Yes	No	Window parameter	1-3	Window size is set to 2 to compute right/left context, and redundant keyphrases are removed using Levenshtein distance with threshold of 0.8.
Text Rank	No	Yes	Weighting is by selecting top 33% words as candidate	No	Words in a key phrase (window) is set to 3, the top-percent is set to 0.33 to generate candidate keyphrases from top 33% highest ranked words. POS tags used are noun, proper nouns, and adjectives.
Single Rank	No	Yes	# of co-occurrences of words using window parameter	No	POS tags used are nouns, proper nouns, and adjectives. Window parameter is set to 10
Topic Rank	Yes	Yes	Weights assigned using random walk	No	The longest sequence of nouns, proper nouns, and adjectives are used to create candidate phrases. The clustering of terms is calculated using average linkage graph strategy and setting the threshold parameter to 0.74.
Topical Page Rank	Yes	Yes	Single topical page rank in a given window	No	The window is set to 10 and the candidate keyphrases are selected using a regular expression “zero or more adjectives followed by one or more nouns”. The LDA model is used to calculate word topics.
Position Rank	No	Yes	Using sum of word’s score computed using random walk.	3	Window size is 10 and the POS tags nouns, proper nouns and adjectives are used. Uses grammar rule to only consider those phrases that have zero or more adjectives followed by one or more nouns.
Multipartite Rank	Yes	Yes	Weights calculated using random walk.	No	The longest sequence of nouns, proper nouns, and adjectives is selected for candidate phrases. The weights are controlled by setting the alpha value to 1.1, threshold to 0.74, and by using average linking method.
KEA	Yes	No	SRS doc. Freq.	1-3	SRS document freq. is used, and the default model file defined in PKE package is used. The default package uses Naïve Bayes to train the model.
Wingus	No	Yes	SRS doc. Freq.	No	Only noun phrases are used to create candidate selection. The default model is used that is trained using Naïve Bayes.
TOPIA	No	No	SRS doc. Freq.	No	We set the number of words in a phrase be between 3 and 5. The algorithm works with some pre-defined set of rules to determine key phrases. We used the default rules for key extraction.
RAKE	Yes	No	SRS doc. Freq.	No	We set the number of words in a phrase between 3 and 5.

ML algorithms: KESRI utilized ML KPE algorithms that belonged to both supervised and unsupervised learning domain. When used in unsupervised mode, KESRI extracted key phrases from fault logs alone whereas when used in supervised mode, KESRI used fault logs, SRS and training models trained on specific features (e.g., frequency of terms, POS tag or combination of POS tags).

Keyphrase extraction: The KESRI approach extracted keyphrases using KPE algorithms which were then compared against the manually extracted keyphrases (from an expert and discussed in the validation section of this chapter). This subsection only discusses the algorithmic application of KPE.

KPE algorithms (listed in Table 5.1) were implemented to extract keyphrases of any length N (where $1 \leq N \leq \infty$), but in this study N was set to 1 (i.e., 1-word). The details of individual algorithms are beyond the scope of this research and some more details discussed in chapter 2. To explain the working of KESRI approach, a concise and high-level overview of working of one algorithm per category is presented in Table 5.2 using the same sample fault (after preprocessing). The keyphrases were extracted and ranked (in decreasing order) by KPE algorithms based on the highest value of weight that a phrase carried (i.e., *the numerical value that signifies importance of a term in a text*). Every KPE algorithm family used different weight assigning principle guided by their theoretical underpinnings. For example, few KPE algorithms used SRS document frequencies (i.e., TFIDF), and few based on co-occurrences of certain combination of POS tags like nouns and adjectives etc., to assign weights to keyphrases (refer Table 5.1).

The KESRI approach enabled the requirements authors to select top ranked X number of phrases based on N-word length (where, $1 \leq X \leq \infty$, and $1 \leq N \leq \infty$). However, the KESRI approach

can theoretically retrieve infinite # of top phrases and N-word phrases, but there is usually an upper bound on the value of X and N, and varies based on the length of the document that is being used to extract keyphrases (e.g., $1 \leq X \leq 350$, and $1 \leq N \leq 5$ for fault logs in our validation study). Our evaluation results showed that the higher value of N results in very low X number of phrases. The example output of keyphrases from our sample fault example (with X=5, and N=2) is shown in the Table 5.2.

Table 5.2. Working example of KESRI for selected family of KPE algorithms

	Unsupervised Graph Based	**Unsupervised Statistical Based**	**Supervised feature based**
Algorithm	Text Rank	YAKE	KEA
Overview	Text Rank Algorithm constructs a graph of co-occurring combination of POS tag (e.g., only nouns and adjectives within a window of 2 consecutive words in a text) that are added to a graph with an edge denoting co-occurrence. If a word appears more than once, then that word has more than one edge to denote co-occurrence with multiple words. The words are ranked based on the # of edges and the importance of its co-occurring words.	YAKE extract phrases based on characteristics such as Casing (case aspect of a word), position (assuming that relevant words occur at the beginning), word frequency, relatedness (words occurring to the left/right of a word), and word difference (i.e., how often a candidate word appears in different sentences). The ranking of key phrases is based on heuristics, and TFs.	KEA identifies key phrases using Naive Bayes supervised training model. The SRS document in our case is used to train the model. KEA identifies key phrases using lexical methods to calculate feature scores for its candidates. The test set (i.e. fault logs) is tested against the trained model to extract most important phrases.
Key phrases extracted	Initial value	Value rvariable, 1000 inconsistent, initial value, inconsistent value, defined 1000	Value rvariable, initial value, rvariable defined, defined 1000, 1000 inconsistent

5.2. Experiment Design

This section present details regarding the application and evaluation of KESRI approach on a sample SRS document that was reviewed by a skilled set of inspectors. This section presents the design of inspection study that was conducted at NDSU along with variables that were exploited, and the data collected during the inspection study. This section also present RQs that were investigated and the metrics that were used to validate the KSERI approach.

5.2.1. Research Questions

Two major research questions were formulated and investigated in this study, and these are presented as following.

RQ-1: How effectively does KESRI approach extracts keyphrases from fault logs?

RQ-2: What family of keyphrase extraction algorithms are most applicable to requirements fault logs?

5.2.2. Algorithms Used

The independent variables included different categories of KPE algorithms (listed in Table 5.3) and the length of key phrases extracted from each algorithm *(i.e., **1-words).*** The dependent variables included precision, recall and F-measure of these independent variables (Table 5.4).

Table 5.3. Key extraction algorithm categories

Category	**Sub-category**	**Algorithms**
Unsupervised	Statistical	KPMiner, YAKE, RAKE, TOPIA
	Graph Based	TextRank, Single Rank, Topic rank, Topical page rank, Position rank, and Multipartite rank
Supervised	Feature based	KEA, and WINGUS

5.3. Experiment Procedure

This section presents details about the inspection study (that was conducted to collect fault logs), validation metrics and algorithmic details (adapting KPE algorithms, creating training sets), and criterion used to evaluate the performance of KPE algorithms.

Table 5.4. Evaluation metrics for KESRI approach

Metrics	**Definition**	**Formulae**
Precision (P)	Denotes the # of selected relevant phrases	TP/(TP+FP)
Recall (R)	Denotes the # of selected relevant phrases	TP/ (TP+FN)
F-measure (F)	It is the harmonic mean of both precision and recall	2 (P*R)/(P+R)

5.3.1. Variables Used

This section discusses various variables exploited in this study along with data collection procedures and metrics used.

***Inspection study*:** The inspection study was conducted at NDSU where inspectors were trained on using fault checklist (FC) technique to review and report faults in an externally developed SRS document and it was validated using metrics shown in Table 5.4.

***SRS artifact*:** The parking garage control system (PGCS) requirements document was inspected during the study. PGCS had been used in prior inspection studies including our prior work [72], [76], [77]. PGCS requirements document is 14 pages long and is seeded with 35 defects across 61 functional and non-functional requirements.

***Inspectors and data collected*:** A total of 41 inspectors inspected the PGCS document. The inspectors were computer science students (both graduates and undergraduates) and had at least 2 years of work experience. Students were trained on how to use FC and they subsequently used FC to inspect the PGCS document. A total of 201 true fault-logs (i.e., true faults) were reported during inspections and were input to KESRI approach.

5.3.2. Application of KESRI on PGCS fault logs

This section presents more details about implementation of the KESRI approach using python 3.7 library version along with API from different packages like NetworkX (for graph models), Gensim (for semantic similarity approaches like LSI and LDA), Jellyfish (for syntactic similarity), NLTK (for basic NLP operations like stop words removal, POS tag generation), PKE library (to implement various supervised and unsupervised keyphrase extraction algorithms), TOPIA (term extraction), and RAKE (key phrase extraction). This subsection provides details on

how KESRI was adapted to extract keyphrases from PGCS fault logs. This subsection is centered on the steps involved in extracting keyphrases from SRS fault logs.

***Step 1 - input - preprocessing of SRS and fault logs*:** The preprocessing of fault logs and SRS is performed by converting all the acronyms to their full abbreviations to maintain consistency across the text and to avoid misleading frequency counts (e.g., to avoid separate frequency calculation of the acronym 'SRS' and 'software requirements specification'). Next, all mathematical expressions (e.g. 'a>0') were manually spaced to treat them as individual tokens. This enabled KESRI approach to generate frequency count for 'a', '>', and '0' individually. The entire SRS and fault logs were preprocessed to maintain consistency, such as, '*xvariable*', where 'x' was substituted with name of single character variable (e.g., 'a' in a>0 was converted to *avariable*>0). This preprocessing step would ensure that no relevant phrases are lost during the extraction which is a common drawback of some of the existing KPE algorithms that tend to exclude single length characters.

Additionally, these single character variables across different functionalities in an SRS and if these are lost during the KPE implementation, then I would lose the ability to map these phrases to individual requirements (a future step when mapping phrases to requirements that needs fixation). For example, the variable 'r' referred in an SRS for '*hourly_rate*' in parking garage scenario and is also referred in many other functionalities such as, *bill_generation*, *weekday_rate, and weekend_rate (1.5 times hourly rate on weekday)*. So, these single character variables should be extracted (e.g., using Xvariable format) to enable mapping them to faulty requirements in an SRS.

***Step 2 - output - extraction of keyphrases*:** The keyphrases of length up to N-words (with $1 \leq N \leq 3$) can be extracted with KESRI approach and can be used in supervised and unsupervised

mode. Supervised KPE algorithms require term frequencies to train the model prior to keyphrase extraction. The term frequency count (in an SRS) were used to assign the weights (i.e. a numerical value that denotes the importance of each term) which in turn were used to calculate the importance of a phrase extracted from the fault logs (details about weighting discussed in Table 5.1). The disintegration of text (both SRS and fault logs) into N-words for frequency counts was done using N-grams (discussed in chapter 2) followed by weight assignment to rank keyphrases.

Step 3 - validation of KESRI output: To facilitate validation of KESRI, the manual KPE was performed by the domain expert to obtain the most relevant keyphrases from preprocessed fault logs but without ML assistance. This manual step identified all relevant terms from the fault logs. For example, in the sample preprocessed fault log considered (in section 5.1), the most relevant terms extracted from domain expert included "*inconsistent", "value", and rvariable*". The manual extraction included a constraint that these terms should also appear in an SRS. The KPE algorithms are currently not able to implement this feature (i.e., to only extract keywords from fault logs that also appear in an SRS). Because inspectors report faults using unrestricted NL and those logs may include phrases (e.g., understand, inconsistent) that don't exist in PGCS SRS. For example, considering the sample isolated fault log, *"the initial value rvariable defined 10000 but defined 1000 inconsistent value rvariable"*, the set of 1-word phrases extracted from KESRI approach included {'rvariable', 'defined', 'inconsistent', '10000'}. In the same example, manual extraction of 1-word phrases included {'rvariable', 'value', '10000', and '1000'}. Few major observations when comparing the outputs of KESRI vs. Manual extraction based on this one sample scenario is discussed below:

- TPs are {‘rvariable’, ‘10000’} because these phrases appeared in KESRI output and were also manually extracted;
- FPs are {‘defined’, ‘inconsistent’} because these terms were only extracted from KESRI and did not appear during the manual extraction because of the constraint that terms to be extracted should also appear in an SRS. This was done because different inspectors tend to use different verbiage to indicate the same fault and I am only considering phrases extracted from fault logs that also appear in an SRS.
- FNs are {‘value’, ‘1000’} because these does not appear in the KESRI output but were manually extracted. These values did not appear in KESRI because I used an isolated fault (as an example to explain the evaluation process), but the KPE algorithms (supervised and unsupervised) uses different algorithmic underpinnings to extract keyphrases. Results would be better with a greater number of fault logs.

Next, the entire fault logs (201 in number) are input to KESRI that output a set of 1-word phrases that are evaluated using the set of manually extracted keyphrases. The values of evaluation metrics are calculated using TPs, FPs, and FNs (as shown in the example above) for all the KPE algorithms

5.4. Results

This section presents the results centered around two RQs that evaluated the performance of 11 different KPE algorithms (belonging to supervised and unsupervised category) when extracting keyphrases from fault logs. To help the reader, the F-measure values for all 11 algorithms are grouped based on the length of keyphrases extracted. These results are shown in Figure 5.3 and discussed below. Major observations regarding the results presented in Figure 5.3 follows.

- ***1-word phrases***: F-measure of all algorithms varies between a minimum of 48 (for TOPIA) and a maximum of 66 (for KEA). KEA performed the best but still misses some relevant phrases that are otherwise extracted from an expert.
- ***2-word phrases***: F-measure for 2-word phrases was higher across all algorithms (when compared against F-measure for 1-word phrases). Yet again, KEA performed the best with F-measure of 77%, a big improvement.
- ***3-word phrases***: F-measure for 3-word phrases could only be calculated for KEA and Wingus because all other algorithms could not output enough 3-word keyphrases to enable F-measure calculation. Of the two that did report the values, KEA performed best with an F-measure of 83%.

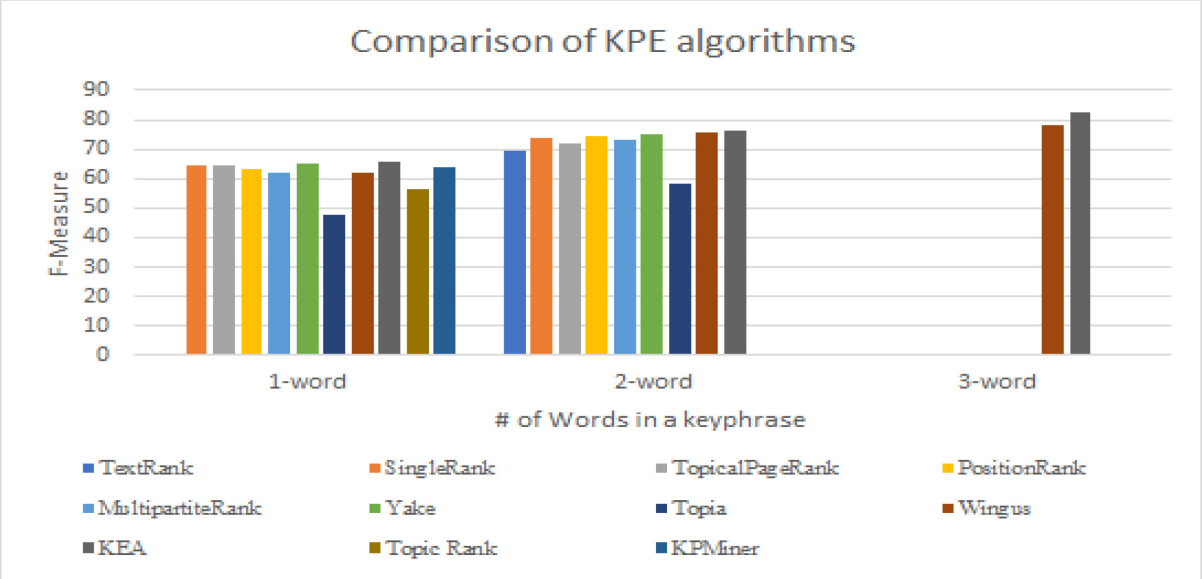

Figure 5.3. Comparison results of KPE algorithms

Implications*:* Based on the above key findings, KEA was most likely to retrieve most relevant keyphrases from requirement inspections fault logs as would have retrieved manually by an inspector. It can be speculated that supervised learning algorithms (Wingus and KEA) i.e.

feature based are better suited than most unsupervised learning algorithms (both statistical and graph based) to extract relevant keyphrases.

5.5. Discussions

Few algorithms (e.g., TopicRank, TextRank, and RAKE) did not retrieve adequate number of 3-word phrases, and this scenario was most prevalent with graph-based algorithms. It was observed that the following factors were crucial for the retrieval of low number of phrases.

- Most of the graph-based algorithms extract nouns, proper nouns, and adjectives from the text (fault logs in our case). Also, these graph-based algorithms extract keyphrases based on certain combinations of nouns with adjectives, or extraction of those keyphrases where noun and adjectives appear within a certain window of words etc. There were only 201 reported fault logs that were further preprocessed (removal of stopwords) and resulted in even shorter text. This was one of the prominent reasons for certain algorithms such as TopicRank, TextRank etc., resulted in very few numbers of phrases.
- Another reason for retrieval of limited number of phrases was that the RAKE algorithm was configured to retrieve the words that have minimum frequency of occurring > 3. Since, our faults logs are preprocessed and was a smaller text corpus, which made it difficult for the algorithms to extract keyphrases that occur frequently (i.e. more than 3). Adjusting the frequency of occurrence to less than 3, compromised with the effectiveness of keyphrases.

The following conclusions can be drawn regarding the best performing algorithms:

- Supervised learning algorithms performed best (based on their higher F value) because they used training model to identify the keyphrases. In our case, the model was trained on

actual PGCS SRS and was later used to extract keyphrases. The trained model was created using Naive Bayes classifier.

