Software Issue Reports Classification using Association Mining

Mohd Syafiq Bin Zolkeply

A dissertation submitted for the partial fulfilment
for the degree of Doctor of Philosophy
in Computer Science

School of Computer Science and Informatics
Cardiff University
September 2023

Abstract

When a software system transitions into the maintenance phase, users often raise a significant number of issues pertaining to the system. These issues encompass a wide range of topics, including software bug reports, user experience sharing, and feature improvement requests. Therefore, it is necessary to classify them before they can be given to the appropriate developers for resolution. However, the process of manually categorising these issue reports is characterised by a significant amount of labour, a substantial investment of time, and a susceptibility to errors. Although there have been studies conducted on the automation of this process, they primarily depend on prevalent or recurring patterns found within the datasets. In instances where reports diverge from prevailing patterns, there is a higher likelihood of misclassification occurring. This thesis is driven by the motivation to present a novel approach for classifying software issue reports, drawing inspiration from the technique of classification using association mining. In contrast to the conventional approach of just mining dominant patterns from the data, the thesis revealed the efficacy of mining both dominant and weak patterns from the data. Furthermore, it demonstrated the potential of using these patterns collectively in order to categorise issue reports. The experimental results demonstrate that our novel approach, which was evaluated on benchmark datasets derived from four open source software systems, has comparable accuracy to the current state-of-the-art methods. Furthermore, our method possesses unique advantages over the existing approaches.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

Thank You, Almighty God, Allah SWT, for all the blessings that You have showered on me throughout my PhD journey and for making it a success.

Dr. Jianhua Shao, my advisor, whose advice, encouragement, and insight have been priceless to me, has my eternal gratitude. Thank you for being such a wonderful mentor to me; your patience and commitment to my work have been a great source of motivation.

I want to thank everyone in my family for their thoughts and prayers, as well as their unwavering love, support, and words of encouragement. Their faith in me has been a constant source of inspiration, keeping me going during tough times.

In addition, I'd like to thank my friends Drs. Helmi, Haziq, Sharih, Wan Mohd Asyraf, Noyu, Hizal, Bassam, Fikry, Fahad, and Wafi, who have been there for me when I've needed them the most with words of encouragement, laughter, and diversion.

I owe a great deal of gratitude to the School of Computer Science and Informatics at Cardiff University for providing an excellent learning environment and unending support throughout my PhD studies, as well as to Majlis Amanah Rakyat (MARA) for providing the financial resources that made my studies possible.

Finally, I want to thank everyone who has helped me get to this point in my education and who has contributed in any way to my work.

Chapter 1

Introduction

1.1 Introduction

Maintaining an operational software system is an essential phase in software life-cycle. As claimed by Lehman's well-known Law of Continuing Change, software in a real world environment must evolve and it is inevitable or it will become less useful and end up obsolete [40]. It was reported, that 67% of the cost in software life-cycle was consumed by software maintenance [69]. Thus, it is essential that we are able to maintain a software system effectively. Well-managed software maintenance activities will ensure a software system's longevity and to stay fit for its purpose [6].

Generally, the software maintenance and evolution phase begins when a software system has completed its development cycle and may be handed over to the client or may be released directly. [75]. During the operational period, the software will be used by the intended users. As the users become more experienced with the software, they may suggest extra functionality or features [71] to be added to the system, but they may also encounter circumstances where the software does not perform as expected according to its specifications. When this happens, we say that the software has bugs, defects or is faulty [4]. Some researchers have observed that deviation from specification can also be caused by the changing of operating environment which requires the software to be upgraded so that it will

continue to operate as expected. [6, 71, 20].

Software maintenance activities generally begin with user requests for example, users suggesting new feature to be incorporated into the system or bugs fixing or upgrades to be carried out. These requests are typically recorded as an issue report in an Issue Tracking System (ITS). After an issue report is received, it needs to be classified according to its type and category [44]. A series of activities are then carried out to address the issue raised by the user [20]. Figure 1.1 depicts an example of issue report classified as *bug* extracted from the HttpClient open source project.

Project	HttpClient
Bug ID	HTTPCLIENT-8
Type | Priority	Bug | Major
Title	`LogSource.setLevel` incorrectly uses entrySet

Description	When I call `LogSource.setLevel`, I get the following exception:
	
	`java.lang.ClassCastException: java.util.HashMap$Entry at org.apache.commons.httpclient.log.LogSource.setLevel` (LogSource.java:158)
	
	The Calling code is:
	
	`LogSource.setLevel (Log.OFF);`
	
	The error (I believe) that you should get the value set from the map, not the entry set (in LogSource):
	
	`static public void setLevel(int level) {`
	`  Iterator it = _logs.entrySet().iterator();` <-- should be `_logs.values()`
	`  while(it.hasNext()) {`
	`  Log log = (Log)(it.next());`
	`  log.setLevel(level);`
	`  }`
	`}`

Figure 1.1: Example of Issue Report

As can be seen from this example, issue reports are typically written in free text. This makes classifying issue reports difficult and error prone. For example, some may assume that having code snippet is a strong indicator that an issue report is bug, but this is not always the case.

1.2 Research Motivation

In this research, we study how issue reports may be classified automatically. The reporters may also carry out the classifications, depending on the ITS. During *classification*, they attempt to identify the type of issue to be dealt with and to have an initial assessment of which component in the source code is likely to be impacted when the issue is implemented [81]. The classification task may also involve identifying which developer/team are most appropriate to review and address it. This is not an easy task. Studies have shown that, there can be more than 1000 reports received for an Open Source Software (OSS) system [1] daily. In addition, issue reports are presented in a natural language which adds to the challenge to classify into their respective categories correctly [40, 20, 76]. Therefore, classifying issue reports is time-consuming, labour intensive and error-prone when it is performed manually.

It was reported that, it took 90 days to manually classify 7000 issue reports from an Open Source Software system [25]. As discovered by Herzig et al [25] in their study, 39% of the issue reports submitted have been misclassified and among issue reports that were marked as bug, more than 30% were non-bug report. Previous study by Cavalcanti et al [12] reported that more than *10 minutes* is spent on average to make a decision on its type when classifying an issue report on Open Source projects and, in private projects, this time can exceed *20 minutes*. Runeson et al [66] further reported that approximately *30 minutes* were spent on average to analyse a single issue report in Sony Ericsson. This is clearly too costly, especially modern software systems are getting larger and for this task is

[1] Open-source software (OSS) is computer software with its source code made available with a license in which the copyright holder provides the rights to study, change, and distribute the software to anyone and for any purpose. Open-source software may be developed in a collaborative public manner. According to scientists who studied it, open-source software is a prominent example of open collaboration. The term is often written without a hyphen as "open source software". Example of OSS, Apache Tomcat and Eclipse Plugin.

desirable.

In an ideal world, software project teams should be spending more time to collect information on which source files or component in the software that needs to be fixed or augmented to rectify the issue [34, 39] rather than spending time with the classification task. Furthermore, misclassification of issue report can happen and this can lead to late software release due to issue reports tossing (reassignment of report to different developers). Thus, classifying issue reports efficiently and effectively is a challenging task. This is because issue reports can contain mixture of natural language, code snippet, stack traces, or links [9, 53]. As a result, wrong decisions are made, resulting in delays or less issue reports being implemented [41].

1.3 Research Problem

To address the issues discussed above, some attempts have been made to classify issue reports automatically. For example, Antoniol et al. [4] experimented the use of standard machine learning techniques such as Naïve Bayes to classify issue reports extracted from several open source issue tracking systems into bug or non-bug reports. They answered a question whether or not the information contained in an issue report can be used to classify it into *bugs* or *non-bug* (*other types of request*). Zhou et al [86] and Q. Fan et al [19] used a combination of text and data mining techniques for classification, but they made use of additional information such as severity, priority and component that may be found in an issue report.

The same approach used by Natthakul et al [62], where they made use of *title*, *description* and *discussion* information found in an issue report and incorporate *topic modeling* technique to derive set of topics used as features when training the classifier. Patil et al. [60] proposed a concept-based classification by training a large corpus of wikipedia pages to find semantic similarity between terms found in

issue reports and wikipedia pages. However, their study requires a large amount of training data for the classifier to build a good model where training time would be significant.