- Most of the unsupervised (graph based and statistical) algorithms performed poorly when compared with supervised algorithms.

6. MAPPING FAULT LOGS TO SRS REQUIREMENTS

This proposed work is the extension of our work (Keyphrase Extraction in Software Requirements Inspections) a.k.a., KESRI approach (discussed in chapter 5) that automated the extraction of key phrases from NL fault logs (output of the inspection process). Post-inspections, KESRI can help requirements authors by extracting relevant keyphrases from NL fault logs but they need to be mapped to individual SRS requirements. This mapping of extracted keyphrases to SRS requirements is a cumbersome, time-consuming, and tedious process. The current work maps the keyphrases (that conveys the fault context) to the requirements in an SRS. This chapter presents our proposed approach i.e., Mapping Of Keyphrases to SRS Approach (MOKSA) that input keyphrases of length 3-words (generated in by KESRI), SRS document, and similarity algorithm. The output of MOKSA algorithm is faulty requirements that are highly impacted by the faults from fault logs.

6.1. Proposed Approach

As discussed in chapter 5, the MOKSA takes input keyphrases (e.g., 3-word length), SRS document (PGCS), and a similarity measure (i.e., semantic similarity) to identify fault-prone requirements that need fixation. This section presents the working example of mapping keyphrases to the SRS document with semantic similarity measures. The overall proposed approach is shown in Figure 6.1, and the example keyphrases from KEA are shown in Table 6.1.

As shown in Figure 6.1, the extracted keyphrases from the fault logs were inputted to the MOKSA. Our prior work automated the KPE from fault logs. Based on the prior results, the keyphrases of length 3-words were being extracted from class of ML algorithm (belonging to supervised feature-based algorithms) because they produced best results. To help the reader, the proposed mapping approach (shown in Figure 6.1) is explained using top-11 ranked keyphrases

(of 3-word length) extracted from a type of supervised feature-based ML algorithm (KEA). Each step in Figure 6.1 is described as following.

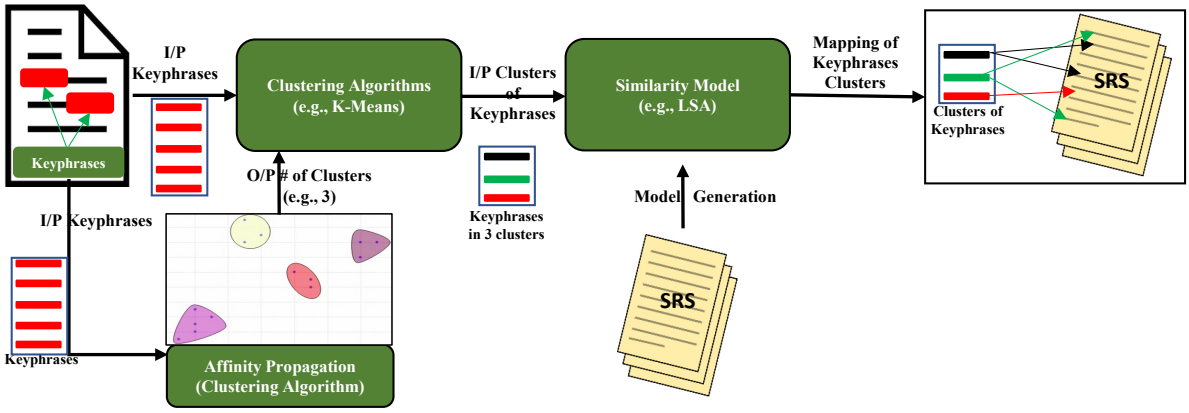

Figure 6.1. Overview of MOKSA approach

***Identifying the # of clusters*:** MOKSA takes input the user specified number of keyphrases (e.g., 11-keyphrases used in this example) and outputs # of clusters needed to most appropriately group the keyphrases as opposed to using a user predefined number of clusters. This is done through clustering algorithm called '*Affinity Propagation*'.

Clustering the keyphrases: Using a variety of clustering algorithms (agglomerative, K-Means, affinity propagation, spectral, and ward hierarchical), the keyphrases extracted were grouped (based on their similarity) into # of clusters (identified during the Step 1). As seen in Table 6.1, 11-keyphrases when fed to Affinity Propagation algorithm outputs the number of clusters (3 in this case) which were in turn fed to clustering algorithms (K-means in this case), and outputs the placement of those keyphrases in each of those 3 clusters (that are color coded). So, essentially, at the end of step-2, related problems were grouped together that were identified during the review and can next be mapped to the requirements in SRS.

***Training the model to map keyphrases to SRS*:** During this step, MOKSA trained a model using semantic similarity algorithms (e.g., LSA) to learn the NL semantics of the SRS

(e.g., PGCS) document. The learning algorithms find patterns in the learning data (i.e., SRS document) that could later mapped the queried data (i.e., clustered keyphrases) to the requirements in an SRS.

Table 6.1. Output of MOKSA approach using sample 3-word keyphrases

Keyphrases from KEA		**Affinity Propagation**	**K-Means Clusters**
Step-1 (3-word length)	car leaving parking, number parking spaces, parking space available, driver call someone, someone help cashier, previously 1000 10000, monthly ticket increase, ticket increase rvariable, purchasing monthly ticket, nothing say gate, say gate close	3 clusters (Step-2)	**Step-3** {car leaving parking, number parking spaces, parking space available, someone help cashier}, **{nothing say gate, say gate close}**, {driver call someone, previously 1000 10000, monthly ticket increase, ticket increase rvariable, purchasing monthly ticket}

Testing the model: The clusters of keyphrases (output of *clustering the keyphrases* step) were queried against the supervised ML model trained on SRS (i.e., PGCS). This step queries each cluster (containing keyphrases from fault logs) one at time against the trained model and outputs the list of requirements that were most related to clustered keyphrases i.e., maps clustered keyphrases to potentially faulty requirements in an SRS.

6.2. Study Design

This section present details regarding experiment design that was used to evaluate MOKSA including research questions explored, variables exploited, data collected and evaluation metrics used in this study.

6.2.1. Research Questions

Two major research questions were evaluated in this study and are listed below.

RQ-1: Which clustering algorithms provide the most accurate clusters for keyphrases from fault logs?

RQ-2: Which semantic similarity measure (LSA or LDA) performs more accurate mapping of keyphrase clusters to SRS requirements?

6.2.1.1. MOKSA study design

This section presents some important details about the design of artifacts, inspectors, and evaluation metrics used in this study.

***Artifacts and inspectors*:** This study is an extension of KESRI and utilizes same # of inspectors (i.e., 41) and artifact (PGCS that resulted in 201 fault logs).

***Evaluation metrics*:** The MOKSA included clustering algorithms for clustering keyphrases (that were being evaluated - RQ1) and also included semantic similarity techniques (that were being also being compared - RQ2). To answer RQ1, literature search [81] yielded three most commonly used cluster evaluation metrics that were applied when evaluating MOKSA's key clustering algorithms. A brief description of these three metrics follows:

- *Rand Index*: considers pair of keyphrases (at a time) and then evaluates their cluster placement by comparing it with manual clustering. Rand Index outputs whether the pairs of keyphrases are placed in the correct cluster (assuming an expert is placing keywords in clusters correctly) [81]. As an example, if 6 keyphrases are partitioned in two different manners (manual partition and algorithm partition), Rand Index performs pairwise comparison (for all 15 pair of keyphrases 6C2) and outputs the # of agreements and # of disagreements in manual vs. algorithmic clustering.
    - # of pairs of elements that are both correctly clustered by both manual and algorithm (referred as a);
    - # of pairs of elements that one or both are incorrectly clustered by algorithm (again assuming manual clustering the correct) and referred as b;

The rand index is calculated as (# of pairs in agreement)/ (total # of pairs), i.e., (a)/ (a+b). The denominator is total number of pairs, so, rand index represents the frequency of agreements over total # of pairs. The output value can range from 0 (no agreement) or 1 (complete agreement) between algorithm and manual clustering.

- ***Information theory-based metrics (adjusted and normalized) mutual information***: evaluates the cluster performance based on their purity i.e., keyphrases placed in the correct clusters. But purity is easily manipulated to generate high value e.g., for singleton clusters the purity is as high as 1. To adopt a more precise measure, adjusted mutual information (AMI) and normalized mutual information (NMI) between two clusters is being used to measure the information one cluster carries regarding the true cluster belongingness. We normalized mutual information score to scale the results between 0 (poor clustering) and 1 (perfect clustering). NMI is not adjusted to evaluate predictions by a chance i.e., the NMI score is unbounded when a large value for # of clusters is used for fixed number of data elements. On the other hand, the adjusted mutual information is adjusted for predictions based on chance and can be scaled to have values in range [0, 1]. Both these metrics are compared along with Rand Index as part of RQ1 [81] [82].

In regards to RQ2, the mapping of keyphrase clusters to individual SRS requirements (generated by LSA and LDA) are being evaluated by comparing it against the clusters generated by the domain expert (often referred to as evaluation by semantics [81]).

6.3. Experiment Procedure and Validation

This section provides working details of MOKSA including description of tuning parameters used in algorithms used in this study. MOKSA was executed using Python 3.7 library version along with API from other packages including NetworkX (for graph models), Gensim

(for semantic similarity approaches like LSA and LDA) and SKLearn library (to implement various clustering algorithms). The overview of experiment procedure is presented below and is organized around steps in Figure 6.1.

***Step 0- preprocessing input to MOKSA*:** At the completion of inspection study, a total of 201 true faults were identified. These fault logs were fed to KESRI approach (Chapter 5) and resulted in a list of top-N keyphrases (where N is 60) and each keyphrase is of length 3. A list of top-60 key phrases are inputted to MOKSA.

***Step 1 - identifying the # of clusters*:** MOKSA executed '*Affinity Propagation*' clustering algorithm and outputs the # of clusters where all the keyphrases can be placed. The top-60 keyphrases (output of Step 0) when fed to *Affinity Propagation* algorithm resulted in 14 unique clusters.

***Step 2 - placement of keyphrases in each cluster*:** Clustering algorithm grouped top-60 keyphrases (output of Step 0) into 14 clusters (output of Step 1). This process was executed for each of five clustering algorithm (used in MOKSA) and their performance was compared using evaluation metrics to identify the best performing clustering algorithm.

Validation: Each clustering algorithm placed each keyphrase to a particular cluster which were evaluated against the manual cluster assignment (by the domain experts). Then, evaluation metrics (random index, adjusted mutual information gain, and normalized mutual information gain) measured the performance of clustering algorithms in terms of values ranging from 0 (poor clustering) to 1 (best clustering). The clustering algorithms that were categorized as pairwise distance between neighbors were run 10 times and then average of all the values was used as the final value for the evaluation metrics. This step was performed because K-Means randomly selects the initial means for the given # of clusters (i.e. 14 in this case) that results in different

values for each run. So, to estimate the metric value for these clustering algorithms, it is a common practice to run it several time and report the mean of all the runs as the final value.

Step 3 - mapping of keyphrase clusters to individual SRS requirements: To map keyphrase clusters (output of Step 2) to individual SRS requirements, PGCS requirements were used to train the model with two competing similarity algorithms (LSA vs. LDA). Next, the trained model was queried by the keyphrase clusters (identified by best performing clustering algorithm from Step 2) and mapping of clusters to PGCS requirements was generated by LSA and LDA. This mapping was evaluated by a domain expert, who validated the mapping produced by LSA and LDA by comparing it against their predicted mapping (i.e. if they were to manually map keyphrase clusters to requirements in an SRS). During the comparison, output of LSA/LDA were generated by varying different similarity threshold values (ranging from 80% similarity to 100%).

6.4. Results and Discussion

This section presents the result in terms of RQs identified in Section 6.2.

RQ-1: evaluates performance of 5 clustering algorithms in terms of the placement of top-60 keyphrases in each of 14 clusters. Table 6.2 shows a comparative evaluation of these 5 clustering algorithms using three evaluation metrics. Major observations from Table 6.2 are as follows:

Table 6.2. Cluster evaluation metrics

Metrics	**Pairwise neighbor distance**			**Graph based neighbor distance**	
	K-Mean	**Ward Hierarchical**	**Agglomerative**	**Affinity Propagation**	**Spectral**
Rand index	0.52	0.62	0.62	0.72	0.36
Adjusted mutual information	0.61	0.71	0.70	0.73	0.49
Normalized mutual information	0.79	0.84	0.83	0.86	0.74

- Based on Rand index, the performance of clustering algorithms in terms of level of agreement varied ranging from 36% agreement (Spectral) to 72% agreement (Affinity propagation). While a*ffinity propagation* algorithm performed best, the overall performance of algorithms was generally low for Rand index.
- Based on information theory metrics, algorithms exhibited better performance. Yet again, affinity propagation performed the best and Spectral and K-Means algorithms performed worse. While the value (73%) was same as for Rand Index, NMI metric produced best results. Most of the algorithms clustered keyphrases at a high rate, specifically, ward hierarchical, Agglomerative and Affinity propagation were best performing.

Discussion on RQ-1: Additional insights from the results (in Table 6.2) are discussed below:

- *Type of Metric*: The Rand index values showed large disparity among clustering algorithms and may not be applicable to all family of clustering algorithms. Some algorithms used bottom-up clustering approach and pair wise comparison was not best suited to evaluate their performance. Also, Rand Index was more rudimentary metric where only # of complete agreements points were considered. Both NMI and AMI metrics measured mutual dependence between two clusters and used information gain to evaluate clustering performance as opposed to just relying on # of pure agreements. The AMI has lower score than NMI because there were more restrictions-imposed w.r.t generating predictions by chance. However, AMI score is a better indicator for situations where # of clusters are very large and it is not very true predictive of cluster performance when # of clusters are low (as in our approach with only 14 clusters).

- *Worst Performing*: S*pectral clustering* algorithm, graph-based algorithm (that performs best for fully connected graph) uses linkage between graph nodes to calculate the pairwise distance between the keyphrases in each cluster (pairwise distance between the nodes of a graph). During the execution, it was discovered that Spectral clustering did not always result into fully connected graph which in turn impacted the clustering performance.
- *Best Performing*: Affinity propagation identified the prominent exemplars from the list of keyphrases in each cluster. The exemplars were iteratively updated by calculating their suitability to become the exemplar of another cluster. This iteration process continued until the algorithm converged (i.e., no more exemplars were identified). As the final clustering was a result of multiple iterations to identify true exemplars, this algorithm reported the best score among all clustering algorithms.

Implications: From the results and the discussion around RQ1 presented above, it can be concluded that the most appropriate clustering algorithm to cluster keyphrases from software fault logs were *Affinity propagation and hierarchical clustering algorithms.* Assuming the type of the data (i.e., keyphrases from software fault logs) the # of clusters was small and there is negligible amount of 'chance' to miscalculate MI; the NMI metric can be true representative of most appropriate clustering evaluator.

RQ-2: This RQ evaluated the mapping of keyphrases clusters (produced by *Affinity Propagation* algorithm - Best performing clustering algorithm) to individual PGCS requirements. Two semantic based mapping approaches (LSA and LDA) were evaluated using two different training models created for LSA and for LDA separately over an entire PGCS SRS document that included 61 requirements (functional and non-functional). During the testing, each

of 14 keyphrase clusters were queried (using LSA and LDA) against the respective training models. The output generated from LSA and from LDA is shown in Figure 6.2.

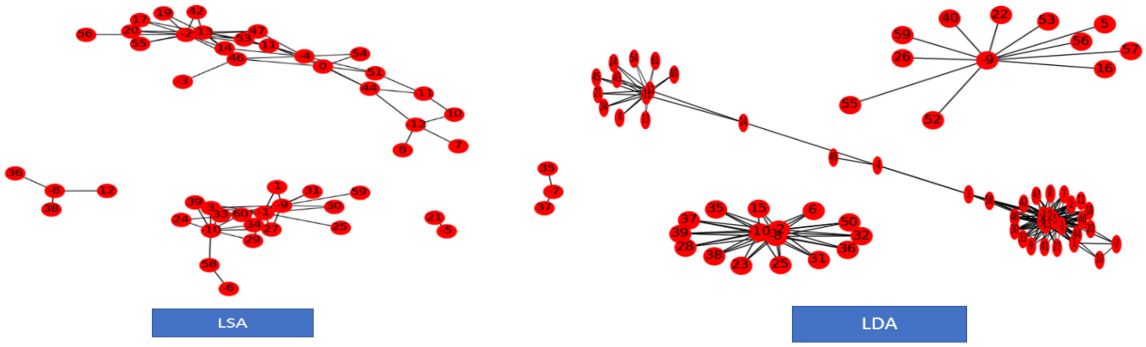

Figure 6.2. Cluster evaluation using graphs with LSA and LDA

The values shown in individual nodes can be positive or negative. The positive values denote the requirements # in PGCS (this value can range from 1-first requirement to 61-last requirement). The negative values denote the # of keyphrase cluster (this value can range from 0-first cluster to 13-last cluster). The edges represent similarity between keyphrase cluster and individual requirements. For example, a subset of LDA output shown in Figure 6.3, and it indicates that node denoted with -9 (cluster #10) is mapped to individual requirements denoted by nodes 5, 16, 22, 26, 40, 52, 53, 55, 56, 57, 59. This means that keyphrases in cluster#10 are similar to the text of requirements that it is linked to.