While moderate success has been reported in these studies, there is a strong correspondence between the level of accuracy that all of the studies achieved. Table 1.1 shows the Average *F-Score* reported by three studies in this area. These studies are selected since they were using the same benchmark dataset provided by Herzig et al [25].

Table 1.1: Average *F-Score* of Previous Studies (*10-fold CV Setup*)

Studies / Projects	Http-Client	Lucene	Jackrabbit
S1-Natthakul et al [62] (2013)	0.729	0.797	0.738
S2-Nitish et al [59] (2016)	0.744	0.834	0.837
S3-Pannavat et al [77] (2017)	0.809	0.853	0.788

As can be observed from the table, we can see that the range of average *F-Score* achieved falls between 0.7 - 0.85 for all studies. This has led us to conjecture that some issue reports are naturally difficult to classify, for example due to the imbalance distribution of bug and non-bug reports within the dataset or due to ambiguous reports such those containing both bug and non-bug issues in one report.

Consider the imbalance problem, for example when an imbalance dataset is used to train a classifier, the learning tends to be biased towards the data that has more instances compared to the one with less instances. That is, a conventional learning algorithm typically looks for some "dominant" patterns from issue reports to build a classifier. To illustrate this, let us examine the three different issue reports presented below. In Figure 1.2 the first reporting a bug whilst the second and third requesting a new feature.

R1, Type: Bug
Error releasing chunked connections with no response body. http method base release connection does not successfully release the connection if closing the response stream throws an exception.

R2, Type: Feature Request
Allow declare *error* and declare warning to support type expressions.

R3, Type: Feature Request
Request to include *error* message when debugging is failed.

Figure 1.2: Example software issue reports

In Figure 1.2, `error` appears in all three reports. As such, `error` is not discriminative enough, thus is likely to be ignored and more dominant terms or combination of terms will be sought to build a classifier by the existing approaches.

We argue that non-dominant patterns are also important to building an effective classifier for issue reports. In some situation, rules generated by *dominant pattern* might not capture enough useful patterns to build an effective classifier. Thus, we raise a question, ***does dominant pattern provide enough coverage of rules when building a classifier?***

To illustrate and support our argument on why dominant patterns are not sufficient to build an effective classifier, we use the following example. Let $\mathbb{R}$ be a set of issue reports with 130 reports in total and the number of reports in $\mathbb{R}$

containing the term `error` is 100 occurrences. Of the 100 occurrences, 67 were extracted from the reports known as `bug`. We write this as a rule in the form of *error* $\rightarrow$ *bug*. While other 33 occurrences constitute a rule that represents the `non-bug` reports, *error* $\rightarrow$ *nonbug*. As mentioned earlier, existing approach usually sought for dominant pattern to build classifier. As a result, the rule that is more dominant, will be kept to use for classification of any unseen report later, in this case *error* $\rightarrow$ *bug* since it has more than 50% occurrences. The rule *error* $\rightarrow$ *nonbug* will be discarded as otherwise the classifier will no longer be consistent. Now let us assume that we the following two unseen reports $R1$ and $R2$ to be classified as *bug* or *non-bug* for the type and we will use *error* $\rightarrow$ *bug* as the rule for classification.

$R1 = \langle$ An error is received with unknown warning when executing `classException()` method, **`type?`** $\rangle$

$R2 = \langle$ Improve error handling message in API class, **`type?`** $\rangle$

From the reports, we can conclude that $R1$ is reporting a bug since the reporter is receiving an error when executing the `classException()` method. Whereas $R2$ is an improvement request to handle *error* message for the API class. In this example, the term *error* appears in both report but in a different context. If we use *error* $\rightarrow$ *bug* as classification rule, both reports will be classified as *bug*. As a result $R2$ will be misclassified. This is the situation where *dominant patterns* become less useful for building a classifier.

We argue that not just dominant rules, but some other rules, as long as they occur frequently enough in a dataset should be kept in order to build an effective classifier for software issue reports. We call such rules `credible`. Basically, `credible` rules are the result of considering `weak` rules that are omitted by conventional classification algorithm due to classifier consistency requirements.

1.4 Research Hypothesis and Contributions

We hypothesise that dominant patterns may not provide adequate coverage to build an effective classifier. Thus, instead of looking for dominant patterns, our approach looks for any *credible* patterns. That is, when a pattern occurs frequently enough, even if it is not dominant, we consider it to be useful.

For example, suppose that `error` appeared in 20 issue reports, 10 in bug reports and 10 in feature request reports. If 10 occurrences of a term are deemed to be significant (hence credible), then we will extract both patterns (in the form of rules *error* $\rightarrow$ *bug report* and *error* $\rightarrow$ *feature request*) and use them collectively to classify a report through a majority vote. Our method is inspired by the classification association rule mining methodology [45, 42].

We decided to build our solution on top of the Classification based on Association Rule Mining (CARM) methodology, as it allows for flexibility in the generation of rules by relaxing the thresholds for support and confidence. This relaxation permits the inclusion of a greater variety of patterns, including both dominant and less frequent (credible) rules. By not limiting the classifier solely to dominant patterns, this approach is especially beneficial in contexts where issue reports are highly varied, such as open-source software projects or complex enterprise environments where reports may cover a diverse set of functionalities and features.

In such settings, relaxing the rule thresholds can enable the classifier to capture nuanced and context-specific patterns that more rigid classification methods may otherwise overlook. This can lead to more adaptable and accurate classification outcomes, particularly in projects where certain categories or issues occur less frequently but are nonetheless critical to efficient software maintenance. Thus, leveraging CARM allows our approach to accommodate a broader spectrum of issue types, enhancing classification accuracy and reducing misclassification risks across diverse software projects. The main contributions of the thesis are as

follows:

- We propose a CARM based method to learn a classifier from issue report data. We have designed two voting methods: one majority based and one support augmented. With our method, we only need to extract *title* and *summary* from an issue report. This is in contrast to other studies on classifying issue reports where combination of other features extracted from issue reports like *priority*, *severity* etc are used to enhance accuracy. Thus, our classification method has a wider applicability and can still function when other feature such as *priority* and *severity* are not available from issue reports. Our experiment revealed promising results in classification when compared to existing techniques.

- Our method relies on the extraction of keywords from issue reports. In this research we have considered and combined various feature selection techniques to achieve improved classification performance when comparing to the state of the art methods, as demonstrated through our experiments.

- We empirically evaluate our proposed approach using three different open source software systems through experiments with different settings to offer insight into our proposed methods.

- We engineered an 'unknown' classification category to capture issue reports that do not strongly align with either the bug or non-bug categories. This intentional design choice illustrates a limitation of binary classification, which often forces all reports into predefined classes, even when they do not fit well. By introducing an 'unknown' classification, we address this issue by allowing reports that lack clear classification indicators to remain unclassified, reducing the likelihood of misclassification and highlighting the need for a more flexible, multi-class approach in complex software issue reporting.

1.5 Thesis Organisation

Chapter 2 reviews the existing work relevant to our problem of software issue reports classification. We briefly discuss the existing software issue report classification methods, and review different techniques used to address software issue report classification.

Chapter 3 gives the details of our proposed approach. We define relevant concepts and introduce some necessary notions. Then, we provide an overview of two main steps in our method namely `Rules Generation` and `Reports Classification`.

In Chapter 4 we introduced an *Augmented based Voting* algorithm for classification as an enhancement of our proposed approach presented in Chapter 3.

In Chapter 5 we describe the characteristics of chosen datasets used in the experiments and the measurement employed in evaluation. We present the necessary Natural Language Processing steps taken to prepare the data before building our classifier with the data. The steps are presented in our `Text Normalization` procedure. Then, we empirically evaluate and compare the methods that we have proposed with the state-of-the-art. This chapter demonstrates the implementation of our proposed approach described in Chapter 3 and the enhancement version of our baseline presented in Chapter 4.

Chapter 6 summarises, concludes and provide some pointers for future direction of the research.

Chapter 2

Background

This section examines software maintenance, which serves as a background for the studies. Essentially, we focus on introducing general concepts and ideas about software maintenance. We will focus on a specific issue that we wish to investigate as we move along the broad topic of software maintenance. In this discussion, we have established our research context, and our research will go in that direction from here on out. Additionally, we consider current issues and difficulties as well as software maintenance issues that stand in the way of software lifecycle success.

2.1 Overview of Software Maintenance

There are several stages in the software engineering life cycle. Generally, requirements engineering is the first step. It is the initial phase, which entails activities aimed at eliciting stakeholders' requirements for development [63]. Once an agreement has been reached, the process of designing, implementing, testing, and deploying the solution begins. These phases are collectively referred to as the Software Development Life Cycle (SDLC). It may take months to develop software, but as long as the software is capable of serving the intended users, it will be put into operation [75].

In general, as defined by IEEE Standards for Software Maintenance [20], software maintenance means

"Modification of a software product after delivery to correct faults, to improve performance or other attributes or to adapt the product to a modified environment"

Once the software had been handed over to stakeholder or its real users, the software will be used as it was intended. During its operational lifetime, the software will be undergoing series of software maintenance activities [6].

In contrast to other phases in software process life-cycle, software maintenance incurs the largest amount of cost, time and effort to realize. This is due to the fact that, during software maintenance the activity implementing bug fixing, executing new feature request and adapting the software to new environment are typically to carry out [6, 71, 20].

Software maintenance involves a couple of stages. It begins when users or stakeholders fill in a modification request form (issue report here after), requesting for changes or modification to the software that is currently being used. The changes required are triggered from new features request, bug fixing, change of organization or business policy and migration to modern technologies in order to acquire better performance of software [71, 13].

The classification of reported issues is typically undertaken by specific roles within a software development team. In smaller teams, this task often falls to project managers; team leads, or dedicated triage engineers with significant domain knowledge. They manually review incoming issue reports, analyzing their textual descriptions, code snippets, and additional metadata to assign appropriate categories, such as bugs, feature requests, or improvements. In larger or open-source projects, community contributors or junior developers may assist in pre-classification, but the final decision often lies with core team members who validate and refine the classification to maintain consistency. This manual approach is time-consuming and error-prone, particularly as the volume and ambiguity of reports increase.

Once the reported issue is accepted, the software project team will perform a classification task to determine to which software maintenance activity the issue belongs. This is a crucial task to perform as wrong classification will affect the cost, time and manpower for the implementation of a reported issue [6].

2.2 Issue Report and Its Life-cycle

When an issue is reported to the Issue Tracking System (ITS). It will go through multiple phases to complete a life-cycle until the developer who is assigned to it declares the report resolved. Figure 2.1 illustrates the life-cycle of an issue in different phases.

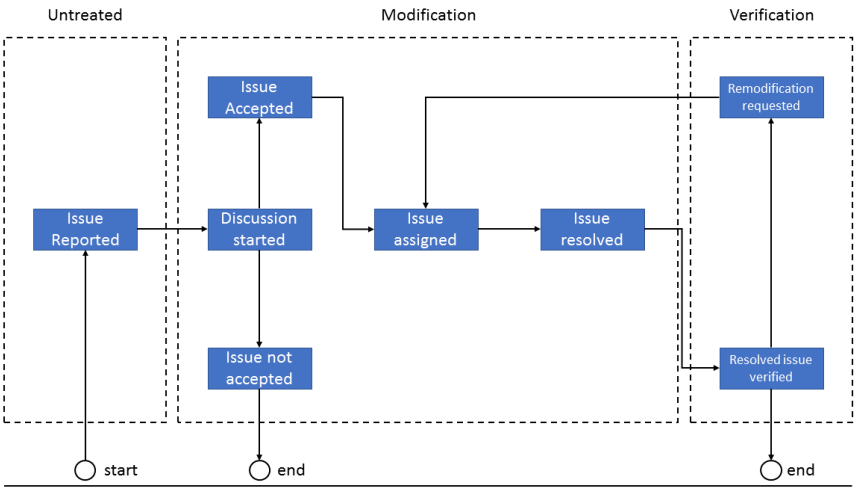

Figure 2.1: Issue Report Management

First, in the ***Untreated Phase***, a new Issue is created in the Issue repository. Each Issue contains different information fields, which are essential for the issue to be properly understood and implemented. For instance, it contains a short and long description of the requested change, the type of the change (e.g. defect or enhancement), the target component impacted by the change, and the software version. Depending on the Issue repository being used, it is possible to define more informational fields when configuring the repository. Some studies [44, 85] suggested that ***Untreated Phase*** is also known as the ***Classification Phase*** where the type of issue reported is identified (e.g *bug* or *non-bug*).

In the ***Modification Phase***, the Issue can either be accepted for implementation or rejected, and discussion concerning the Issue aspects takes place. Such a

discussion is carried out through comments that can be inserted directly into the Issue. Note that it is neither necessary to have a discussion before accepting an Issue nor for accepting it. There are many reasons for not accepting an Issue such as: poor descriptions, redundancy, that is, whenever the reported issue refers to an existing Issue, or it is not planned to fix the reported Issue, etc. Although discussion on every Issue proposed is not a requirement, it is often useful to have discussion on whether an Issue should be accepted or not. However, if an Issue is accepted, then it must be assigned to a developer who will perform the modifications per request, as shown in Figure 2.1. Assigning an Issue starts by finding a suitable developer to deal with the issue. This step is very important because finding the appropriate developer is crucial for obtaining the lowest, economically feasible fix time [48]. This developer should be one who has sufficient knowledge in the relevant aspects reported in the Issue [3]. The assignment also needs to comply with the developer's workload and the priority of the Issue [28]. Thus, due to these characteristics, assigning Issue is a labour intensive and time-consuming task mainly because it is performed manually [5, 29]. In fact, in many projects the number of Issues that are reported and need to be assigned can vary from dozens to hundreds per day [12]. As a result, if assignments are not properly performed, the project schedule can be affected the cost.

After the developer has implemented the Issue, it proceeds to the ***Verification Phase***, as shown in Figure 2.1. This phase encompasses of verification that the Issue has been properly implemented, which is commonly done by the quality assurance team. If the implementation carried out for the Issue needs an additional repair, it returns to the developer who performed the implementation earlier. It is worth mentioning that if a developer was not able to fix an Issue, it can be reassigned to others until an appropriate developer is found. When the Issue has been fixed properly, this work-flow reaches its end, and the change can be released for production.

The aforementioned procedures will continue running iteratively basis since software maintenance is a process of continuous change of the needs in a particular direction and it happens during the entire software life-cycle [51].

2.2.1 Software Maintenance in Open Source Software Perspective

This section further discusses the issues of performing maintenance activity in open source software. An open source software is a software that is freely available for use by any users across the globe. Due to its distributed nature, with a large number of users around the world, it is challenging to perform software maintenance since it receives all sorts of issue reports from its users.

The classification task encounters significant challenges due to the variety of issue report content. For instance, reports may include ambiguous descriptions (e.g., 'the system crashed unexpectedly' without specifics), inconsistent terminology (e.g., 'bug', 'issue', 'defect' used interchangeably), or a mix of natural language and code snippets that complicates automated processing. Additionally, incomplete reports, such as missing steps to reproduce the issue, further hinder accurate categorization.

Typically, users of open source software submit issue reports via a Bug Tracking System (BTS). The maintainer can make BTS available to accept any type of report and it will perform a classification process to categorise each report in terms of maintenance activities. Due to the fact that the BTS is open to all users, users can request anything by logging their issue reports. Typically, a user must categorise the type of issue report they wish to log. There are numerous subcategories for an issue report. In their study, Herzig et al. [25] identified 13 categories that are frequently mentioned by users when submitting an issue report to BTS. Table 2.1 outlines the categories defined by Herzig et al. in their

study.