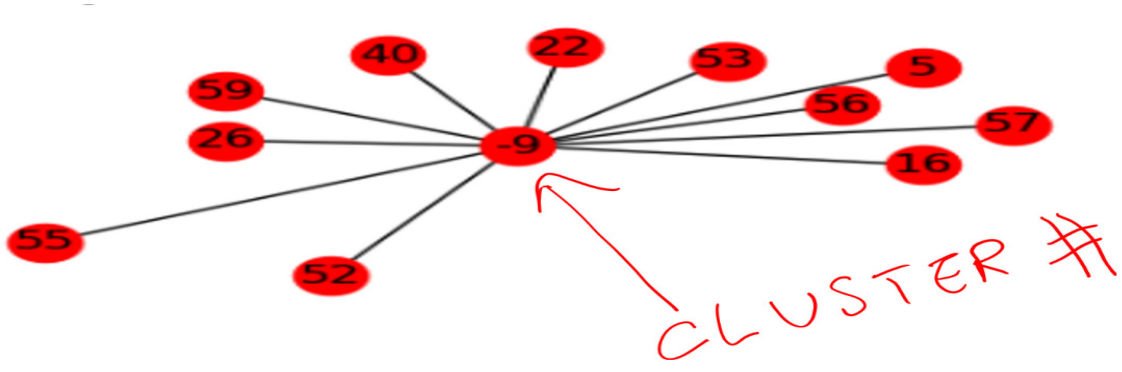

Figure 6.3. Example of one cluster from LDA

To evaluate the correctness of this clustering output, a general practice [81] is to manually perform the mapping (as a requirements author would do) and identify disagreements between automated output and manual output. To provide an overview of the results shown in Figure 6.2, two major observations became apparent as listed below and discussed:

Best performing algorithm: LDA performed a more accurate mapping of keyphrase clusters to requirements and is recommended to be used for post-fixation analysis. The process of evaluation of this RQ was performed by a domain expert through analysis of semantics of requirements generated with LSA and LDA mapping. The discussion about important observations (identified by the domain expert) is presented as following.

- LDA is a probabilistic algorithm that uses Bayesian classification to find the similarity of NL text within the trained model, whereas, the LSA uses singular value decomposition (SVD) based on tf-idf scores of the terms (proof is beyond the scope of this paper). LSA uses tf-idf score to assign the weights to the words during model training and then uses the same principal for queried words to map to similar requirements.
- The queried clusters of keyphrases had inspector specific words that did not exist in the training and hence resulted in mapping them to wrong requirements. Whereas the LDA using Bayesian classification (which is probabilistic) assigned mapping of unseen words based on probabilities.

Similarity threshold: One of the major analysis involved at identifying/evaluating the mapping was the determination of the threshold at which similarity between keyphrase text and PGCS requirements was most appropriate. The most appropriate similarity threshold value was identified to be around 90% that generated most accurate mapping. The mapping of keyphrase clusters to requirements was approved or disapproved based on the similarity score threshold

generated during this process i.e. if a cluster A is tested for similarity with requirement R results in a similarity score value of 0.70, that implies that A & R have 70% similarity to each other. A high value of similarity score (more than 0.95) ensured that the highest semantically similar results are approved but it may result in unwanted exclusion of true results. Also, a low value for similarity threshold can result in inappropriate mapping, where, the mapping of keyphrase cluster is approved because the inappropriate mapping satisfied the threshold value. To evaluate the optimal threshold value, the authors executed the experiment with multiple threshold values and identified that the most accurate result of mapping is achieved by setting the threshold value to be around 90%. The mapping of clusters to requirements is shown as a graph in Figure 6.2.

Implications: The LDA graph generated independent/dense clusters of requirements impacted by one or more faults, while the clusters of LSA were slightly more interconnected. Manual evaluation reported that the LSA was prone to miss to include some of the requirements within the clusters (which can be mapped into the clusters by setting a low threshold value than 90%, but then it results in inappropriate mapping). Based on these results, LDA was recommended for mapping keyphrase clusters to SRS requirements.

7. CHANGE IMPACT ANALYSIS FOR FAULT FIXATION SUPPORT

The proposed work in this chapter attempted to automate fault-fixation during software inspection process by finding IRRs. The IRRs were identified by first generating a semantic score of the SRS, followed by querying each requirement against the generated semantic score (see Figure 7.1) to identify related fault prone requirements during fault fixation (goal-3). The related requirements for each query were then transformed into a graph model to be mined using graph mining algorithms (cliques, K-Clique communities etc.) to extract strongly similar requirements (i.e. IRRs) based on similarity threshold. The IRRs can help requirements authors pre-inspection to improve quality of SRS document by removing *redundant and extraneous* requirements. Post-inspection, IRRs can be used to identify those requirements that may have been impacted or need similar fixes when fixing the faults reported during the requirements inspection.

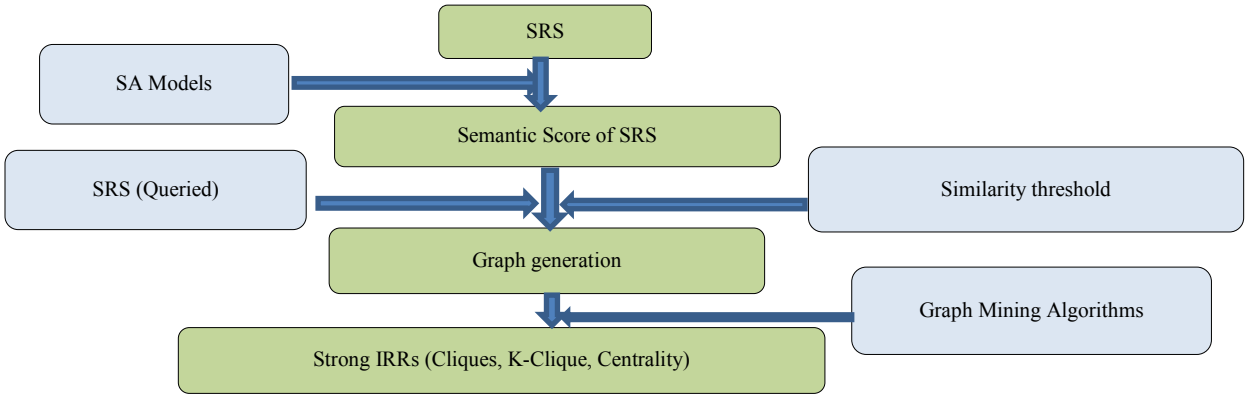

Figure 7.1. Overall steps for evaluating CIA during fault fixations

7.1. Proposed Approach

The proposed work in this paper has two major RQs that are identified, and our approach is discussed around these RQs.

7.1.1. Research Questions

RQ-1: Generation of IRRs for an SRS to help requirements author pre-inspection**.**

RQ-2: Analysis of impact of a change using IRRs post-inspection

7.1.1.1. Proposed approach for RQ-1

This RQ helps the requirements author pre-inspection to locate redundant and extraneous requirements; these requirements can be extracted from graph of IRRs (Figure 7.2). The overall procedure is explained in following steps:

Step 1: This RQ aimed at applying semantic analysis over complete SRS document to develop term-document matrix.

Step 2: Next, each requirement was queried against term-document matrix obtained for SRS document to find most similar requirements for that query.

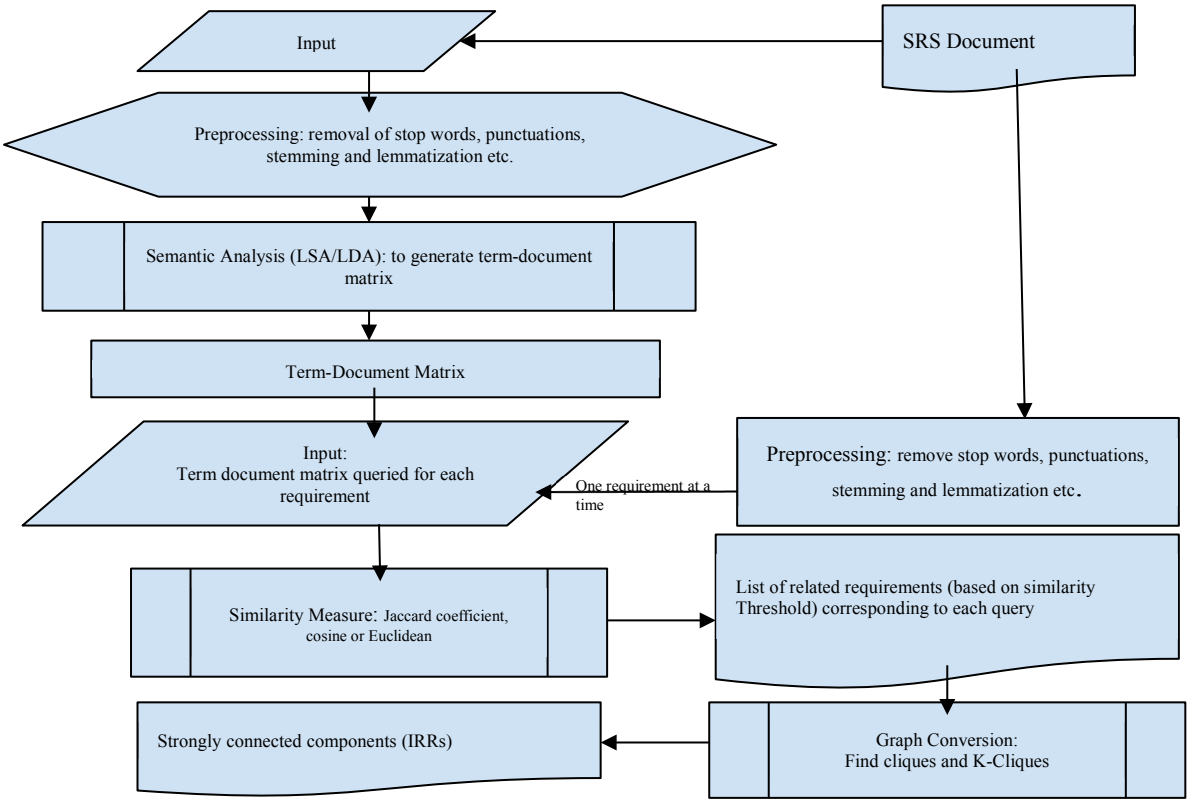

Figure 7.2. Steps involved in generation of IRRs

Step 3: The selection of similar requirements is based on most optimal value of similarity threshold (refer Figure 7.2 for *'similarity measure'* step). The optimal value of similarity threshold is a challenge and needs to be evaluated experimentally.

Step 4: These similar requirements obtained as a result in step 3 for each query were labelled as related requirements corresponding to that query.

Step 5: The output in step 4 for all queried requirements were converted to graph to apply graph mining techniques e.g. Clique mining or K-Cliques mining.

Step 6: The graph obtained in step 5 can be analyzed for extraneous requirements by mining for any singleton node. Redundant requirements can be extracted by setting a higher similarity threshold value.

Validation of RQ-1*:* The validation of RQ1 can be achieved by analyzing redundant and extraneous requirements obtained with actual seeded redundant requirements in SRS document. SRS document (like RIM, PGCS and LAS) contained this information (i.e. requirement #, fault-types etc.). The expected outcome (i.e. redundant requirements) can be identified with higher value of similarity threshold (close to 1).

7.1.1.2. Proposed approach for RQ-2

Several requirements in an SRS were interrelated to many other functionalities in a software project under development. The knowledge of IRRs can help understand CIA because a change in one requirement tend to affect semantically similar requirements. The overall approach is shown in Figure 7.2 and is described in following steps:

Step 1 to 5: These steps are same as described in RQ1.

Step 6: Any cliques or K-Clique found for the graph in step 5 gives strongly interrelated requirements (IRRs) in an SRS document.

Step 7: The output in step 6 is used to analyze change-impacted requirements in an SRS document post-inspection.

***Validation of RQ-2*:** The validation of RQ2 was two-fold: 1) analyzing change-impacted requirements and, 2) re-inspecting requirements that needed a fix. Validation of CIA can be achieved by analyzing output of K-Clique communities (Figure 7.3) and finding N-hop paths to determine impact of change on subsequent requirements. For example, see Figure 7.3, where the impact of change in requirement-A was analyzed on requirements B and C that are 2-hops away (highlighted in green) from the source of change (i.e. A). This analysis was useful in predicting ripple effect of a change. For the validation purpose, a domain expert analyzed the requirements and created a group of those requirements that were highly similar and would impact other requirements if there were a fault. Next, the graph mining algorithms were used to extract the strong IRRs and were validated against the prediction of a domain expert.

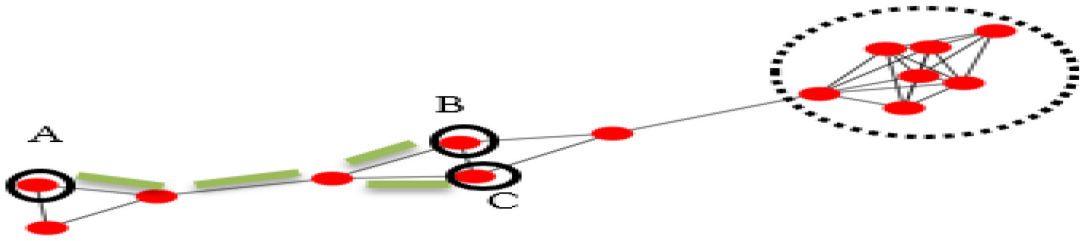

Figure 7.3. Outcome and evaluation of RQ2

Post-inspection, validation of requirements that needs a fix can be achieved using IRRs (obtained by graph mining algorithms) while changing or fixing a seeded fault in an SRS document (see dotted circle in Figure 7.3). SRS documents (like RIM, PGCS and LAS) contain the fault information (i.e. requirement number etc.) along with fault-types. Inconsistent information (II) fault-type describes inconsistency in similar requirements but across various parts within the SRS document. The expected outcome (i.e. interrelated requirements) of RQ2

should have at least fault-type II as part of cliques and K-Cliques. The K-Cliques were calculated using clique percolation method (i.e., incrementally building over K-1 cliques to construct K-Clique). The algorithmic description of Clique percolation method is out of the scope of this research and is not discussed. The existence of II fault-types in strongly connected components can assure that the proposed work (RQ2) was able to detect potential fault-prone areas.

7.2. Experiment Design

This section presents experiment design followed to implement proposed approach. The experiment procedure as shown in Figure 7.2 was followed while implementing semantic analysis (SA) algorithms to develop IRRs for an SRS. Few details regarding experiment design are as follows:

- ***Semantic analysis algorithm used***: The initial results were obtained using LSA algorithm.
- ***SRS artifact:*** The SRS artifact (wonders of weather) that is being tested in this approach is developed by software engineer graduate students at North Dakota State University (NDSU) for real-world requirements and was four pages long that consists of 21 requirements. This SRS is selected because of its small number of requirements that can be checked/evaluate comfortably.
- ***Preprocessing:*** Natural language toolkit (NLTK) was used to remove stop words, punctuations and perform lemmatization in the SRS document. Acronyms were substituted with full names to construct a rich vocabulary.
- ***Toolkits used:*** Processing of SRS text was handled using NLTK to filter stop words and infrequent words. Gensim open source package was used to execute LSA algorithms. To implement graph concepts, a python-based package named ‘NetworkX’ was used. The

results from queried requirements (step 2 in RQ1) were further filtered keeping those requirements that had similarity score greater than 95%.

- ***Validation of results:*** The results obtained were validated against the IRRs extracted by a domain expert versus IRRs extracted using LSA model.

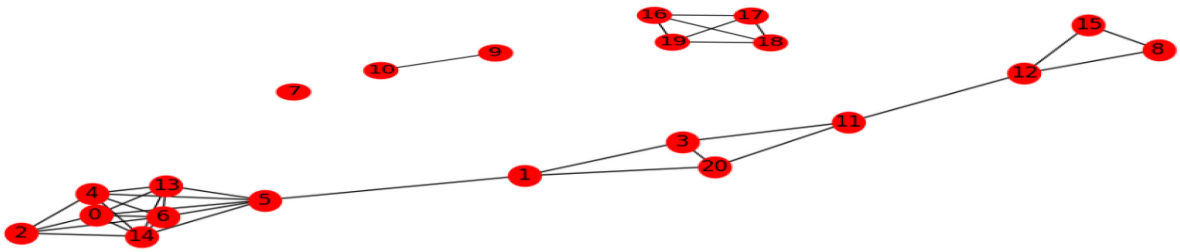

Figure 7.4. The IRR graph generated using LSA

7.3. Results and Discussion

The results obtained are shown and discussed using Figure 7.4, Figure 7.5, Figure 7.6, and Table 7.1. The requirement numbers (shown in Table 7.1) are obtained by implementing various graph mining algorithms. The requirement numbers in square brackets (i.e. []) shows semantically similar requirements. Three graph mining algorithms were implemented and are shown in Table 7.1. The results are discussed corresponding to RQs identified earlier.

Table 7.1. Resulting IRRs with graph mining approaches

Attributes	**Description**
# of Reqd.	21
Cliques	[0, 4, 14, 6, 5, 13], [0, 4, 14, 6, 2], [16, 17, 18, 19], [1, 3, 20], [3, 11, 20], [8, 12, 15], [9, 10], [11, 12], [1, 5], [7]
3-clique Communities	[0, 2, 4, 5, 6, 13, 14], [1, 3, 20, 11], [16, 17, 18, 19], [8, 12, 15],
Centrality	[0, 4, 5, 6, 14] value=0.4

***RQ-1: IRRs and pre-inspection decisions*:** The graph of IRRs that was generated for the SRS used in this study is shown in Figure 7.4. There were few highly IRRs (cluster of nodes), singleton (that exists standalone) and redundant requirements as shown in Figure 7.4. These

requirements were manually tested and evaluated for extraneous or standalone or redundant requirements. The results in Table 7.1 were obtained by implementing various graph mining algorithms. For example, results from clique mining algorithm (req # [7], [9 & 10], [11 & 12]) were standalone and represented highly co-related requirements. The author manually checked the relevance of results and found out that requirement #7 was standalone requirement i.e. it presented important functionality but was not related to other functional requirements. This is crucial insight on assessing CIA if a change is made in requirement #7 which is standalone. Requirements #9-10 and 11-12 were very similar (fairly redundant) because they contained information about some common functionalities. Pre-inspection, these results can be tested for term-substitution faults to limit ambiguity and redundancy across similar requirements.