Category	Description
Bug	Issue reports documenting a problem which impairs or prevents the correct functioning of a software and causes deviation from expected results. A bug can be an error, defect, failure or fault.
RFE	Issue reports which document request for enhancement (RFE) such as the addition of new functionality or a new feature.
Improvement	Issue reports specifying perfective maintenance task to improve the overall performance of software, e.g., to change a piece of code to produce results from a database faster.
Documentation	Issue reports which refer to updating external links or documentation related to code, e.g., to update API documentation in HTML format.
Task	Issue reports which specify a task that needs to be done.
Build System	Issue reports related to problems in build systems, which are used to automate several software build activities such as compiling the code and running test cases.
Refactoring	Issue reports specifying refactoring of source code, i.e., changing the non-functional attributes of a piece of code which improves its maintainability.
Design Defect	Issue reports about problems in the design of software, e.g., code smells.
Test	Issue reports which are related to test cases.
Cleanup	Issue reports about code cleanup, e.g., to clean unwanted code such as redundant code, dead code, etc
Backport	Issue reports related to backporting where a fix or patch of any flaw on the current version is applied to an older version of a system.
Specification	Issue reports related to changes in the requirement specification documents.
Others	Issue reports that cannot be classified to any of the above categories.

Table 2.1: Issues Reports Categories [25]

Classification at a fine resolution enables the software maintenance team to gain a better grasp of the issues being reported by the users. Despite its capacity to

facilitate comprehension, fine-grain classification may result in misclassification. Cavalcanti et al. noted in their study that issue reports are not always about software maintenance operations and that some are frequently about requests for assistance, architectural discussions, and legal and licencing difficulties (a common case for open source projects). Due to the range of issue report categories, misclassifications are inevitable. For instance, a novice developer may misclassify an issue report as a new feature. This type of misclassification can result in project delays, as misclassified issue reports may be assigned to the incorrect developers or receive less attention than they should.

2.2.2 Issues and Challenges in Classifying Issue Reports

When dealing with a large volume of issue reports we must consider the problem of misclassification. Herzig et al. [25] did a study in which they manually labelled over 7000 issue reports and discovered that 39% were misclassified. Additionally, they reported that a full analysis of a single issue report took an average of four minutes. This instance demonstrates how difficult it is to deal with natural language. This could result in ambiguity, increased processing time, and possibly uncertainty. These factors can lead to misclassification.

A few other studies related to misclassification were conducted to show the consequences of misclassification. Falessi et al. [18] in their study focused on the impact of misclassification of bug reports toward verification and validation (V&V) in a software project. They argue that misclassifying bug report will affect the effectiveness of V&V activity. Zeller et al. [83, 26] formed a standard operating procedure on how to make use of bug reports extracted from an issue management system for code quality analysis. This effort was to show that there should be a linkage between bugs reported in the system with source code and if misclassification incurred it is hard to trace which bugs report should be related to which source code.

As a summary for this sub-section, we believe misclassification is the root problem that will lead to varieties of adverse side effects. From project delay to increment of cost. Since maintenance work contributes 67% of total cost in software process life cycle, we, need to address this misclassification problem as early as possible during software maintenance life cycle to avoid any difficulties in future.

2.3 Analysis of Existing Approaches

In this section, we present works that are related to our research context. We divide our explanation of related works into three subsections. The first section discusses works related to the study of issue or bugs report in supporting software maintenance activity. The second subsection discusses works or studies related to automated classification involving text data in software engineering field. The last subsection summarizes the related works and how our research differs from those practices that have been conducted by previous researcher.

The need for improving automated classification arises due to the inefficiencies and limitations of manual approaches. Manual classification is highly labor-intensive, requiring significant time and expertise from developers or project managers. Additionally, inconsistencies in report categorization occur due to human error, varying levels of domain knowledge, and subjective interpretations of report content. Automating this process not only reduces human effort but also enhances scalability and consistency in classification, particularly as the volume of issue reports grows in large-scale software projects.

2.3.1 Works Related to Issue Reports Classification

Constructing an automated solution for issue report classification adopting techniques from natural language processing and machine learning has been receiving

an increased attention from software maintenance research community recently. It is worth mentioning that classifying issue reports is an essential sub-task of the overall bug triage process. The primary goal of bug triage is to classify unsolved issue reports into their categories and assigning them to appropriate developers for further action. There is a large body of work on bug triage using machine learning and text mining techniques over the past decade [87, 56]

To the best of our knowledge, an automated solution for issue report classification was first introduced by Antoniol et al. [4] in 2008 using textual corpus from the open source BTS (Moziila, Eclipse and Jboss). They conducted a study to explore the possibility to classify corrective maintenance request known as bug fixing from large corpus extracted from Bug Tracking Systems (BTS). In the study, they aimed to answer the following research questions:

- To what extent the information contained in issues posted on bug tracking systems can be used to classify such issues, distinguishing bugs (i.e., corrective maintenance) from other activities (i.e, enhancement, refactoring)?
- What are the terms/fields that machine learning techniques use to discern bugs from other issues?
- Do machine learning techniques perform better than grep and regular expression matching, in general, techniques often used to analyse CVS/SVN logs and classify commits between bugs and other activities?

Although the result shows data in BTS are relevant to identify bugs, the main drawback of this study is that, it is only capable of distinguishing between bugs and non-bugs, not other types of issues encountered in software evolution and maintenance. A more detailed classification will be useful for a software developer.

The work by Antoniol et al was further enhanced by Hindle et al [27] in 2009. Their approach was slightly different by considering a few more classification

classes. Instead of solely focusing on bug and non-bug categorization they extend the Swansons Taxonomy [76] by introducing few more categories related to their study purposes. They produced classification classes such as Feature Addition, Maintenance, Module Management, Legal, Non-functional source code changes, Source Control System (SCS) Management and Meta-Program. Their study mainly focuses on large commit dataset. The data that is usually generated when programmer makes changes on the particular source code. There is a few log files that will be generated once a programmer has performed such commit changes. The proposed approach used a supervised machine learning technique and validated with seven different projects. Although automation technique is achieved in this case, this study is exposed to some validity threat. The extension of categorization that has been performed is not usable by other study since it is tailored to work with large commit dataset. In fact, the validity of the classification is questionable.

In 2013 Herzig et al. [25] conducted a novel study by performing manual labelling of more than 7000 issue reports extracted from five different open source projects. The reason for investing such effort is because they wanted to show that the issue report contained in any open source Bug Tracking System (BTS) cannot be directly used for mining to find insight. This is because the data contains noise that leads to inaccuracy of any automated effort of classification. From their finding, it was reported 39% of the studied data were misclassified and 16% to 42% of the misclassified data do not belong to its category after reclassification. This study had an impact on the previous studies that simply used the data contained in BTS for automated classification.

Motivated by this effort, in 2014 Pavneet et al. [35] proposed an automated technique that was capable of performing reclassification of an issue report into an appropriate category. The approach worked by extracting various feature values from a bug report and predicts if a bug report needs to be reclassified and its re-

classified category. The approach was evaluated by reclassifying more than 7,000 bug reports from HTTP Client, Jackrabbit, Lucene-Java, Rhino, and Tomcat 5 into 1 out of 13 categories. The results of the experiments show achievement of weighted precision, recall, and F1 (F-measure) scores in the ranges of 0.58-0.71, 0.61-0.72, and 0.57-0.71 respectively. Regarding F1, which is the harmonic mean of precision and recall, the approach can substantially outperform several other techniques by 28.88%-416.66%.

The same researchers conducted another study to investigate the potential biases in bug localization [36]. This study examines to what extent these potential biases affect the results of a bug localization technique and whether bug localization researchers need to consider these potential biases when evaluating their solutions. They analyzed issue reports from three different projects: HTTP-Client, Jackrabbit, and Lucene-Java to examine the impact of high three biases on bug localization. The results show that one of these biases significantly and substantially affects bug localization results, while the other two biases have a negligible or minor impact.

Other related studies conducted by Rodríguez-Pérez et al. [65] In their study they developed a tool to solve the problem by providing all relevant information required for decision making to determine an issue corresponds to a bug report or not. The tool works by automatically extracting information from project's repositories. It offers a web-based interface which allows collaboration, traceability, and transparency in the identification of bug reports. These available features make the identification easier, faster, and more reliable.

Nam et al. [57] proposed a novel unsupervised learning approach to label the unlabeled data to perform prediction. CLA and CLAMI, that illustrates the potential capability to predict defective files on unlabeled datasets in an automated manner without requiring the manual effort. The fundamental idea of both approaches is to label an unlabeled dataset by using the magnitude of metric values.

The approach was validated on seven open-source projects, and the outcome of CLAMI approach led to the promising prediction performances, 0.636 and 0.723 in average f-measure and AUC, that are comparable to those of defect prediction based on supervised learning.

Chawla et al. [14] conducted a study to automatically label an issue to be identified as bug or other request using the fuzzy set theory. Three open source software systems namely HTTPClient, Jackrabbit and Lucene were selected for experiments purposes. The outcome of experiments managed to achieve an accuracy of 87%, 83.5% and 90.8% and F- Measure score of 0.83, 0.79 and 0.84 respectively. This is a considerable improvement as compared to the earlier reported work on these three datasets-using topic modeling approach to tackle the similar problem domain.

Shi et al. [72] proposed a prediction approach to predict which requirements are likely to evolve by learning from its evolution history. The defined a series of metrics to characterize historic evolution information to be used as attributes for prediction. They validated the approach through a case study and managed to obtain promising results. The results indicate that the defined metrics are sensitive to the history of requirements evolution, and the prediction method can reach a valuable outcome for requirement engineers to balance their workload and risks.

Zhang et al. [85] in their survey study highlighted an exhaustive survey on the existing work on bug-report analysis. They also claimed that bug reports are significance software artifacts that describe software bugs, especially in open-source software. Due to the increasing number of bug reports, more researches are insterested to venture into research that used bug report for analysis. Some of them conduct an automated inspection to discover duplication of bug reports and localizing bugs based on bug reports. This scenario shows that study on bug reports are worth pursuing and will bring an impactful contribution towards

body of knowledge.

In summary, while several studies have explored manual and rule-based approaches for classifying issue reports, these methods are often limited by their reliance on human intervention and predefined rules. Such approaches lack the flexibility to adapt to the variability and complexity of real-world datasets.

2.3.2 Works Related to Automated Classification

The automated classification of non-functional requirements (*NFRs*) has been an area of growing interest due to the labour-intensive nature of manually classifying requirements expressed in natural language. Early foundational work by Cleland-Huang et al. [15, 16] explored the use of information retrieval techniques to detect and classify NFRs. Their approach focused on extracting features from natural language requirements to reduce the manual effort required for classification, especially when handling large volumes of requirements. They argued that early detection of NFRs plays a critical role in enabling system-level constraints to be integrated into the architectural design phase, preventing costly refactoring later in the development process. The study demonstrated the potential of their automated approach across a diverse set of structured and unstructured documents, such as requirements specifications, meeting minutes, and stakeholder notes, which often contain scattered and non-categorised NFRs. Their evaluation, conducted using requirements specifications developed by MS students at DePaul University, showcased the feasibility of automating this task, although challenges such as accuracy and scalability persisted.

Building on this foundational work, subsequent research has sought to enhance the efficiency and accuracy of automated classification methods. One notable contribution is the semi-supervised learning approach introduced by Casamayor et al. [11], which aimed to address some of the limitations inherent in fully su-

pervised methods. Using the same data set used in the study by Cleland-Huang et al., Casamayor et al. proposed a classification method that utilised a small set of labelled requirements to train an initial classifier while also incorporating the knowledge provided by the unlabelled requirements. This semi-supervised technique exploited textual properties and iterative feedback from analysts to progressively improve classification accuracy. Their results showed that the semi-supervised approach achieved accuracy rates exceeding 70%, outperforming supervised methods in terms of both accuracy and the amount of manual labelling required.

While the semi-supervised approach introduced by Casamayor et al. demonstrated a significant reduction in human effort compared to fully supervised methods, their study also highlighted the inevitability of human intervention in the evaluation process. Analysts were still required to validate the accuracy of the classification results after each iteration of the learning process, emphasising the ongoing need for a balance between automation and manual oversight in such tasks. Despite these limitations, the integration of feedback mechanisms into the classification process marked a notable step forward, demonstrating the potential for iterative, user-driven learning to enhance automated classification performance.

Chawla et al. [14] developed a new automatic severity classification model to classify the bugs into two classes i.e. bug and other request. Authors adopt term frequency (TF-IDF) and latent semantic indexing (LSI) algorithms for selecting relevant features. Further, fuzzy logic based classifier is used for classification task. Three well known datasets are considered to validate the performance of fuzzy classifier. The simulation results are assessed using accuracy performance metric. The simulation results of fuzzy classifier are compared with LR, NB and ADTree classifiers. The experimental results indicated that fuzzy classifier obtains higher accuracy rate than LR, NB and ADTree classifiers and average

accuracy ranges in between 82-84% for all three datasets.

Anas et al. (2016) [49] proposed a novel unsupervised and computationally efficient approach for detecting, classifying, and tracking NFRs. The research analysis is predicated on the premise that NFRs are frequently implicitly enforced by FRs. This knowledge is critical for capturing, modelling, and identifying specific NFRs. To summarise, the semantic similarity technique is used to identify NFRs within FRs. The research is motivated by the fact that current NFR detection and classification methods require supervision because the model must be well-trained using classified manual data in order to classify previously unseen instances. Such data are not always accessible. Indeed, experts from diverse fields employ a variety of terminologies. A classifier that has been trained on a particular application may not necessarily work in another area. The research's strength is in its unsupervised approach to conducting automated classification, detection, and tracing of NFRs using FRs documents. The methods used to complete the aforementioned task are well-explained, and the outcome demonstrates a high precision percentage. The following limitations apply to this study. External validity is the first. Because the dataset consisted of only three midsize software systems with limited functional requirements, the results generated were not generalizable beyond the specific experimental settings. These midsize systems may exhibit characteristics that differ from those of large scale systems from a different domain. Additionally, all selected systems are object-oriented (OO) Java-based. Thus, systems written in structured or other programming languages may not be compatible with the approach. Apart from that, the algorithm for selecting clusters and similarity measures are not holistic. Other techniques are not considered in order to ascertain their applicability in this context. The number of NFRs categories is extremely limited, as ISO defines more fine-grained NFRs categories. Internal validity is also debatable due to the human judgement used to create the classification and traceability answer set. Humans have a tendency to create bias, and dealing with personnel from various projects may also contribute

to misunderstanding. Additionally, the results indicate that this approach has a low precision percentage of 53 percent on average.

Several additional studies used a similar technique [64, 74, 33, 32] to detect and classify textual documents into appropriate classes or categories.

To address the limitations of manual and rule-based methods, machine learning-based approaches have been proposed. However, these studies often focus on highly supported patterns, overlooking less frequent but contextually significant features, and struggle with handling unstructured content, such as code snippets and ambiguous terminology.

2.3.3 Summary of Related Works

We identified a range of related works closely aligned with our study, which focuses on solving the classification problem of issue reports in Open-Source Software (OSS). Our selection is based on the availability of datasets widely used by other researchers in this area. To promote empirical research value, we utilize the same datasets used by [25, 36, 38] and demonstrate their reproducibility and applicability within our research context.

Our findings are comparable to those of Zhou et al. [86], Pandey et al. [59], and Antoniol et al. [4], but our approach is distinct in several key ways. Pandey et al. [59] emphasised the use of a single supervised machine learning approach, which can improve precision by leveraging large amounts of labelled data. However, this method relies on a labour-intensive manual labelling process. Zhou et al. [86] and Antoniol et al. [4] focused on binary classification, determining whether an issue report represents a bug or not. While these studies provided valuable contributions, they predominantly relied on dominant patterns, such as highly frequent terms, often neglecting less frequent but contextually significant features. This limitation reduces their ability to handle ambiguous cases and edge scenarios

effectively.