***RQ-2: IRRs to find fault-prone areas and to study CIA*:** The results (IRRs) obtained in Table 7.1 are used to identify fault-prone requirements to study CIA and the evaluation is shown in Figure 7.5 and Figure 7.6.

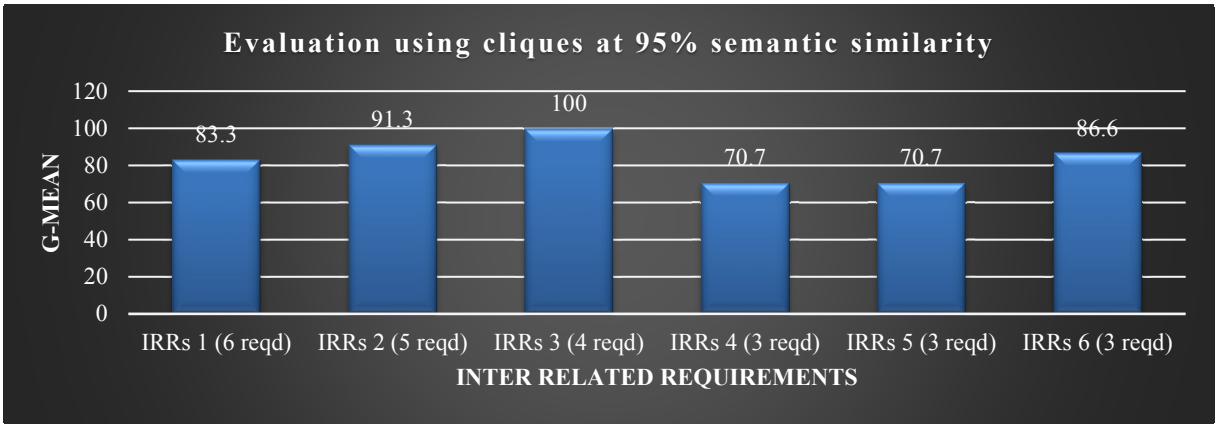

Figure 7.5. Clique evaluation

The analysis is performed using G-mean (X-axis) as evaluation metrics and two graph mining approaches (i.e., Clique mining and K-Clique mining) that extracted IRRs (Y-axis). Both the graph mining algorithms extracted IRRs using 95% semantic similarity threshold and

contains different # of requirements in each IRR (see Figure 7.5 and Figure 7.6). The results are discussed as following.

- *Clique evaluation*: cliques provided strongly IRRs (or fault-prone areas) and validation against manually extracted IRRs also revealed that these requirements are highly correlated. Since the cliques are very restrictive and extract only those grouping that are highly connected to each other and hence, these missed few requirements that were otherwise identified as related by a domain expert. Therefore, there g-mean value ranged between 70 and 100. But these cliques were able to identify some strong related requirements (e.g., requirement # [16, 17, 18, 19], [1, 3, 20]), and any change in one requirement (say req# 3 in 1-3-20 clique) required to re-inspect req# 1 and 20 because 1-3-20 are fault-prone (strong IRRs).

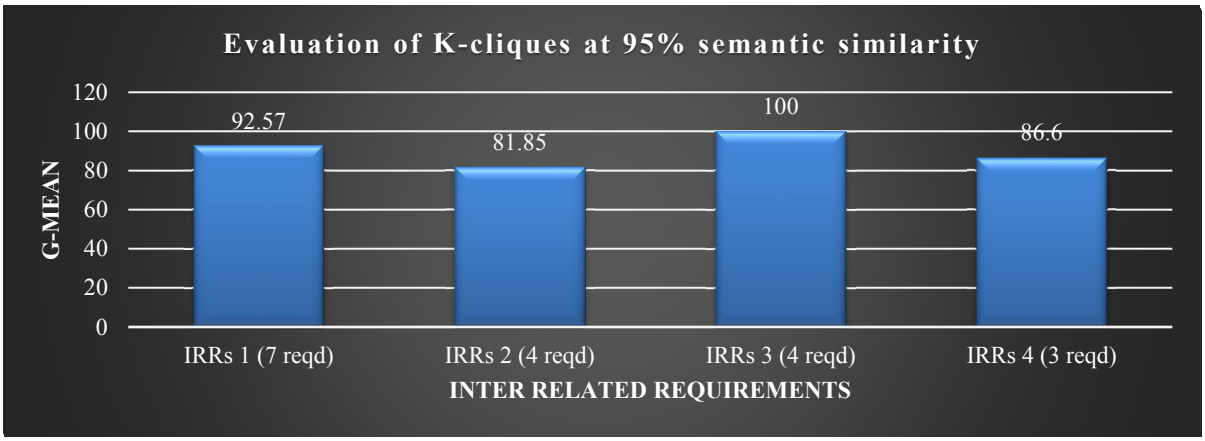

Figure 7.6. K-Clique evaluation

- *K-Clique Communities*: The study of CIA was further strengthen using K-Clique communities (K=3 in Table 7.1 and Figure 7.6). The K-Cliques were generated using clique percolation method and since the SRS document used in this study only had 21 requirements, so a smaller value of K (i.e., 3) was used to extract strong IRRs. The requirements were not documented considering to be related only to a specific group of

requirements. So, the K-Cliques are better suited to find loose (but still high) relatedness among requirements. From Figure 7.6, it was observed that the K-Cliques had shown better G-mean value than Cliques and the value ranges between 81 and 100. This graph mining algorithm allowed author to further check the impact of Change in req#3 on subsequent requirement (i.e. req# 11 in K-clique 1-3-11-20) and this ripple effect was identified using K-clique community algorithm or finding N-Hops in a graph (explained in section 7.1). Centrality measures central requirements in a graph. More the centrality (i.e. close to 1), the more CIA is needed. The proposed algorithms presented few emerging insights and will be explored in future on few more industry strength SRSs to better understand the most applicable SA and graph mining algorithms.

8. APPLICATION OF ML TO OTHER AREAS

This chapter presents few of the application area (w.r.t graphs) that was also explored as part of this dissertation. The long-term goal of this application is to be able to extract ripple effect during requirements CIA during fault fixations. But this research is not limited to requirements alone and can strengthen general approaches in graph theory, especially, path search algorithms, finding all the paths of different lengths in a graph (also well suited to explore ripple effect of a change if requirements are represented in the form of a graph). For this purpose, this dissertation attempted development of an algorithm that is scalable, dynamic and robust to find existing paths in a graph. The existing path search algorithms in a graph do not scale well (when it comes to efficiency) as data size increases. These existing algorithms are not applicable for distributed processing due to increased pre-processing requirements and complexity. When using the existing algorithms, any addition of new edges to existing graph requires re-identification of paths between all vertices and re-computation of relevant metrics from scratch.

Vertical data structure has been used successfully in machine learning applications like exploring scalability in semantic web data management and classification using nearest neighbor classifiers and have shown improvement in efficiency and accuracy. Vertical bit vectors are processed independent of each other and this makes them an ideal fit for distributed processing. In this chapter, this dissertation discussed a vertical breadth first multi-level algorithm to find all paths in a graph.

8.1. Overview of Vertical Breadth First Multi-Level Algorithm

Our proposed approach is shown using a sample graph shown in Figure 8.1 and is further explained using the high-level steps shown in Figure 8.2 and described as following:

Step 1: Initial vertical bit vectors are obtained from adjacency matrix and are labelled for each vertex. That is, for a graph with four vertices, each column of adjacency matrix represents four initial vertical bit vectors (E1, E2, E3, and E4 are the four initial vertical bit vectors in Figure 8.1).

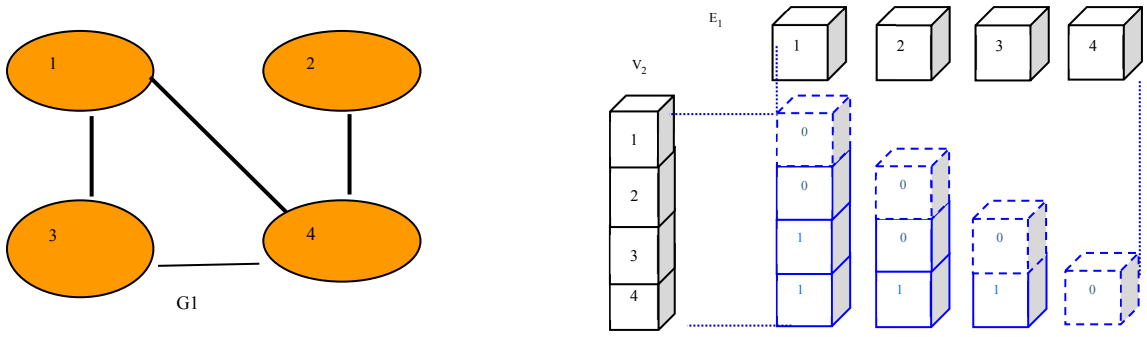

Figure 8.1. Example graph and lower triangular matrix representation

Step 2: Next, I applied a combination of breadth-first approach and logical operations to further explore existing paths from initial vertical bit vectors (more details appear in Figure 8.5).

Step 3: The algorithm keeps exploring until no new paths are found and then all the explored paths are grouped into levels to form a multilevel path tree (details in Figure 8.6).

Step 4: The paths explored are indexed and stored in index list to enable faster look-up.

Step 5: The shortest-paths can be found by querying the index list that was created in Step 4.

Our proposed solution does not require re-computation when new edges are added in a graph, thereby avoiding useless extra computations.

Contributions: I evaluated my algorithm on undirected and unweighted graphs because, in an undirected graph, the edge runs both ways between two vertices. Implementation of my proposed algorithm on undirected graphs can be helpful in analyzing graphs of social networks, social communities, internet networks etc., to mine social relations.

The contribution is demonstrated by the ability of my proposed algorithm to accommodate addition or deletion of an edge within the social media network, without re-computation. The breadth-first approach is versatile and can also be implemented in depth-first manner to suit distributed processing algorithms without excessive pre-processing overhead. The proposed algorithm presents a mathematical formula that are generalized for any size of graph i.e. our algorithm is scalable.

Similar terms: The terms *multi-level bit vector tree* (MBVT) and *Allpath* algorithm are used interchangeably in this paper. Throughout this paper, the term graph refers to unweighted bidirectional graph. The major advantage of using this sort of the graph is that when analyzing a social network graph, the presence of undirected link shows the existence of some relation between two persons. So, adding weight to the link does not represent any physical distance of separation among two vertices and is not relevant to our study.

8.2. Key Concepts Related to Graphs

This section provides some necessary details regarding terminology associated with graphs and vertical bit vector approach. The graph G1 is shown in Figure 8.1 with 4 vertices, 4 edges and its lower triangular matrix representation. Please note that the graph G1 in Figure 8.1 is used as reference to explain various terminologies defined below:

Vertices and edges: The 'v' in graph G1 represent set of vertices and 'e' represents set of edges that joins any two vertices. In graph G1, v contains 1, 2, 3, and 4 labelled vertices and e contains edges 1-3, 1-4, 2-4 and 3-4. Each edge is a pair (x, y) where x, y $\in$ v.

Loop and sub-graph: With respect to graphs, a *loop* is an edge from a vertex to itself. A graph that has a cycle is called *cyclic* graph and *acyclic* otherwise. An undirected graph without

any loop is called a *simple graph*. Various terminologies associated with graphs such as for H to be a subgraph of G, it must satisfy a condition such that H= (vH, eH) if vH $\subseteq$ v, eH $\subseteq$ e.

Undirected graph: The graph in which direction of edges is not considered i.e. an edge is considered both ways. For example, G1 is undirected graph, where an edge 1→3 ≈ 3→1.

Clique: A *subgraph* is called a *clique* if there is an edge between all pairs of nodes. It is an N-clique, if N vertices within a clique has edges between each possible pair. For example in G1, The edges connecting vertices 1, 3 and 4 makes a *3-clique* because every vertex is reachable directly from any other vertex.

Degree of a graph: Degree of a vertex in undirected graph is generally denoted as $v_i \in v$, where v_i is the number of edges incident on that vertex. For example, degree of vertex-4 is 3.

Adjacency matrix: The mathematical matrix representation to denote existence of an edge between all vertices of a graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph e.g. in Figure 8.1, presence of an edge between a pair of vertices is shown with 1 if there is an edge and 0 otherwise. The adjacency matrix has all zeros on its diagonal for a simple graph. This matrix is symmetric if the graph is undirected.

Vertical data structure: Vertical data structure refers to horizontal processing of vertical columns of an adjacency matrix. For the graph G1 (Figure 8.3), the vertical data structure is built from each column of adjacency matrix. The vertical columns (E1, E2, E3, E4) in Figure 8.3 represent the vertical data structures corresponding to each node in graph G1. Each vertical column (e.g., E1) represent the vertices that can be reached from vertex-1. One of the advantages of vertical data structure is that it can be independently explored to find all the paths from any starting vertex. The computational overhead and update of vertical data structure (with addition) of an edge is discussed later.

Ideally, in a connected graph, the connecting edge between two vertices should be considered only once to avoid repetition and extra work. Our algorithm avoided this extra pre-processing by using lower triangular matrix representation (Figure 8.1). Moreover, I marked the edge between vertexes '*v*' to itself with a 0 because our algorithm does not include any cycle during path generation.

8.3. Terminology Associated with Proposed Work

This section presents a detailed discussion on underlying mathematics and working of our proposed algorithm. The proposed algorithm operates over vertical bit vectors that generates a new path vectors (also in vertical form) using logical operations to form subsequent levels of a multilevel path tree. Hence, this algorithm is referred to as multi-level bit vector tree (MBVT) in this paper. The overall working of our proposed algorithm is shown in Figure 8.2.

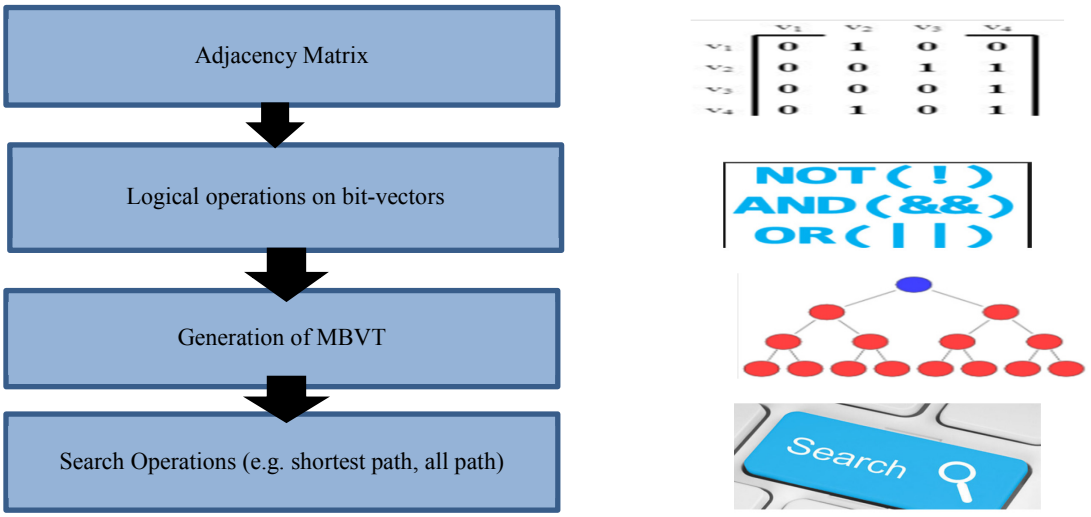

Figure 8.2. Overall working of proposed algorithm

The MBVT represents the paths in a graph, so in this study, it is interchangeably referred to as multilevel path trees or path trees or Allpath trees algorithm. The adjacency matrix serves as a root in path tree algorithm and newly generated bit-vectors (or paths) become subsequent levels (explained later in this section). The paths are generated and added at appropriate levels of

MBVT until the algorithm reaches stopping condition. The final MBVT is the basis to search for the shortest path when the source and destination nodes are given (discussed later). The advantage of MBVT is that it does not require re-computation with the addition and removal of a node from the graph.

The proposed algorithm is explained by using graph G1 shown in Figure 8.1 (also in Figure 8.3). Ideally, in a connected graph, the connecting edge between two vertices should be considered only once to avoid repetition. The graph G1 shown in Figure 8.3 has 4 vertices and 4 edges. This graph has been chosen from a simplicity point of view, to enable the authors in describing the working of the proposed algorithm.

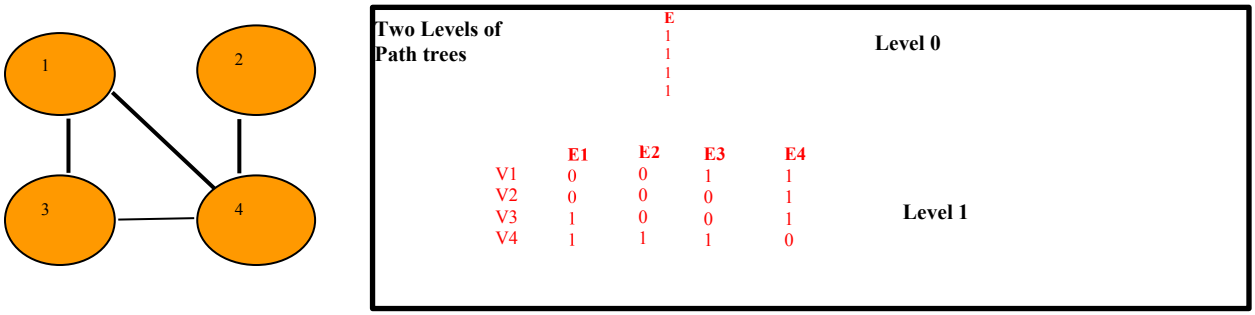

Figure 8.3. Two levels of path tree of graph G1

Definitions and Acronyms: The following definition and acronyms have been used in this paper.