Our work addresses these gaps by introducing the concept of credible patterns, which incorporates less frequent but meaningful features into the classification process. By integrating association rule mining with a support-based mechanism, our approach enhances the robustness and adaptability of binary classification. Unlike prior works that solely determine whether an issue report is a bug, our method improves the granularity of classification by leveraging features extracted from both textual and non-textual descriptions. This distinction offers a novel framework for refining issue report categorization, overcoming key limitations of prior studies.

As discussed in Sections 2.3.1 and 2.3.2, existing approaches face several challenges, including reliance on human intervention, difficulty in handling unstructured data, and neglect of less frequent yet credible patterns. This thesis proposes a hybrid approach that combines both dominant and credible patterns to improve classification accuracy while addressing these limitations.

The research in this thesis is motivated by the challenges identified in previous studies and aims to address key gaps in issue report classification. The following research question guides the study:

Primary Research Question

How can the inclusion of credible patterns (less frequent but meaningful features) may improve the accuracy and robustness of automated issue report classification?

The following section will detail our strategy for resolving the issue.

Chapter 3

Majority Vote Classification using Association Rule Mining

A classification rule mining method is evaluated based on several factors. The objective is to derive from the data a set of rules that, when applied to the prediction of newly acquired datasets, may achieve a high degree of accuracy in classification.

Hence, this section presents our method in detail, which is inspired by classification based on association rule mining. We first introduce some necessary concepts and then describe the two key steps of our method: rule generation and report classification.

3.1 Association Rules Mining

Association Rule Mining (ARM) is a crucial and widely implemented technique in the fields of data mining and machine learning, providing profound insights into relationships within large datasets. The primary focus is to explore how different items relate and interact, primarily used in market basket analysis to understand the relationships between different purchased products. The two fundamental metrics, "support" and "confidence," serve as pillars in assessing the strength and reliability of the inferred association rules. This section offers an in-depth

exploration of these concepts, their applications, implications, and significance in various domains.

ARM is a method for discovering intriguing and significant patterns, correlations, and relationships between elements in a collection. It is a critical component of knowledge discovery in databases (KDD), assisting academics and professionals from a wide range of fields in extracting important insights and comprehending intrinsic linkages inside complicated, enormous information.

ARM is versatile, with its applications extending beyond the realms of market basket analysis. It plays a pivotal role in healthcare for predicting disease patterns and occurrences, in finance for analyzing customer spending behavior, and in e-commerce for optimizing product recommendations, thus spanning a range of industries and contributing significantly to advancements in these domains.

3.1.1 Apriori Algorithm

The Apriori algorithm is a seminal approach for mining frequent itemsets and generating association rules. It utilizes the principle that all non-empty subsets of a frequent itemset must also be frequent. The algorithm operates in two main steps:

1. Identification of the frequent itemsets.
2. Generation of association rules from the identified frequent itemsets.

3.1.2 Antecedent and Consequent

In an association rule $X \Rightarrow Y$, X is the Antecedent, and Y is the Consequent. The antecedent is the condition of the rule, while the consequent is the result. Understanding these components is crucial for interpreting the implications of the derived association rules accurately.

3.1.3 Support

The prevalence or frequency of an itemset within the dataset is represented by support. It is a metric that shows how frequently things appear together in the dataset, showing the general prevalence and influence of certain item combinations.

The mathematical representation of support is expressed as follows:

$$\text{Support}(X) = \frac{\text{Number of Transactions containing } X}{\text{Total Number of Transactions}}$$

This implies that support, by providing a relative frequency of the occurrence of the itemset in the dataset, offers a quantitative insight into the prevalence of specific item combinations in the overall dataset.

Consider a practical scenario where a dataset consists of 100 transactions in a retail store, and in 15 of those transactions, customers purchased both milk and bread together. In this scenario, the support for the itemset {milk, bread} would be calculated as follows:

$$\text{Support}(\{milk, bread\}) = \frac{15}{100} = 0.15$$

This implies that the combination of milk and bread occurs in 15% of all transactions, showcasing a moderate frequency of this itemset, serving as a basis for further analysis and rule generation.

A high support value indicates that the itemset is common, and the rules generated from such itemsets are more likely to be significant. It is crucial for filtering out infrequent itemsets and focusing on the ones that have a meaningful presence in the dataset, providing a foundational basis for generating reliable and robust association rules.

While support is indispensable for identifying prevalent itemsets, it is crucial to carefully choose the support threshold. A very high threshold may result in

missing out on potentially meaningful, albeit less frequent, itemsets. Conversely, a very low threshold may yield an overwhelming number of itemsets, including many that are not meaningful, complicating the analysis.

3.1.4 Confidence

Confidence is another pivotal metric, representing the conditional probability that a transaction containing item X also contains item Y. It serves as an indicator of the reliability and strength of the inferred association rules and helps in assessing how frequently the rules are proven true.

The mathematical representation of confidence is expressed as follows:

$$\text{Confidence}(X \Rightarrow Y) = \frac{\text{Support}(X \cap Y)}{\text{Support}(X)}$$

It provides a quantitative measure, a probability, depicting how likely item Y is purchased when item X is purchased, enabling analysts to draw inferences about purchasing behaviors and potential product associations.

Extending the previous example, if in 8 out of the 10 transactions where customers bought milk, they also bought bread, the confidence for the rule $\{\text{milk}\} \Rightarrow \{\text{bread}\}$ would be calculated as follows:

$$\text{Confidence}(\{milk\} \Rightarrow \{bread\}) = \frac{\text{Support}(\{milk, bread\})}{\text{Support}(\{milk\})}$$

$$\text{Confidence}(\{milk\} \Rightarrow \{bread\}) = \frac{0.1}{\text{Support}(\{milk\})}$$

Assuming the support for milk is 0.2:

$$\text{Confidence} = \frac{0.1}{0.2} = 0.5$$

So, the confidence that a customer will buy bread given they have bought milk is 0.5.

Confidence is paramount for evaluating the robustness and reliability of the association rules generated. It allows analysts to gauge the strength of the implications made by the association rules and filter out rules that do not meet the reliability threshold, focusing on the ones that are more likely to hold true.

While high confidence values are indicative of strong associations, relying solely on confidence can lead to misleading conclusions as it does not take into account the base prevalence of the itemset. Consequently, it is pivotal to consider both support and confidence to derive meaningful and reliable insights and to avoid the pitfalls of relying on a singular metric.

3.1.5 Significance and Utility

Support aids in identifying substantial itemsets, eliminating infrequent and potentially irrelevant associations, and providing a ground truth basis for further analysis. In contrast, confidence assists in delineating rules with sufficient reliability, acting as a filter to discern the strength of the implications made by the rules discovered through the mining process.

By configuring thresholds for support and confidence, one can streamline the generated rules, focusing on those with higher reliability and relevance, thus enabling data-driven, precise, and impactful decision-making.

Balancing support and confidence is paramount. High support and high confidence imply a strong, frequent rule, but having too stringent a threshold might result in missed opportunities to uncover less obvious, yet valuable insights. Conversely, low thresholds may yield too many rules, complicating the analysis and leading to potentially spurious and irrelevant conclusions.

3.1.6 Summary of Association Rules Mining

Association Rule Mining, underlined by metrics such as support and confidence, is indispensable in unearthing valuable insights and correlations in large datasets. The interplay between support and confidence enables the extraction of meaningful patterns, which can be pivotal for various industries to optimize their strategies and operations.

While support offers insights into the prevalence of itemsets, enabling the identification of prevalent patterns and trends, confidence provides a reliability measure for the generated rules, allowing for the assessment of the implications and predictions made by the rules.

3.2 Proposed Approach

Without loss of generality we assume that an issue report R is represented as a vector of terms $R = \langle t_1, t_2, \ldots, t_n, C \rangle$, where each $t_i, 1 \leq i \leq n$ is a distinct term extracted from the issue report text and C is its category. For example, the report given in Example 1.2 is represented as

$$R = \langle error, warning, type, exception, feature \rangle$$

Here, R is represented, rather arbitrarily, by a vector that contains the nouns appeared in the issue report and a category term `feature`. We note that different vectorization of an issue report is possible. In this section, however, we concentrate on how a classifier may be derived from vectorised reports, rather than how a report may be best vectorised.