- ***Edgepath***: It is denoted as a vertical bit vector and shows the presence of every edge from vertex 'v_i' to vertex 'v_j' in a graph containing N vertices (i, j ∈ N). ***Edgepaths*** are defined for each vertex as (E_1,…..,E_n); if N is 2 then there is a single edge (one path) connecting two vertices and if N≥2, then there are N edgepaths corresponding to each vertex.

For example, if N=3 then there are three edgepath vectors (E_1, E_2, E_3) and there can be at most two path/two edges starting from vertex 1 to vertex 3. The value of edgepath E_1 (0, 0, 1) means there is only one edge from vertex-1 to vertex-3 i.e. presence of 1 in an edgepath vector shows the existence of a path. Since the edgepath is corresponding to vertex-1 (i.e. E_1) and 1 is at index 3 (i.e. 001), there is a path from vertex-1 to vertex-3. In a similar way, if edgepath of vertex-2 is E_2 (1, 0, 1) then it means there are two edges from vertex-2; one edge runs from vertex-2 to vertex-1 and another from vertex-2 to vertex-3.

- ***Vector Length (VL)*** in our approach refers to the number of vertices present in the graph. For example, graph G1 in Figure 1 has 4 vertices so the *vector length (VL)* is 4 for this graph.
- ***Multilevel path tree:*** It is the path tree that contains all the paths reachable from each vertex. This tree is generated in the form of a hierarchy, where the root is an adjacency matrix and subsequent hierarchies are generated through logical operations (more details appear later).
- ***Level*** defines the hierarchy within multilevel path tree e.g. graph G1 in Figure 8.3, *Level*-1 represents all the paths that are at one edge distance from a specific source vertex. So, edges from V1 (1 is source vertex in this case) in Level-1 are [(V1, V3), (V1, V4)].

Similarly, Level-2 would show all the paths that are at 2-edge distance from the source vertex. If the source vertex is 1 then from Figure 3, the edges in level 2 are [(V1, V3, V4), (V1, V4, V3), (V1, V4, V2)].

Generation of edgepath vectors: In our all-path tree algorithm, each edge in the graph is generated with a mathematical formula. Edge path vector $E_k(V_m)$ in Equation 1 below has been used to generate all the edges present in a graph G1, where k, $V_m \in$ VL (Vector Length). The

generation of edgepaths vectors (E_1, E_2, E_3, and E_4) at level-1 in Figure 8.3 follows conversion formulae in equation 1.

$$E_k(V_m) = \begin{cases} 1, & \text{if } \forall_{k,m}, \exists \text{ an edge between } V_k \text{ and } V_m \text{ and } V_k \neq V_m, k,m \in VL \\ 0, & \text{otherwise} \end{cases} \quad \text{(Eq. 1)}$$

Generation of vertex mask path vector: The mask vector corresponding to each vertex acts as a filtering condition to exclude source vertex in the MBVT during all-path generation. The $M_k(V_m)$ in Equation 2, is used to label mask for each vertex in the graph G1 where k, $Vm \in$ VL (see Figure 8.4).

$$M_k(V_m) = \begin{cases} 1, & \forall_{k,m}, \text{ for bit number} = k \text{ and } k, m \in VL \\ 0, & \text{otherwise} \end{cases} \quad \text{(Eq. 2)}$$

Additionally, the complement of vertex mask (denoted by M'_h, where $h \in VL$) is used to exclude already generated vertices in MBVT during all-path generation. The step by step details of the operation are explained next.

Multilevel path tree algorithm: The all path algorithms use tree like data structure that stores data in vertical bits (i.e. column wise). The multilevel vertical bit-vector tree (MVBT) data structure is known as path tree (see Figure 8.3).

In Figure 8.3, the edge mask E_k (V_m) at level-1, consists of vertical bit-vectors in the form of 0 and 1 (for $\forall$ $V_m \in VL$). The value of 0 means there is no path from vertex k to vertex m and 1 means that there is a path from vertex k to vertex m $\forall_k, V_m \in VL$. A level-0 in Figure 8.3 represents the structure of bit vectors at the top of the tree.

For example: At level-0, the value 1 for all the vertices show that every vertex is reachable within the graph i.e. there is at least one path from a vertex k to some other vertex m in level-1. Similarly, the presence of 0 at level-0 for any *index* would show isolated vertex in a

graph. The notation for edge mask and vertex mask in the following section is going to be E_k and M_k for the sake of simplicity instead of writing $E_k(V_m)$ and $M_k(V_m)$ where k, $V_m \in VL$.

8.4. Proposed Approach

The proposed algorithm (all-path) starts with generation of adjacency matrix of the graph G1 (see Figure 8.4) followed by the creation of the edgepath vectors E_k and vertex mask vectors M_k. Compliments of vertex mask (i.e. M'_k) are created for the graph G1 to restrict the already generated vertices during generation of all the paths.

In Figure 8.4, the edge mask E_k is shown as a vertical bit vector. The length of E_k is equal to vertex length (VL), which is 4 in this case. Similarly, M'_k and M_k are written into vertical bit vectors. The all-path algorithm is developed to find all 2-length paths, 3-length paths up to the longest path present in the graph without having the need to re-compute during addition or deletion of a vertex. The underlying mathematical formulae of proposed algorithm is discussed next.

Generalized formulae for all length paths: In this section, a generalized mathematical expression for finding all-length paths in a graph has been explained. Our proposed algorithm makes use of breadth first approach but on vertically structured data. Our algorithm is generalized, and it can be implemented in depth-first manner. The advantage of using vertical vector representation is that the algorithm is scalable to graphs of all sizes.

The mathematical expression works by computing the logical AND between E_k and M'j (see Equation 3). These values are obtained from a list, which is denoted as $ListE_{i,\ldots,j}$ (see Equation 3). $ListE_{i,\ldots,j}$ has the index values of vertices that has 1 in their edgepaths. The steps to generate all-length paths using generalized formulae is explained with an example as follows:

Step 1: To explain the working, an edgepath vector E_1 $(0, 0, 1, 1)^T$ from level-1 is considered from graph G1 (see Figure 8.4). Here the superscript T is transpose that will represent the vector E_1 in vertical form. This example first compute 2-length path and that's why the general formulae $E_{h,\,k}$ is used to compute it (Equation 3).

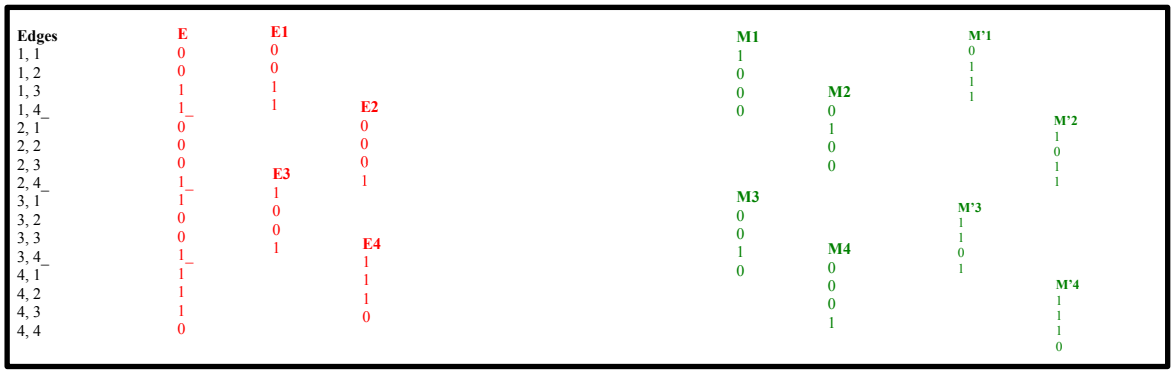

Figure 8.4. Path tree form of graph G1

Step 2: To generate level-2 for E_1, the value of 1 is present at index 3 and 4. That shows that there is a path from 1→3 and 1→4 (This is also visible in adjacency matrix). The $ListE_1$, in this case contains the values {3, 4}. These values represent k in generalized formulae.

$$k = \begin{cases} \textit{Index of m,} & \textit{where m=1 in } ListE_{i...j} \end{cases}$$

$$\textit{N-length path } E_{h,\,k} = E_k \;\&\&\; M'_h,\ (\forall \textit{other k, } E_{h,\,k}=0), \textit{and } N=2$$

$$\textit{N-length path } E_{h,\,i...j,\,k} = E_k \;\&\&\; M'_{i...j},\ (\forall \textit{other k, } E_{h,\,i...j,\,k}=0), \textit{ and } N\geq 3 \qquad \text{(Eq. 3)}$$

Step 3: The values of k are read one at a time to generate next level path vectors i.e. for level-2 starting at source vertex 1. The paths through vertex 3 (1→3→?) and vertex 4 (1→4→?) are explored and represented as edgepaths E_{13} and E_{14}. The symbol '?' is the vertex at level-2 yet to explore (see Figure 8.5).

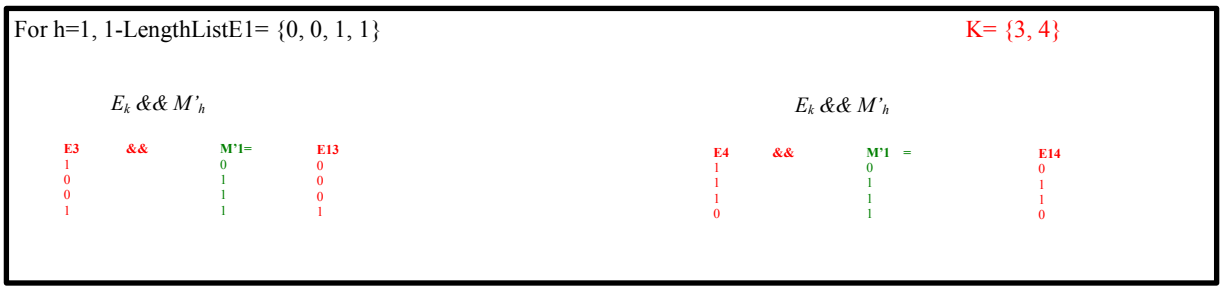

Figure 8.5. The 2-length paths from vertex 1

Step 4: Next, E_k and M'_h values are calculated. The first value of k from the $ListE_1$ is 3 (see step 3 above), so the E_k becomes E_3 and h is the starting vertex i.e. vertex 1. So, the value of M'_h becomes M'_1. Once, these values have been generated, the logical AND is performed between bit-vector E_k and M'_h as shown in Figure 8.5. The resultant edgepath vector from logical AND process is E_{13} $(0001)^T$ and E_{14} $(0110)^T$. The edgepath vector E_{13} has 1 at index 4, and the path is read as 1→3→4.

Step 5: The 2-length edgepath E_{13} $(0001)^T$ is explored to generate 3-length path for the MBVT approach using the formulae $E_{h, i\ldots j, k}$ as shown in Equation 3. E_{13} has a value of 1 at index 4, so, the value of k is 4, i…j is 3 and h is 1. The logical AND between E_4, M'_3 and M'_1 yields the edgepath E_{134} $(0100)^T$ displaying the path as 1→3→4→2. If the edgepath E_{134} is further explored to obtain 4-length path using the same formulae then it results in E_{1342} $(0000)^T$ and this is the stopping condition for further exploration.

Observation: The versatility of the algorithm lies in its capability to find all the paths by using logical AND operation over the MBVT, where the 1-length paths can be easily calculated from the level-1 of E_k. Whereas, 2-length and 3-length paths are calculated recursively from their immediate predecessors (as explained in step 1 to 5 above). The working of formulae in Equation 3 is shown in Figure 8.6 that explains how the algorithm explores path from vertex-1 (i.e. h=1)

until stopping condition is met. In a similar way, edgepath vector E_{142} and E_{143} can be recursively explored.

Any loop formed at any level of path tree will not be explored while generating next level edgepaths because logical AND operation for such a case will always generate output bit vector containing all 0's. The complement of M_j makes sure that the immediate vertex j, just traversed from a given source vertex is not added again in path tree while generating subsequent levels from that source vertex. In generalized formulae, the subscript (i….j) represents all the vertices to which the existing path from the starting vertex have been counted. The proof of general formulae for all-path lengths $\forall N$, $N \geq 3$, has explained in theorem 1 below.

Theorem 1. The N-length path $E_{h, i\ldots j, k}$ in a given graph $\forall_N$, $N \geq 3$ can be found by performing logical AND between the vertical bit vectors of E_k AND $M'_{i\ldots j}$ for $\forall_k$, where $k \in$ (N-1)-length List $E_{i\ldots j}$.

Proof: For N=1, 1-length path is obtained from the basic path vectors of E_k, $\forall k$, $k \in VL$, so 1-length paths are calculated from the adjacency matrix formed for every vertex to every other vertex and that's why there is no logical AND operations involved in calculating 1-length paths. For N=2, there is only one most recent visited vertex (in this case from source vertex), so logical AND is performed between E_k and M'_h, i…j (i…j for N=2 is empty, as there are no intermediate vertices other than source vertex, which is denoted by h). So, in other words, $M'_{h, i\ldots j}$ can be replaced by M'_h where (i…j) represents source vertex as the most recent visited vertex. The reason for choosing the complement of most recent vertex is to restrict visiting same vertices again.

When $N \geq 3$, there are at least 1 most recent visited vertex other than the source vertex. So, it is required to not visit them again because we do not want any loop back to the visited

vertices. So, it is required to compliment all the vertex masks of the visited vertices other than the source vertex and it is always going to be 1 less than the total number of visited vertices because source is counted as visited. Hence M'i…j =M'i &&M'i1 &&M'i2&&M'i3…..&&M'j $\forall$i … j, source vertex h$\notin$ (i … j).

The visited vertices are generated one after another in a sequential manner and are generated from the path trees at $(N-1)^{th}$ iteration. Lastly, the resulting path tree of $M'_{i\ldots j}$ is logically AND by E_k at N^{th} level, because at that level I want to visit all the vertices from E_k excluding those which have been already visited ($M'_{i\ldots j}$). Hence, this is the proof for Theorem 1.

End condition for algorithm: The end condition is not hard to determine. The algorithm stops when all the paths have been found and the condition which determines whether all the paths have been determined is the most recent path tree value of $E_{h,\, i\ldots j,\, k}$ from generalized formulae. If the path tree of all resulting $E_{h,\, i\ldots j,\, k}$ is $(0, 0, 0, 0)^T$ for all the vertices, that would tell that all the paths of various lengths have been found in the graph.

Theorem 2. The value 0 for all indexes in any edge mask $E_{h,i\ldots j,k}, \forall_h$, h$\in$VL, with h as source vertex shows that there does not exist any un-counted path.

Proof: The vertex V_i which has been visited from a starting source vertex V_h are not included again by calculating logical AND with complement of M'_i. That means for any next iteration, the algorithm will calculate the logical AND between compliments of vertex mask (M_k) of all in visited vertices list. This implies that when there is 0 value at all the indexes at edge mask path tree ($E_{h,\, i\ldots j,\, k}$), and thus the algorithm cannot go any iteration further as there does not exist any new non-visited vertex that can be reached from current vertex. Hence, the value of k will also be 0 and there will not be any edge mask for k equals 0. So, it acts as end condition for the iteration.

Space analysis of proposed algorithm: It can be seen from Figure 8.6 that the algorithm requires exponential amount of space. For a simple graph G1, there were total of 84 edgepath vectors (4 in level-1, 16 in level-2 and 64 in level-3) generated in MBVT. In general, the total number of edgepath vectors of length equal to VL for a MBVT having N levels can be calculated from the following equation (Eq. 4):

$$\#\ \textit{of edgepaths} = \sum_{i=1}^{N} (VL)^i, \quad \textit{Where VL is the vector length (i.e. 4 for G1)} \qquad \text{(Eq. 4)}$$

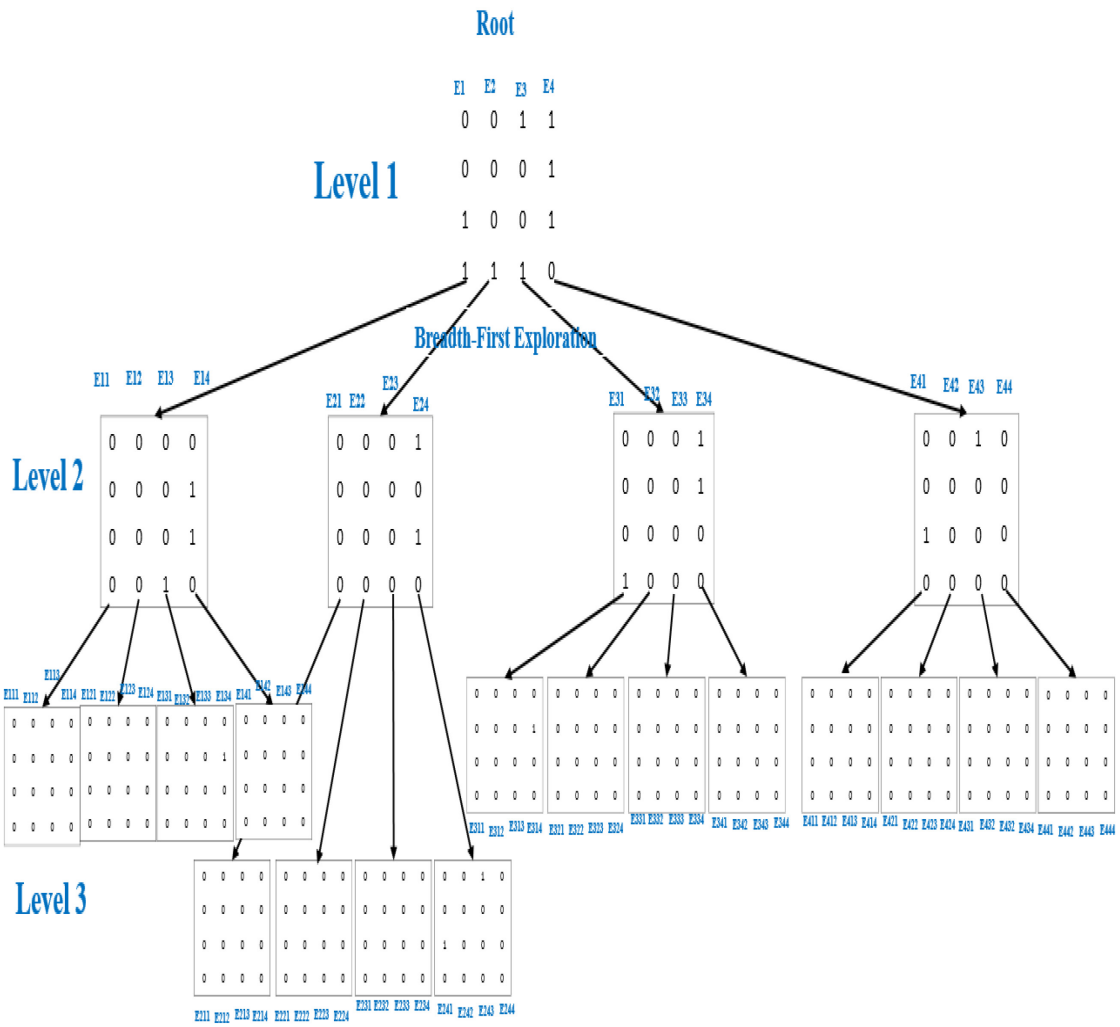

Figure 8.6. Generation of multi-level bit vector tree (MBVT)

It is also seen in MBVT that a lot of computation is involved that results in zero edgepath vector (i.e. E $(0000)^T$). It is also seen that a vertex having a value 0 in an edgepath never appear

in subsequent paths, so, it could be excluded from exploration to save space. For example, E_1 $(0011)^T$ has value 0 for vertex 1 and vertex 2. To check all the paths generated from E_1, it is seen that edgepath vectors for E_{11} and E_{12} are zero and also, their subsequent levels also results in zero edgepath vectors. So, this unnecessary computation could be avoided to enable faster generation of all-paths and lookup during path search.

It is very important to present a suitable mechanism to make efficient use of space. I developed an algorithm that is another novel contribution to convert this exponential space requirement into linear space using indexing. Next section discusses generation of indexes and then conversion of these indexes to generate path labels.

8.5. Finding Shortest Paths

The shortest path is found from the MBVT generated earlier. The MBVT is one-time process and has all the paths generated from each starting vertex. The addition or removal of any edge in the graph can be updated within the MBVT generated for all-paths (discussed in section 8.4). The approach to find shortest paths start from the top level (root or level-0) of MBVT and continues until the path is found or the leaf edgepath vector is reached. The root of the graph G1 has the edgepath vector E $(1111)^T$ at level-0 (see Figure 8.7) and has 3 levels. Search of shortest path through each edgepath that was generated in MBVT is computationally expensive. As discussed earlier, zero edgepath vectors add computational and space complexity, so, I present only useful edgepath vectors in Figure 8.7. As discussed earlier, the MBVT requires exponential amount of space. So, this section also discusses our novel approach to convert exponential amount of space into linear amount of space. This is achieved through generation of indexes to keep track of paths. The index generation is discussed in next sub-section.

Generation of index list: The values at level-0 show that there are 1-length paths that exists from all the vertices i.e. there is no isolated vertex. In order to generate index list, the index of 1's in all the edgepaths in level-1 are important to be known to generate subsequent level of MBVT. The following steps explains the generation of index list.

Step 1: The index list at level-1 is formed by searching for the value of 1 in each edgepath vector. The index list is generated by scanning edgepaths in column-raster scan i.e. E_1 is scanned first and the indexes 3 and 4 are stored in the list, followed by a scan of E_2 edgepath vector. The index list of level-1 is shown in Figure 8.7.

Step 2: Indexes of next level edgepath vectors are only included in index list if their logical operation results in non-zero-bit vector. The advantage of storing indexes into a list is that it ensures efficient and quick search through the path tree.

Step 3: It is very easy to generate vertical bit vector from index value (as shown in Figure 8.7). For example, at level-1 the edgepath E1 is $(0011)^T$ and adds value 3 and 4 to index list. Similarly, at level-2 the edgepath E_{13} is $(0001)^T$ and its corresponding index value is 28 [first 16 corresponds to level-1 + 4 for E_{11} + 4 for E_{12} +4 (because 1 is present at 4th bit of E_{13})]. E_{11} and E_{12} have zero-bit vectors and that's why their index value does not exist in index list. In this way, the indices are generated at each level of MBVT.

Step 4: The highest index value for level-2 is 80 (because 16 for level-1 and 64 for level-2). The highest value of an index at level-3 will be 336 i.e. 80 for up to level-2 and 256 for level-3. So, the general formulae for highest number of an index at a given level L can be calculated by the following equation (Eq. 5):

$$\textbf{Highest index \#} = \sum_{i=1}^{L} (VL)^{i+1} \qquad \text{(Eq. 5)}$$

Using this formula, the highest # of index in level-3 is 1360 (i.e. 16+64+256+1024= 1360). It is worth to note that the formulae above just give the highest possible value of index in the index list for a given level, whereas only those indexes are stored in index list where there was a 1 in the edgepath vector (see index list in Figure 8.7).

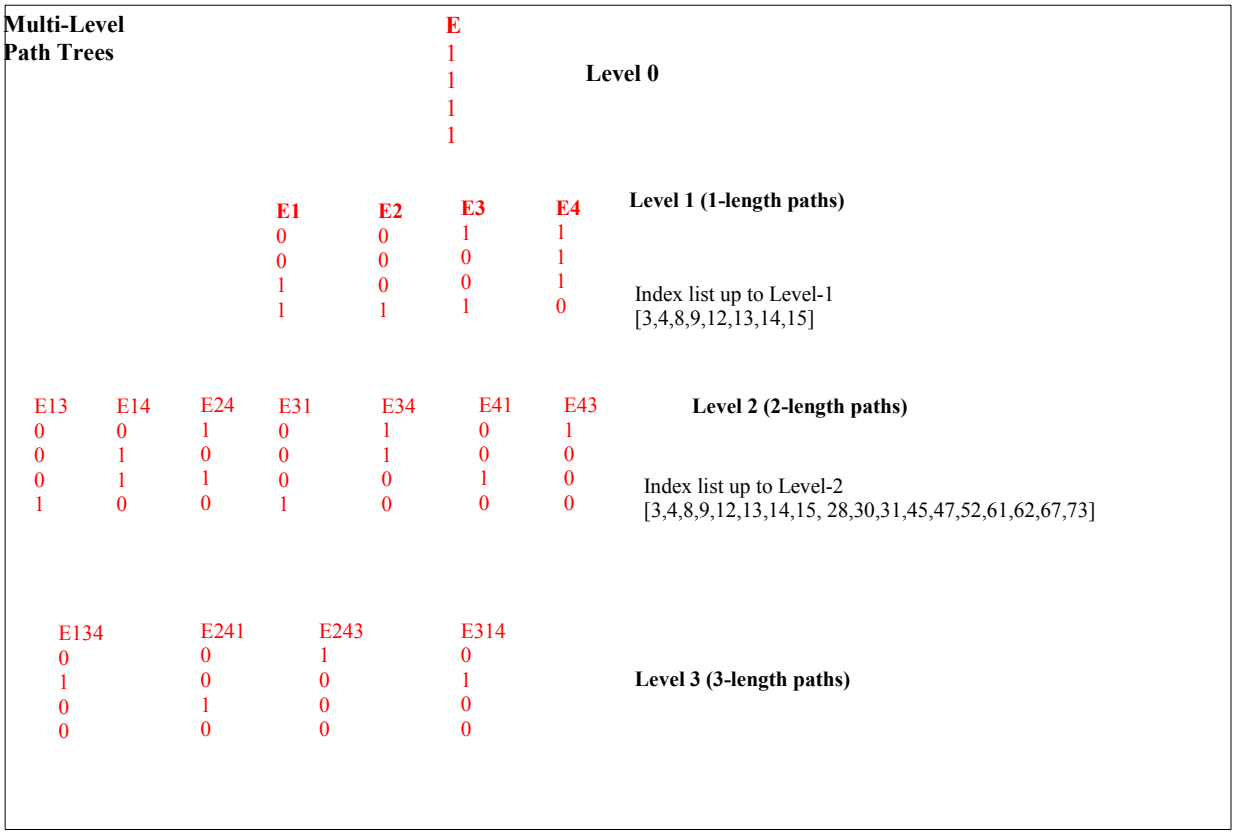

Figure 8.7. Multilevel path trees for graph G1

Observation: The MBVT is very sparse tree with only a few 1s. So, storing only the values of 1 in index list would decrease the exponential space requirements to linear space requirements. The shortest paths are searched using the index list and the algorithm is explained in next sub-section.

Shortest path search algorithm: The index list generated for MBVT can retrieve the edgepaths from the list values to search for any existing shortest path. The conversion from index list to edgepaths is very simple and is explained in this sub-section.

Conversion from index list to edgepaths: The edgepaths E_k from index value are retrieved through simple mathematics. For example, consider an index value of 45 for graph having a Vector Length (VL) of 4. The conversion takes place in following steps:

Step 1: Initially the index value is processed to retrieve the level of MBVT that this index value refers to. This is simply checked using the formulae defined to calculate highest index number i.e. $\sum_{i=1}^{L}(VL)^{i+1}$.

Step 2*:** The formulae when tested on index value 45, retrieves level-2 for this index value. This is calculated by checking for range of each level based on VL. The level-1 lies between 1 and 16 (VL^2), level-2 lies between 17 and 80=16 + 64 (VL^2 +VL^3) and so on. Clearly, the index value falls into level-2; so, ***45 is 29th place in level-2.

Step 3: Once, the information about the level of index value is known then it is very easy to compute intermediate path for this index value. Next, the formulae to generate intermediate path is explained as follows:

1. ***Value of k:*** 29 modulus (%) VL gives k (i.e. 1).
2. ***Value of h:*** (29 / $VL^{currentLevel}$) + 1 gives h i.e. 2. [value of current level is 2]
3. ***Value of intermediate path***: It is calculated as [(29%$VL^{currentLevel}$)/ $VL^{currentLevel-1}$ +1] outputs 4. Here the value of current level is 2 and It is recursively iterated until value of level is 1.

The outcome path for index value 45 is E_h, intermediate, k i.e. E (2→4→1). Let's take another example with an index value of 126 that is explained in next sub-section (i.e. E_{1342} in Figure 8.6).

Generalizability of our index conversion algorithm: The generalizability of step 2 and step 3 above for any level of MBVT is explained by taking an example of index value 126

corresponding to edgepath vector E_{134} $(0100)^T$ for graph G1. In other words, this edgepath vector shows the path 1→3→4→2. The steps 2 and 3 should be able to retrieve this path and the working is described as follows:

- As mentioned in step 2, the first requirement is to calculate the level for this value. This is the edgepath vector of graph G1, so this has VL of 4. Using the formulae to find highest index # in a given level L, the highest level for index value 126 is 3. The index value 126 is 46^{th} position in level-3 (see Figure 8.6 to check graph).
- **Value of k**: this is calculated by computing the modulus of 46 with VL, and this gives 2. So, the k=2 in this index case.
- **Value of h**: This is calculated by the formulae [(index_position_in_current_level / $VL^{currentLevel}$) + 1] i.e. [(46/VL^3) + 1] is equals to 1. The value of h is calculated to be 1.
- **Value of intermediate path**: In this case of index value, the intermediate path contains two nodes (i.e. 3 and 4 for E_{1342}). In order to be able to find these two intermediate paths, our algorithm should be able iterate until a stopping condition is reached. The path is iterated until the value of currentlevel becomes 1 in the equation [(index_position_in_current_level % $VL^{currentLevel}$) / $VL^{currentLevel-1}$ +1].
    - **Iteration 1:** The value of currentlevel variable is 3 for the index value 126 and the formulae is calculated as [((46%VL^3)/VL^2) + 1], where VL is 4 for graph G1. This iteration results in the intermediate path value of 3. The value of currentlevel is decrement by 1 after every iteration. So, the value of currentlevel becomes 2 after this iteration and would execute the formulae recursively until it becomes 1.
    - **Iteration 2:** The value of currentlevel is 2, so the formulae is calculated as [((46%VL^2)/VL^1) + 1], and this iteration results in 4. The value of currentlevel

becomes 1 and the algorithm comes out of loop with two values as output i.e. 3 and 4. This was our desired intermediate path.

The formulae described in step 2 and 3, was able to generate intermediate path for different levels. Hence, the algorithm can reproduce the path from the index value. The path is $E_{h,\ intermediate\_path,\ k}$ and this becomes E_{1342} from the values of h and k calculated above.

Explanation of shortest path search with an example: The strength of our MBVT approach is the tree traversal approach using the index lists. In a normal tree traversal approach, a shortest path from a vertex h to k (where h, k $\in$VL) is searched within the multi-level path tree from $E_{\mathbf{h}}$ until first ***k*** appears.

- ***Shortest path search from MBVT*:** To find shortest path from h=1 to k=2 in graph G1 (see path tree in Figure 8.6), level 1 is checked at index h=1 to see if there is a value 1 at index 2 in edgepath vector E_1 $(0011)^T$. There is 0 at index value 2 in this edgepath and it means that there is no direct shortest path from 1 to 2. So, the next level is checked i.e. level 2 within multilevel path tree which represents paths from E_h until I get 1 at index 2. As seen at this level I get 1 at index 2 in level 2 for $E_{1,\ 4}$. So the shortest path is 1→4→2. The shortest path can be very easily looked up by traversing the multilevel tree level by level until I get 1 at the desired index when starting from source vertex bit vector.
- ***Shortest path search from index list*:** The shortest path can be searched from the index list. For example, the path 1→4→2 can be searched from the index values. In this example, the path 1→4→2 can be expressed as E_{142}, which is equivalent of $E_{h,\ intermediate\_path,\ k}$. Here, the h is 1, intermediate path is 4 and k is 2. The shortest path can be checked, first of all, for the value of k using the formulae defined for conversion from index values. If the value of k is 2, then the value of h should be calculated for that index

value. The value of intermediate path should only be explored if h and k have been obtained. The algorithm to find shortest path from index values is shown in Figure 8.8. The algorithm takes, index list, source vertex, destination vertex, and vector length as arguments.

```
ShortestPath (int Indexes, int VL, int source, int dest) {
For each value in Indexes:
Calculate currentlevel from the index
Calculate K from value % VL
 IF K equals dest THEN
    Calculate H from value / VL^currentlevel +1
     IF H equals source THEN
        Calculate intermediatePath (int VL, int currentlevel, int index)
        Print Shortest Path // i.e. H, intermediatePath and K
     ELSE break
 ELSE break
Read Next value from Indexes
END For
}
```

Figure 8.8. Algorithm to search shortest paths from index values

Graph data sets used in this study: There are three data sets that have been taken in this study to perform analysis (see Figure 8.9) and called them graph 1 (G1), graph 2 (G2) and graph 3 (G3). All these graphs had different number of vertices and edges. These graphs were selected to study the execution time effect by taking simple graphs with different nodes and interconnecting edges.

The three graphs are selected based on a linear increase in number of nodes. These graphs are shown in Figure 8.9. Another motivation was to test our algorithm on smaller graph to study and explore efficient methods that could be analyzed manually to validate the results. I tested my algorithm over graphs in which some of the nodes are not reachable (e.g. graph G3, nodes 3 and 4). It was also intended that if promising results are found then our next step would be to use

bigger graphs and execute our algorithm using distributed processing. Some big social network graphs are also next in line to be explored using our proposed algorithm.

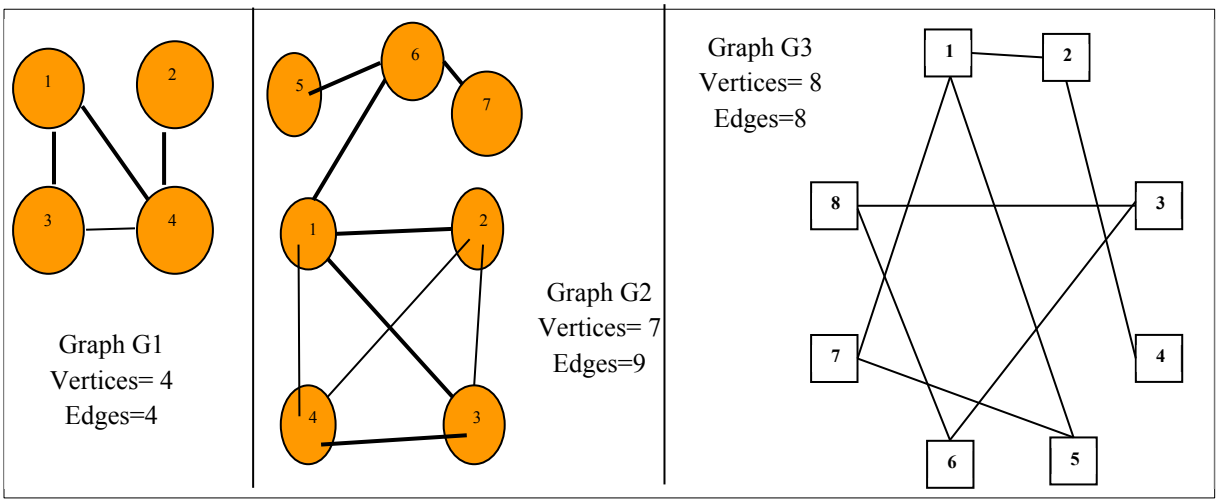

Figure 8.9. Various graph data sets used in this study

8.6. Results and Discussions

Multilevel bit vector tree (MBVT) algorithm requires exponential amount of space to store all the paths that are generated. This proposed approach also presents a way to convert exponential space requirement into linear space by storing paths in the form of indexes. The results and discussion in this section is presented using shortest path search using index list. The search for shortest path using index list is faster because very few comparisons are needed. In our proposed work, MBVT was generated to find all the paths in a graph and index list was used to search for the shortest path.