From a set of vectorised reports $\mathbb{R} = \{V_1, V_2, \ldots, V_m\}$, we wish to find rules of the form

$$r : a_1, a_2, \ldots, a_k \rightarrow c$$

where $a_1, a_2, \ldots, a_k$ are an *association* of terms and c is a report category. For example, $error, warning \rightarrow feature$ is one such rule and suggests that "if `error` and `warning` are present in an issue report, then its category will be `feature request`". Our goal is to derive a set of such rules that can be used to classify issue reports accurately.

Given a set of vectorised reports, it is easy to see many rules can be derived from it and some of the rules may not be sound enough. To ensure that the rules that we derive from a corpus of issue reports have some "minimal credibility", we employ two commonly used measures [2]:

- **support.** This is a count of how many times the association of terms $(a_1, a_2, \ldots, a_k)$ of a rule $(r : a_1, a_2, \ldots, a_k \rightarrow c)$ has occurred in a set of reports $(\mathbb{R})$. Support indicates the *strength* of a rule.
- **confidence.** This is the ratio of the number of times a rule $(r : a_1, a_2, \ldots, a_k \rightarrow c)$ has occurred to the number of times the association of terms of the rule $(a_1, a_2, \ldots, a_k)$ has occurred by itself in a set of reports $(\mathbb{R})$. Confidence indicates the *accuracy* of a rule.

To illustrate these two measures, consider the dataset given in Table 3.1, where each row represents a vectorised issue report. For simplicity of presentation, we consider two categories only: `bug` and `non-bug` here.

Suppose that we have a rule *support*, *allow* $\rightarrow$ *non-bug*. The support for this rule is 2, as the number of times `support` and `allow` occur together in Table 3.1 is 2. The confidence of this rule is $\frac{2}{2}$ =100% as in both occurrences of the association, the category is `non-bug`. On the other hand, if we have rule *support* $\rightarrow$ *non-bug*, then its support is 4 and confidence 80%.

We assume that two parameters, minimum support (*minSupp*) and minimum confidence (*minConf*), are specified, typically by the users of our method and tuned

Table 3.1: Issue Reports as Vectors of Terms

ID	Terms	Category
1	error, bug, break, compile, support	bug
2	except	bug
3	error, warn, allow, support, declare	non-bug
4	error, support, good, declare	non-bug
5	null, except	bug
6	except, compile	bug
7	except, problem, compile	bug
8	warn, support, nice	non-bug
9	add	non-bug
10	allow, support	non-bug

for particular datasets, and we search for all the rules from a set of vectorised reports that have the minimum support and confidence. We call these rules *credible* rules.

3.2.1 Minimal Credibility

One obvious class of rules of interest are those with support and confidence levels that exceed a given minimum. That is, we want to search for the rules with the "minimum credibility" in a particular set of data. Because the actual thresholds for these metrics vary depending on the dataset, the challenge is to design an algorithm that allows a user to specify these values and alter them as needed.

A rule r is said to be credible if it satisfies the following criteria:

- $support(r) \geq minSupport$

- *confidence(r)* $\geq$ *minconfidence*

where *minSupport* and *minconfidence* are user specified thresholds. One possible approach to identifying all "credible" rules by varying the *minSupport* and *minconfidence* from a given set of report vectors is to systematically evaluate all potential combinations of terms across the vectors. By assessing the support and, confidence of each combination, it is feasible to determine if they meet the required thresholds. However, this approach can be considered naive due to its exhaustive nature. This is demonstrated in Algorithm 1.

Algorithm 1 *Naive method for finding credible rules*

input: report vectors $\mathbb{R}$ and class values C
the user defined thresholds $minSupp$, $minConf$

output: a set of credible rules $\mathcal{R}$

1. $R \leftarrow \emptyset$;
2. **for** each c in C **do**
3. **for** each distinct $r : a_1, a_2, \ldots, a_k \rightarrow c$ **do**
4. **if** minSupp(r) $\geq$ minSupp AND minConf(r) $\geq$ minConf
5. $R \leftarrow R \cup r$
6. **return** R

Algorithm Steps

Step 1: [Initialize Rules Set] Initialize an empty set R to store the credible rules found.

Step 2: [Iterate over Class Values] For each class value c in C, process the following steps.

Step 3: [**Iterate over Distinct Rules**] For each distinct rule $r : a_1, a_2, \ldots, a_k \rightarrow c$ in the report vectors $\mathbb{R}$, where $a_1, a_2, \ldots, a_k$ are the antecedents of the rule and c is the consequent, process the following steps.

Step 4: [**Check Minimum Support and Confidence**] If the rule r has a minimum support greater than or equal to the user-defined minimum support and minimum confidence greater than or equal to the user-defined minimum confidence, proceed to the next step. Otherwise, continue to the next distinct rule r.

Step 5: [**Add Credible Rule to Set**] If the rule r meets the criteria established in step 4, then add the rule r to the set R of credible rules.

Step 6: [**Return Credible Rules**] After processing all class values in C and their associated rules in $\mathbb{R}$, return the set R of credible rules found.

Summary of Algorithm 1

This algorithm is straightforward and easy to understand. It goes through each distinct rule associated with each class value, validates them against user-defined thresholds for minimum support and confidence, and consolidates the credible ones into a set which is returned at the end. This method is considered "naïve" as it does not optimize for efficiency, and it checks each rule independently without leveraging any possible optimizations or prior knowledge.

The main issue discovered in Algorithm 1 is that every association between $a_1, a_2, \ldots, a_k$ will derive a set of rules in the form of $r1 : a_1, a_3 \rightarrow a_2$ or $r2 : a_1, c, a_2 \rightarrow a_3$ where in $r1$, a_1 and a_3 are the *antecedant* and a_2 is the *consequent*. As for $r2$, a_1, c and a_2 are the *antecedant* and a_3 is the *consequent*.

Using the example data presented in Table 3.1, if Algorithm 1 was implemented to derive set of credible rules, we might end up getting rules in the following format:

$$
\begin{aligned}
r_1&: \quad bug, break \rightarrow break \\
r_2&: \quad except, non-bug \rightarrow bug \\
r_3&: \quad compile, except \rightarrow bug \\
r_4&: \quad error \rightarrow problem \\
r_5&: \quad support, non-bug \rightarrow add \\
r_6&: \quad declare \rightarrow non\text{-}bug \\
r_7&: \quad allow \rightarrow nice
\end{aligned}
$$

Some the generated rules are not desirable for classification task especially rules r_1, r_4, r_5 and r_7. Those rules are derived since any association discovered in report vector $\mathbb{R}$ and class values C as long as the association meets the user defined thresholds of $minSupp$ and $minConf$.

In order to enhance the efficiency of Algorithm 1, we propose the incorporation of certain "more constrained" classes of rules. In contrast, rather than only identifying rules that meet minimal prerequisites, we will additionally construct criteria that differentiate certain rules as more preferable than others. This approach will assist in reducing the search space during computational processes. The subsequent sections delineate various categories of rules and diverse heuristics that have been formulated for their derivation.

3.2.2 Consequent Restricted Rules Generation

To derive a set of all credible consequent restricted rules, we use the *consequent restricted rules generation* algorithm shown in Algorithm 2 which is inspired by the CARM methodology [45, 42]. By using this method only rules with the consequent of class values C will be generated used for classification of issue reports.