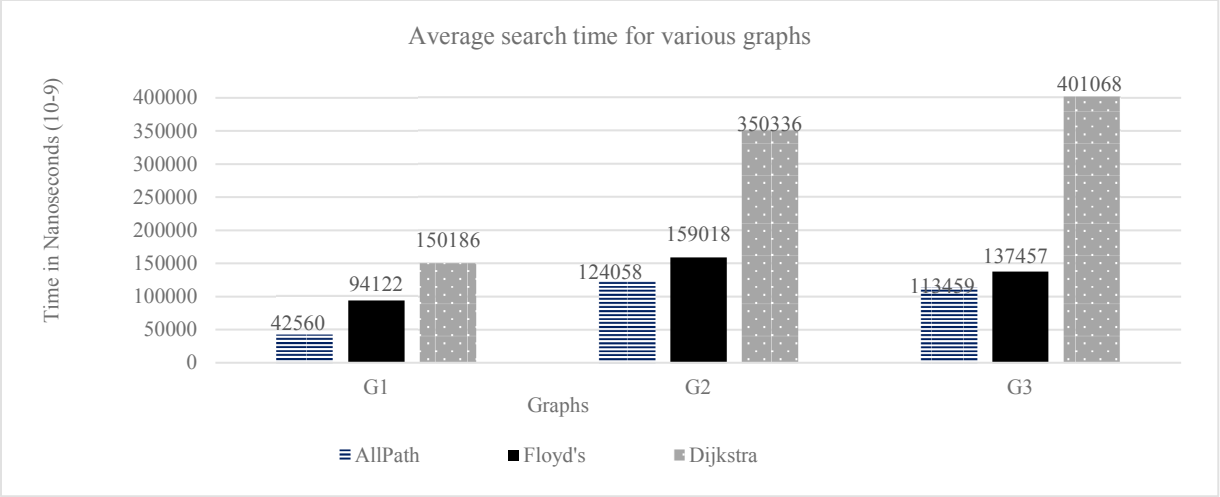

Figure 8.10. Results on average search time for various graphs

This section presents discussion on time taken by our proposed algorithm (Allpath/ MBVT) to search shortest path between source and destination vertices (selected at random during runtime). This section also compares results of shortest path search time (Figure 8.10) and execution time (Figure 8.11) taken by Allpath, Dijkstra and Floyd's algorithm for three graphs namely G1, G2 and G3. This section also compares results of shortest path search time (Figure 8.10) and execution time (Figure 8.11) taken by algorithms in Nano-seconds.

Experiment details: The experiment was run on intel-core-i7 processor with main memory of 4 gigabyte and CPU clock of 2.4 GHz. The algorithms are executed in java programming environment through Eclipse IDE. The experiment was run 100 times to avoid any researcher's bias to obtain more precise and accurate results.

Performance results: The search time (efficiency) for finding shortest path from source to destination vertex was repeated 100 times and the average is plotted in graph shown in Figure 8.10. The source and destination vertices were also selected at random during each experiment run. The execution time (Figure 8.11) for all three algorithms was also repeated 100 times and their average is taken as final execution time. It is seen from Figure 8.10 that Allpath algorithm

outperforms Floyd's and Dijkstra for all three graphs w.r.t search time. Allpath performed best while Dijkstra performed least w.r.t search time. Floyd's algorithm slightly underperformed Allpath. On the contrary in Figure 8.11, Allpath performed exponential w.r.t overall execution time to generate multilevel path vectors. The execution time approximately becomes twice when number of vertices are doubled in a graph (Figure 8.10).

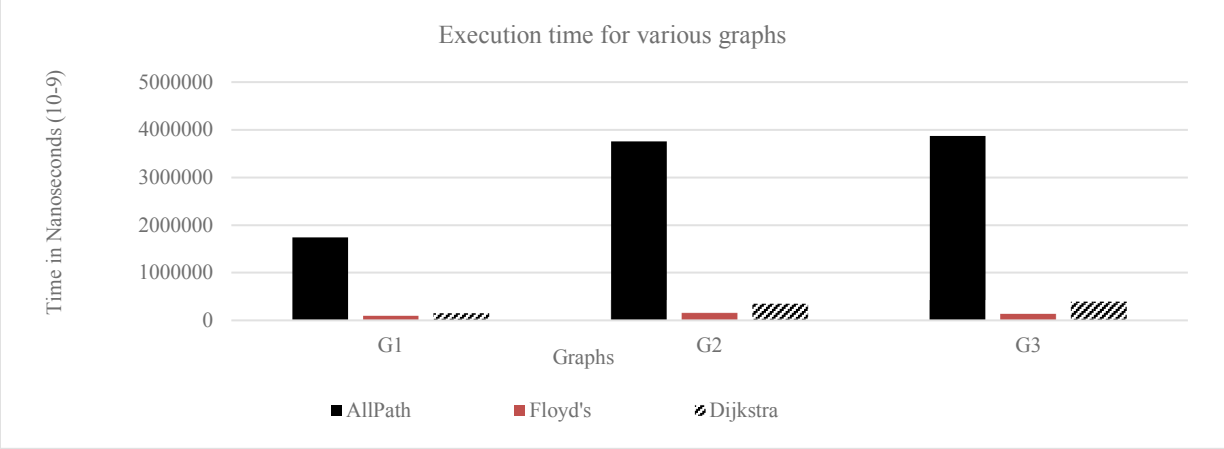

Figure 8.11. Results on overall execution time for all three graphs

Discussion of results: The results of Allpath algorithm can be better understood by exploring its complexity. The result showed significant improvement in shortest path search time (Figure 8.10). Therefore, the analysis of comparisons needed to find shortest paths is performed in Table 8.I. In this section, some theoretical discussion on path update are presented for Allpath (multilevel) algorithm. We present analysis of our Allpath (multilevel) algorithm in terms of number of comparisons with Dijkstra's algorithm (undirected edges and equal weights), Floyd's algorithm. The Table 8.1 shows worst case comparisons between these algorithms.

Symbols used to analyze complexity: In Table 8.1, k is the tree width, r is number of nodes, n is number of edges, M is maximum number of bit vectors present and L is total number of levels in multilevel path tree. The analysis of the result is discussed in next section alongside advantages of index-based solution to our MBVT approach. From the results it can be inferred

that multilevel path tree algorithm is slower at tree generation step while it is faster when it comes to search or update an edge in a graph.

Allpath update complexity: Path update in worst case assumes that if there are maximum possible number of bit vectors (M) at last level of vector length (VL) are present; then based on bit vectors at last level, our multilevel path tree algorithm has maximum of $\log_{VL} M$ levels (i.e. $L= \log_{VL} M$). For example, in a graph with VL=4 and having maximum of 256 total # of bit vectors (M) at last level, then the total # of levels (L) are 4 (i.e. $\log_4 256$). The analysis of complexity has been presented in Table 8.1. If n edges have been updated at each level, then the upper bound of # of updates required at a specific level for the whole graph are ($n \log_{VL} M$).

Table 8.1. Number of comparisons

Algorithms	**Algorithm Operations**	**Comparison for edge Update**
Dijkstra's (single source)	$O (k^2 \log r \log (k \log r))$	$O (k^2 \log r \log (k \log r))$
Floyd's	$O (r^3)$	$O (r^3)$
Allpath (at any level)	O (m x VL)	$O (n \log_{VL} M)$

Allpath complexity: In a similar way, to calculate the complexity of the MBVT algorithm in a worst-case scenario is calculated using number of bit vectors and number of 1s at a given level. Assuming there are m number of bit vectors and each bit vector has VL number of 1s then there are (m x VL) number of bit vectors that are generated at next level. It is also worth noting that if $m \approx VL$ at each level then the algorithm runs in exponential time but it is one time process and new edges can be dynamically added or removed without computing the whole tree again. The shortest path search algorithm described earlier (Figure 8.8) using indexes has resulted in shortest path calculation even faster in quadratic amount of time and expects that this algorithm can further be improved by using distributed processing (subject to evaluations).

Allpath algorithm generates new bit vectors for subsequent level on each occurrence of 1. This makes Allpath algorithm exponential in execution. That's why the execution time taken by Allpath algorithm is more than other two algorithms. The advantages of our proposed algorithm despite exponential complexity is as follows:

- It is very convenient to be executed in distributed computing because bit vectors at root node can be processed independent of each other.
- Generation of multilevel path vectors are one-time process and results in speed gain for shortest path search.
- Path update is easy to accommodate without repeating whole process.

Multilevel tree algorithm has proven to be the superior algorithm (among existing ones) to find the shortest paths with forbidden paths and without forbidden paths. In [83], an algorithm to find multiple shortest paths (k-shortest paths) is presented [83], which (multiple shortest paths) our algorithm is also generating by generating all lengths of paths. Newman in [84] presented an algorithm to find all the collaboration of authors in scientific papers in various fields. Newman described his technique to find shortest path between two authors in his collaboration network using Queues.

Our multilevel path algorithm also finds shortest path between two authors in relatively faster manner with the use of vertical data structures because vertical data structure is best fit for logical operations to perform. Restricted Shortest Path (RSP), $\in$-Approximation, backward-forward heuristic and Lagrangian based methods have major issues of large computational time [85] while our multilevel path algorithm's computation time is less expensive using the logical operations, which can further be reduced to *the factor of O (n* $\log_{VL}$ *M) by the use of indexes* to represent position of 1's in multilevel trees. When indexes are used then I need not to traverse

throughout the bit vector to check the presence of 1's and this makes algorithm even more efficient by skipping unnecessary computations.

9. RESEARCH CONTRIBUTIONS

This section discusses the major contribution of the work described in this dissertation to requirement inspections and research. This section also enlists the publications that will be used to disseminate the work done during the dissertation.

9.1. Contribution to Requirement Inspections and Research

This research illustrated that manual inspection tasks that are cost and time intensive can be automated using state-of-the art ML and graph mining approaches. This dissertation produced techniques that can automate the selection of inspectors pre inspection and analysis of data collected during the inspection (i.e., validation of inspection reviews, identification of fault-prone areas in an SRS document, and analysis of impact of a change during fault fixation). When used in industrial settings, this will improve software development efficiency by providing automated support for requirements author and inspection managers. While ML has been applied in other domains, to the best of my knowledge, this is one of the earliest efforts at using theories grounded in machine learning to develop research solutions that can help improve quality of NL requirements document.

Some of the research contributions include an improved understanding of features of various ML algorithms that will help other researchers when using these approaches in RE or similar domains. Specifically, this dissertation provides useful insights into:

- feature set threshold when extracting most informative features from NL reviews,
- mechanism for training classifiers using reviews from semantically similar domain;
- prominent POS tags and ensemble classifiers most applicable to NL fault logs and NL requirements;
- using eye tracking data to predict the characteristics of most skilled inspectors;

While the approaches proposed and validated in this dissertation (e.g., KESRI) were validated w.r.t NL requirements, they can be applied to extract pertinent information from any unrestricted NL text (i.e., without needing any conformance standards). This appeals to the wide application of this research work beyond software engineering domain. It is hoped that this work will motivate other researchers to employ automated ML approach in software engineering discipline for advancing the SE practice and to improve the quality of requirements produced during the development.

In particular, the application of graph mining approaches to find cliques and K-plex to when using LSA, LDA and other graph mining approaches can help requirements authors visualize interrelated requirements, make appropriate changes and assess the quality of the fixes.

9.2. Publication and Dissemination

This section describes the publications that resulted from the work done for this dissertation. The publication plan is described in terms of articles that have been published, under review, and being prepared for submission.

9.2.1. Refereed Conferences

1. **Singh, M.,** Walia, G., and Goswami, A. "Validation of Inspection Reviews over Variable Features Set Threshold", IEEE International Conference on Machine Learning and Data Science (ICMLDS-2017), Dec 14-15 Dec, India.
2. **Singh, M.,** Walia, G., and Goswami, A. "An Empirical Investigation to Overcome Class-imbalance in Inspection Reviews", IEEE International Conference on Machine Learning and Data Science (ICMLDS-2017), Dec 14-15 Dec, India.
3. **Singh, M.,** Vaibhav, A., Walia, G., and Goswami, A. "Validating Requirements Reviews by Introducing Fault-Type Level Granularity: A Machine Learning Approach", ACM

SIGSOFT Innovations in Software Engineering (ISEC 2018), Feb 9-11, Hyderabad, India.

4. **Singh, M.,** Vaibhav, A., Walia, G."A Vertical Breadth-First Multilevel Path Algorithm to Find All Paths in a Graph", International Symposium on Big Data Management and Analytics (BIDMA 2018), April 24-25, Calgary, Canada.
5. **Singh, M.,** "Automated Validation of Requirement Reviews: A Machine Learning Approach", 26th IEEE International Requirement Engineering Conference (RE '18), August 20-24, 2018, Banff Canada.
6. **Singh, M.**, Walia, G., Goswami, A., "Using Supervised Learning to Guide the Selection of Software Inspectors in Industry", In proceedings of 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018), October 15-18, Memphis, Tennessee, USA.
7. **Singh, M.,** "Semantic Analysis and Graph Mining Approach to Generate Inter-related Software Requirements", ACM sponsored 16th International Doctoral Symposium on Empirical Software Engineering (IDoESE 2018), October 10, Oulu, Finland.

9.2.2. Future (Journal/Conference) Publications

1. **Singh, M.,** Walia, G.S, “Using Keyword Extraction from Inspection Logs to Guide Post Inspection Fault Fixation in Software Requirements”. (to be submitted)
2. **Singh, M.,** Walia, G.S, “Mapping Fault Logs from Software Inspections to Fault Prone Requirements in an SRS Document”. (to be submitted)

10. CONCLUSION AND FUTURE SCOPE

This dissertation employed various ML approaches (both supervised and unsupervised), NLP approaches, and graph mining approaches to automate pre and post inspection activities. This chapter discusses the major implications of this work and presents some future directions.

The software development is a process driven approach that starts with the identification of requirements about the system to be designed and build. To develop a reliable software system, it must be inspected regularly to find and fix faults, which is a tedious and time-consuming activity. The faults that propagate to the later stages are hard to find and costlier to fix. So, it is crucial to find and fix faults during the early phase of software development (i.e., requirements). Inspections are employed to find faults, where skilled inspectors inspect the SRS document and generate the fault logs, which are then handed back to requirements author to fix. The requirements author manually identifies the true fault log before fixing it in the SRS document, which is time-consuming and tedious process.

This dissertation attempted to automate certain pre and post-inspection activities by employing ML approaches. These include using ML to validate true faults versus false positives, identification of faulty requirements from validated true faults (using semantic similarity measures) and assessing the impact of a change (using graph mining approaches) in a requirement during fault fixation.

I successfully implemented ML, NLP, and graph mining algorithms to automate the post-inspection activities and selection of these algorithms was guided based on their successful implementation in other domains of software engineering (e.g., design, testing, text processing etc.). The results in this dissertation validated my approaches and identified some important

insights that will guide future research directions. These important insights and the future scope are discussed for each research goal identified in chapter 1.

10.1. Goal 1 - Validation of Requirement Reviews

Applying various ML algorithms, leveraging NLP algorithms, and employing POS tags; this research showed that useful NL reviews can be automatically detected thereby saving costs that are otherwise spent manually identifying true faults (and false positives). ML algorithms when using POS tags (nouns and adjectives) performed best. Priority classes (i.e., validation prediction through voting) were employed to successfully classify NL reviews.

The future scope of this research is to investigate the impact of deep learning algorithms for validating requirement reviews. More replications of this work using different SRSs across multiple studies will help generalize the results.

10.2. Goal 2 - Finding Fault Prone Requirements

To automate the information extraction from inspection fault logs, three types of KPE algorithms (supervised feature based, unsupervised graph based, and unsupervised statistical) were used. The results showed that supervised learning feature-based algorithms performed best at extracting keyphrases of length 3-words. The researchers can use the KESRI approach to extract any number of keyphrases (of any length) in NL text e.g., extracting bug information from commit messages during mining software repositories.

The MOKSA approach identified most applicable clustering algorithm (nearest neighbor vs. graph based nearest neighbor) to cluster keyphrases, and the most applicable semantic similarity measure (LSA, LDA) to map clusters of keyphrases to requirements in an SRS document. The results found that Affinity propagation (a graph based nearest neighbor) approach

is best suited to cluster keyphrases, and LDA algorithm is best suited to map keyphrases to requirements in an SRS.

The studies were done using artificially seeded faults and researchers knew the master fault list. To be able to apply KESRI and MOKSA in industry, it will need to be validated with artifacts with naturally occurring faults and in a live setting. This is an immediate next step at improving the keyphrase extraction approach presented in this dissertation.

10.3. Goal 3 - Guiding the Change Impact Analysis

This research aimed at generation of interrelated requirements (IRRs) for an SRS document, such that pre-inspection, the requirements author could identify term substitution errors, and redundant requirements, while post inspection, the authors could use IRRs to guide CIA during fixing a requirement. This research reported that the IRRs (generated using LDA) are the best representative of actual related requirements. This research also reported certain graph mining techniques (cliques and K-Plex) can reported highly impacted requirements because of a change during a fault fixation. This research can give insights to the SE researchers identify redundant requirements, redundant errors in software documents.

The immediate future work is to evaluate my proposed approach using other semantic and syntactic algorithms (e.g. RP, cosine similarity etc.). Furthermore, another study as part of the future work is to evaluate ripple effect of CIA using N-hop path algorithms by applying interactive change/fixations in the requirements.

10.4. Goal 4 - Selection of Inspectors

This research investigated the characteristics of an effective inspector that could report faults with significant precision while inspecting an SRS document using eye-tracking equipment. This research also investigated the best suited ML algorithms and feature evaluator

methods to identify inspector's characteristics. The results reported that the best inspector characteristics involves the reading patterns (i.e., linear saccades), fixations on SRS document, and the average time spent on each SRS page during inspection.

The future work includes the replication of this study with different inspectors (i.e., from industry), with additional classification families like Neural Networks, when inspecting different requirements documents, and with varying experiences. Also, I plan to expand the number of attribute evaluation methods in future replications.

10.5. Application to Other Domain

This dissertation presented the novel application of ML algorithm to develop a data structure that can support faster mining of large graph data using parallel processing. The other advantage of this application is that it can support dynamic update of data without having to perform re-computations (which is a desirable characteristic to support distributed processing).

The long-term benefit of this research is to generate all existing paths in a graph of requirements to evaluate CIA.

REFERENCES

[1] A. Goswami, G. Walia, M. McCourt, and G. Padmanabhan, "Using Eye Tracking to Investigate Reading Patterns and Learning Styles of Software Requirement Inspectors to Enhance Inspection Team Outcome," *Proc. 10th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas. - ESEM '16*, vol. 08–09–Sept, pp. 1–10, 2016.

[2] A. Goswami and G. Walia, "An empirical study of the effect of learning styles on the faults found during the software requirements inspection," *2013 IEEE 24th Int. Symp. Softw. Reliab. Eng.*, pp. 330–339, 2013.

[3] A. Goswami, G. S. Walia, and U. Rathod, "Using Learning Styles to Staff and Improve Software Inspection Team Performance," *Proc. - 2016 IEEE 27th Int. Symp. Softw. Reliab. Eng. Work. ISSREW 2016*, pp. 9–12, 2016.

[4] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at Microsoft," *IEEE Int. Work. Conf. Min. Softw. Repos.*, vol. 2015–Augus, pp. 146–156, 2015.

[5] N. Chen, J. Lin, S. C. H. Hoi, X. Xiao, and B. Zhang, "AR-miner: mining informative reviews for developers from mobile app marketplace," *Proc. 36th Int. Conf. Softw. Eng. - ICSE 2014*, pp. 767–778, 2014.

[6] A. Agarwal, B. Xie, I. Vovsha, O. Rambow, and R. Passonneau, "Sentiment analysis of Twitter data," *Assoc. Comput. Linguist.*, pp. 30–38, 2011.

[7] K. Gimpel *et al.*, "Part-of-Speech Tagging for Twitter: Annotation, Features, and Experiments," *Proc. 49th Annu. Meet. Assoc. Comput. Linguist. Shortpapers*, no. 2, pp. 42–47, 2011.

[8] S. R. El-beltagy and A. Rafea, “KP-Miner : Participation in SemEval-2,” no. July, pp. 190–193, 2010.

[9] A. Bougouin, F. Boudin, and B. Daille, “TopicRank: Graph-Based Topic Ranking for Keyphrase Extraction,” *Proc. Sixth Int. Jt. Conf. Nat. Lang. Process.*, no. October, pp. 543–551, 2013.

[10] C. Florescu and C. Caragea, “PositionRank: An Unsupervised Approach to Keyphrase Extraction from Scholarly Documents,” pp. 1105–1115, 2017.

[11] K. Bennani-Smires, C. Musat, A. Hossmann, M. Baeriswyl, and M. Jaggi, “Simple Unsupervised Keyphrase Extraction using Sentence Embeddings,” 2018.

[12] T. K. Landauer, P. W. Foltz, and D. Laham, “An Introduction to Latent Semantic Analysis,” *Discourse Process.*, vol. 25, no. 2–3, pp. 259–284, 1998.

[13] S. T. Dumais, “Latent Semantic Analysis,” *Annu. Rev. Inf. Sci. Technol.*, vol. 38, pp. 1–7, 2005.

[14] H. K. Dam and a Ghose, “Automated Change Impact Analysis for Agent Systems,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 33–42, 2011.

[15] A. . Goswami, G. . Walia, and A. . Singh, “Using learning styles of software professionals to improve their inspection team performance,” *Proc. Int. Conf. Softw. Eng. Knowl. Eng. SEKE*, vol. 2015–Janua, pp. 680–685, 2015.

[16] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, “Automated Extraction and Clustering of Requirements Glossary Terms,” *IEEE Trans. Softw. Eng.*, vol. 43, no. 10, pp. 918–945, 2017.

[17] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, “Software traceability with topic modeling,” *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng. - ICSE ’10*, vol. 1, p. 95, 2010.

[18] B. Dit, D. Poshyvanyk, and A. Marcus, “Measuring the semantic similarity of comments in bug reports,” *Proc. 1st STSM*, p. 6, 2008.

[19] L. V. G. Carreno and K. Winbladh, “Analysis of user comments: An approach for software requirements evolution,” *Proc. - Int. Conf. Softw. Eng.*, pp. 582–591, 2013.

[20] J. Han, M. Kamber, and J. Pei, “Chapter 9: Graph Mining, Social Network Analysis, and Multirelational Data Mining,” *Data Min. concepts Tech.*, pp. 535–589, 2006.

[21] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects,” *Proc. 11th Work. Conf. Min. Softw. Repos. - MSR 2014*, pp. 192–201, 2014.

[22] L. C. Briand, “Novel Applications of Machine Learning in Software Testing,” *2008 Eighth Int. Conf. Qual. Softw.*, pp. 3–10, 2008.

[23] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, “Design pattern mining enhanced by machine learning,” *Icsm 2005 Proc. 21st Ieee Int. Conf. Softw. Maint.*, pp. 295–304, 2005.

[24] P. Tonella and G. Antoniol, “Object Oriented Design Pattern Inference,” *Softw. Maintenance, 1999. (ICSM ’99) Proceedings. IEEE Int. Conf.*, no. November 1998, pp. 230–238, 1999.

[25] C. Bird and A. Bacchelli, “Expectations, Outcomes, and Challenges of Modern Code Review,” pp. 712–721, 2013.

[26] M. A. Hall, “Correlation-based Feature Selection for Machine Learning,” The University of Waikato, 1998.

[27] N. Hariri, C. Castro-Herrera, M. Mirakhorli, J. Cleland-Huang, and B. Mobasher, "Supporting domain analysis through mining and recommending features from online product listings," *IEEE Trans. Softw. Eng.*, vol. 39, no. 12, pp. 1736–1752, 2013.

[28] A. A. Akinyelu and A. O. Adewumi, "Classification of phishing email using random forest machine learning technique," *J. Appl. Math.*, vol. 2014, 2014.

[29] X. Carreras and L. Marquez, "Boosting Trees for Anti-Spam Email Filtering," 2001.

[30] S. Tong and D. Koller, "Support Vector Machine Active Learning with Applications to Text Classification," *J. Mach. Learn. Res.*, pp. 45–66, 2001.

[31] T. S. Guzella and W. M. Caminhas, "A review of machine learning approaches to Spam filtering," *Expert Syst. Appl.*, vol. 36, no. 7, pp. 10206–10222, 2009.

[32] S. Abu-Nimeh, D. Nappa, X. Wang, and S. Nair, "A comparison of machine learning techniques for phishing detection," *Proc. anti-phishing Work. groups 2nd Annu. eCrime Res. summit - eCrime '07*, pp. 60–69, 2007.

[33] E. Blanzieri and A. Bryl, "A survey of learning-based techniques of email spam filtering," *Artif. Intell. Rev.*, vol. 29, no. 1, pp. 63–92, 2008.

[34] J. Ng, D. Joshi, and S. M. Banik, "Applying Data Mining Techniques to Intrusion Detection," *2015 12th Int. Conf. Inf. Technol. - New Gener.*, pp. 800–801, 2015.

[35] S. Agrawal and J. Agrawal, "Survey on anomaly detection using data mining techniques," *Procedia Comput. Sci.*, vol. 60, no. 1, pp. 708–713, 2015.

[36] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[37] P. G. Liu, P. Gundecha, and H. Liu, “INFORMS Tutorials in Operations Research Mining Social Media : A Brief Introduction Mining Social Media : A Brief Introduction,” no. February 2015, 2014.

[38] M. Hu and B. Liu, “Mining and summarizing customer reviews,” *Proc. 2004 ACM SIGKDD Int. Conf. Knowl. Discov. data Min. - KDD ’04*, p. 168, 2004.

[39] P. Jain, K. Verma, A. Kass, and R. G. Vasquez, “Automated Review of Natural Language Requirements Documents: Generating Useful Warnings with User-extensible Glossaries Driving a Simple State Machine,” *Proc. {2Nd} India Softw. Eng. Conf.*, pp. 37–46, 2009.

[40] D. M. Blei, B. B. Edu, A. Y. Ng, A. S. Edu, M. I. Jordan, and J. B. Edu, “Latent Dirichlet Allocation,” *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, 2003.

[41] K. M. Hammouda, D. N. Matute, and M. S. Kamel, “CorePhrase: Keyphrase Extraction for Document Clustering,” pp. 265–274, 2010.

[42] B. Setiaji and F. W. Wibowo, “Chatbot Using a Knowledge in Database: Human-to-Machine Conversation Modeling,” *Proc. - Int. Conf. Intell. Syst. Model. Simulation, ISMS*, no. January 2016, pp. 72–77, 2017.

[43] V. . Anu, G. . Walia, W. . Hu, J. C. . Carver, and G. . Bradshaw, “Effectiveness of human error taxonomy during requirements inspection: An empirical investigation,” *Proc. Int. Conf. Softw. Eng. Knowl. Eng. SEKE*, vol. 2016–Janua, pp. 531–536, 2016.

[44] A. Goswami and G. Walia, “Teaching Software Requirements Inspections to Software Engineering Students through Practical Training and Reflection,” *2016 ASEE Annu. Conf. Expo. Proc.*, vol. 2016–June, 2016.

[45] P. P. Rane, “Automatic Generation of Test Cases for Agile using Natural Language Processing,” p. 97, 2017.

[46] M. Mukadam, C. Bird, and P. C. Rigby, “Gerrit software code review data from android,” *IEEE Int. Work. Conf. Min. Softw. Repos.*, pp. 45–48, 2013.

[47] S. S. So, S. D. Cha, T. J. Shimeall, and Y. R. Kwon, “An empirical evaluation of six methods to detect faults in software,” *Softw. Test. Verif. Reliab.*, vol. 12, no. 3, pp. 155–171, 2002.

[48] A. H. Yousef, “Extracting software static defect models using data mining,” *Ain Shams Eng. J.*, vol. 6, no. 1, pp. 133–144, 2014.

[49] A. Ghose and P. G. Ipeirotis, “Estimating the Helpfulness and Economic Impact of Product Reviews,” *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 10, pp. 1498–1512, 2011.

[50] G. Bavota, “Mining Unstructured Data in Software Repositories: Current and Future Trends,” *2016 IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, pp. 1–12, 2016.

[51] X. Yang, R. G. Kula, N. Yoshida, and H. Iida, “Mining the modern code review repositories,” *Proc. 13th Int. Work. Min. Softw. Repos. - MSR ’16*, pp. 460–463, 2016.

[52] R. S. Wahono, “A Systematic Literature Review of Software Defect Prediction : Research Trends , Datasets , Methods and Frameworks,” *J. Softw. Eng.*, vol. 1, no. 1, pp. 1–16, 2015.

[53] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami, “An Automatic Quality Evaluation for Natural Language Requirements,” *REFSQ 2001 Proc. Seventh Int. Work. RE Found. Softw. Qual.*, vol. 1, pp. 150–164, 2001.

[54] A. Demuth, R. Kretschmer, A. Egyed, and D. Maes, “Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: An experience report,” *Proc. - 2016 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2016*, pp. 529–538, 2017.

[55] J. D. Prusa, T. M. Khoshgoftaar, and A. Napolitano, “Using Feature Selection in Combination with Ensemble Learning Techniques to Improve Tweet Sentiment Classification Performance,” *2015 IEEE 27th Int. Conf. Tools with Artif. Intell.*, pp. 186–193, 2015.

[56] F. Angerer, A. Grimmer, H. Prahofer, and P. Grunbacher, “Configuration-Aware Change Impact Analysis (T),” *2015 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pp. 385–395, 2015.

[57] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, “Automated checking of conformance to requirements templates using natural language processing,” *IEEE Trans. Softw. Eng.*, vol. 41, no. 10, pp. 944–968, 2015.

[58] S. Lessmann, S. Member, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking Classification Models for Software Defect Prediction : A Proposed Framework and Novel Findings,” *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, 2008.

[59] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to detecting duplicate bug reports using natural language and execution information,” *Proc. 13th Int. Conf. Softw. Eng. - ICSE ’08*, p. 461, 2008.

[60] M. A. Hall, “Correlation-based Feature Selection for Machine Learning,” The University of Waikato, 1998.

[61] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer, “Improving requirements glossary construction via clustering,” *Proc. 8th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas. - ESEM ’14*, pp. 1–10, 2014.

[62] S. Singh, H. Murthy, and T. Gonsalves, “Feature Selection for Text Classification Based on Gini Coefficient of Inequality.,” *Fsdm*, pp. 76–85, 2010.

[63] R. Mihalcea and P. Tarau, “Textrank: Bringing order into text," *Proceedings of the 2004 conference on empirical methods in natural language processing*, 2004.

[64] X. Wan and J. Xiao, “CollabRank: towards a collaborative approach to single-document keyphrase extraction,” *Proc. 22nd Int. Conf. Comput. Linguist. Coling 2008*, no. August, pp. 969–976, 2008.

[65] F. Boudin, “Unsupervised Keyphrase Extraction with Multipartite Graphs,” pp. 1–6, 2018.

[66] L. Sterckx, T. Demeester, J. Deleu, and C. Develder, “Topical Word Importance for Fast Keyphrase Extraction,” no. 2, pp. 121–122, 2016.

[67] C. Thornley, “Advances in Information Retrieval,” *J. Doc.*, vol. 68, no. 5, pp. 806–810, 2014.

[68] I. H. Witten, G. W. Paynter, E. Frank, C. Gutwin, and C. G. Nevill-Manning, “KEA: Practical Automatic Keyphrase Extraction,” pp. 1–23, 1999.

[69] T. D. Nguyen and M.-T. Luong, “WINGNUS: Keyphrase Extraction Utilizing Document Logical Structure,” *Comput. Linguist.*, no. July, pp. 166–169, 2010.

[70] M. Sherriff and L. Williams, “Empirical Software Change Impact Analysis using Singular Value Decomposition,” *2008 Int. Conf. Softw. Testing, Verif. Valid.*, pp. 268–277, 2008.

[71] N. Rungta, S. Person, and J. Branchaud, “A change impact analysis to characterize evolving program behaviors,” *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 109–118, 2012.

[72] M. Singh, G. S. Walia, and A. Goswami, “Validation of Inspection Reviews over Variable Features Set Threshold,” in *2017 International Conference on Machine Learning and Data Science (MLDS)*, 2017, pp. 128–135.

[73] E. Guzman and W. Maalej, “How Do Users Like This Feature? A Fine Grained Sentiment

Analysis of App Reviews," in *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, 2014, pp. 153–162.

[74] X. Wang, L. Tang, H. Gao, and H. Liu, "Discovering overlapping groups in social media," *Proc. - IEEE Int. Conf. Data Mining, ICDM*, pp. 569–578, 2010.

[75] H. Liu, *Managing and Mining Graph Data*, vol. 40, no. June 2014. Boston, MA: Springer US, 2010.

[76] M. Singh, G. S. Walia, and A. Goswami, "An Empirical Investigation to Overcome Class-imbalance in Inspection Reviews," in *2017 International Conference on Machine Learning and Data Science (MLDS)*, 2017, pp. 128–135.

[77] M. Singh, V. Anu, G. S. Walia, and A. Goswami, "Validating Requirements Reviews by Introducing Fault-Type Level Granularity," in *Proceedings of the 11th Innovations in Software Engineering Conference on - ISEC '18*, 2018, pp. 1–11.

[78] M. E. Fagan, "Advances in Software Inspections," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 7, pp. 744–751, 1986.

[79] J. C. Carver, N. Nagappan, and A. Page, "The impact of educational background on the effectiveness of requirements inspections: An empirical study," *IEEE Trans. Softw. Eng.*, vol. 34, no. 6, pp. 800–812, 2008.

[80] M. A. Hall and G. Holmes, "Benchmarking attribute selection techniques for data mining," *Data Eng. IEEE Trans.*, vol. 15, no. 6, pp. 1437–1447, 2003.

[81] R. Zafarani, M. A. Abbasi, and H. Liu, "Social media mining: An introduction," *Soc. Media Min. An Introd.*, vol. 9781107018, pp. 1–320, 2014.

[82] S. S. im Walde, "Chapter 4 Clustering Algorithms and Evaluations," *Clust. Algorithms Eval.*, no. 1973, pp. 51–69, 2003.

[83] J. Hershberger, M. Maxel, and S. Suri, “Finding the k shortest simple paths: A new algorithm and its implementation,” *ACM Trans. Algorithms*, 2007.

[84] M. E. J. Newman, “Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality,” *Phys. Rev. E - Stat. Physics, Plasmas, Fluids, Relat. Interdiscip. Top.*, 2001.

[85] F. Kuipers, P. Van Mieghem, T. Korkmaz, and M. Krunz, “An overview of constraint-based path selection algorithms for QoS routing,” *IEEE Commun. Mag.*, 2002.