Algorithm 2 *Finding Consequent Restricted Rules*

input: report vectors $\mathbb{R}$ and class values C
the user defined thresholds $minSupp$, $minConf$

output: a set of consequent restricted rules $\mathcal{R}$

1. $\mathcal{R} \leftarrow \emptyset$;
2. **for** each c_j in C **do**
3. $T_{c_j} \leftarrow \textsc{Select}(\mathbb{R}, c_j])$
4. $L_1 \leftarrow \{\ v \mid minSupp \leq |v|, v$ in the domain of $T_{c_j}\}$
5. **for** $(i = 1,\ L_i \neq \emptyset,\ i{+}{+})$ **do**
6. **for** each $t_1, t_2, \ldots, t_i$ in L_i **do**
7. **if** conf$((t_1, t_2, \ldots, t_i \rightarrow c_j) \geq minConf$
8. $\mathcal{R} \leftarrow \mathcal{R} \cup t_1, t_2, \ldots, t_i \rightarrow c_j$
9. $L_{i+1} \leftarrow \textsc{Generate}(L_i)$
10. **return** $\mathcal{R}$

The algorithm works as follows. Each distinct category c_j is considered in turn (step 2). For each c_j, we select the subset of vectors T_{c_j} from $\mathbb{R}$ that contain c_j as a category (step 3). We then extract single terms from T_{c_j} that have sufficient support (step 4), and we denote this set as L_1 and call it *large single terms*. Note that while we calculate support for each term in T_{c_j} only, the support calculation itself is based on the entire dataset $\mathbb{R}$, not T_{c_j}.

The algorithm then goes into iteration (step 5). Each association of i terms in L_i $(t_1, t_2, \ldots, t_i)$ is paired with category c_j to form a rule $(t_1, t_2, \ldots, t_i \rightarrow c_j)$ and we check if the rule has sufficient confidence (steps 6-7). Note that this confidence calculation needs no further scan of the dataset $\mathbb{R}$, as the support for $t_1, t_2, \ldots, t_i$ is already available, first from step 4 then from step 9 (see below), and the occurrence of $t_1, t_2, \ldots, t_i \rightarrow c_j$ can be obtained by a scan of T_{c_j}. Rules with sufficient confidence are retained and others are discarded (step 8).

Once rule generation is done for associations of i terms, the Generate function attempts to generate associations of $i + 1$ terms from the i terms, following the well-known apriori principle [2] (step 9). We shall briefly illustrate this principle here with an example, and the reader is referred to [2] for details. Suppose that we have a set of terms $\{a, b, c, d, e\}$ and a, b, d each is found to have sufficient support (following step 4), then the apriori principle guarantees that any superset containing c or e will not have sufficient support, hence can safely be ignored in the next round of computation and we only need to examine if (a, b), (a, d) and (b, d) will have sufficient support in the second round. This "bottom-up" rule generation is continued until no more associations of terms may be generated, and the set of derived rules is returned (step 10).

To show how our algorithm works, consider the reports given in Table 3.1 again. We consider two categories, `bug` and `non-bug`, in turn. For `bug`, we extract the following subset as T_{c_j}:

ID	Report	Category
1	error, bug, break, compile, support	bug
2	except	bug
5	null, except	bug
6	except, compile	bug
7	except, problem, compile	bug

We then count the support for each term in T_{c_j}. Suppose that the minimum support is 2. We obtain $L_1 = \{error, compile, support, except\}$. These terms are then used to form rules $\{error \rightarrow bug, compile \rightarrow bug, support \rightarrow bug, except \rightarrow bug\}$, and we check if they have the required confidence. Suppose that minimum confidence is 50%. We derive $\{compile \rightarrow bug, except \rightarrow bug\}$. Note that support must be calculated over the entire dataset, not just those in T_{c_j}.

After this, the algorithm goes into the second round, attempting to find associations having two terms to form rules. Only $((compile, except))$ and $((error, support))$ has sufficient support and by checking confidence the rule $compile, except \rightarrow bug$ also has the minimum confidence. So the rule is retained in $\mathcal{R}$. No more associations of terms may be generated, and computation involving the `bug` category is complete.

Apply the same to the `non-bug` category, we derive additional rules. At the end of the computation we obtain the following set of rules:

r_1: $compile \rightarrow bug$

r_2: $except \rightarrow bug$

r_3: $compile, except \rightarrow bug$

r_4: $error \rightarrow non\text{-}bug$

r_5: $support \rightarrow non\text{-}bug$

r_6: $declare \rightarrow non\text{-}bug$

r_7: $allow \rightarrow non\text{-}bug$

r_8: $allow, support \rightarrow non\text{-}bug$

r_9: $declare, support \rightarrow non\text{-}bug$

r_{10}: $error, support \rightarrow non\text{-}bug$

r_{11}: $error, support, declare \rightarrow non\text{-}bug$

3.2.3 Report Classification

We now consider how the derived rules may be used to classify issue reports. Different from decision trees or classification rules derived by traditional methods [79], multiple rules generated by our association mining may be fired during the classification process. For example, suppose that we have a report $V = \langle compile, except, error, ? \rangle$ to classify. It is easy to see that rules r_1, r_2, r_3, r_4 would apply, thus may assign multiple categories to a single report. To deal with this situation, we resort to *majority vote*. This is shown in Algorithm 3.

Algorithm 3 *Majority Vote Classification*

input: a set of restricted consequent credible rules $\mathcal{R}$ and
a vectorised issue report $V = \langle t_1, t_2, \ldots, t_k \rangle$

output: the category of V

1. **for** each rule $r : t_1, t_2, \ldots, t_k \rightarrow c_j$ in $\mathcal{R}$ **do**
2. **if** V *covers* r
3. $Count_{c_j}$++
4. $S \leftarrow \max_{\forall c_j}(Count_{c_j})$
5. **if** $|S| > 1$
6. **return** "unable to classify"
7. **else**
8. **return** c_j associated with S

We take each derived rule $r : t_1, t_2, \ldots, t_k \rightarrow c_j$ in turn (step 1). If the vector to be classified (V) covers r, i.e. every term in the antecedent of r appears in V (step 2), then r is fired and we increase the counter for category c_j by 1 (step 3). Once all the rules are checked, we consider the categories that have the largest counts (step 4). When there is a clear winner (i.e. there is a single largest count), this category will be returned as the category for the report. If there is a tie,

then our method will report that it is unable to classify the vector. This point is important and we will explain further in the experiment section in the thesis.

To illustrate how our majority vote based classification works, consider the following 3 rules, where ? indicates the category to be determined:

$$\begin{aligned} V_1\text{:} &\quad \langle compile, ? \rangle \\ V_2\text{:} &\quad \langle compile, except, error, ? \rangle \\ V_3\text{:} &\quad \langle compile, declare, ? \rangle \end{aligned}$$

For vector V_1, only rule r_1 will fire, so $Count_{bug} = 1$ and $Count_{non\text{-}bug} = 0$ and the report is classified as a `bug` report. For Vector V_2, rules r_1, r_2, r_3, r_4 will fire and we have $Count_{bug} = 3$ and $Count_{non\text{-}bug} = 1$. Our majority vote will select `bug` as the category for this report. Finally for V_3, rules r_1, r_6 will fire, so we end up with $Count_{bug} = 1$ and $Count_{non\text{-}bug} = 1$. Since we have a tie between two categories, our method will return "unable to classify" as a result.

Rather than seeking dominant patterns, as is commonly done in existing methodologies, our methodology involves identifying all plausible patterns in the form of term associations from issue reports. These patterns are then used to generate several rules. Although individual rules may not exhibit significant "strength," particularly when the minimum support is set to a low value, their cumulative application aids in accurately classifying issue reports, as demonstrated by the examples given.

Our approach can be interpreted as constructing an ensemble classifier. In the context of classifying a given report, each fired rule can be viewed as an individual classifier. The collective decision-making process is then achieved through a majority vote, effectively functioning as an ensemble.

In this approach, the extent of disagreement—meaning the specific degree or frequency with which certain rules support opposing classifications—is not considered significant. This is because the method relies on the presence of a majority

classification rule rather than weighting or quantifying each individual rule's contribution to the classification. By focusing on the final count of rules supporting each category, we simplify the decision-making process, ensuring consistency and interpretability without the added complexity of measuring partial disagreement levels. This design choice also aligns with the goal of reducing ambiguity in classification, where only a clear majority, rather than nuanced disagreement levels, determines the final category.

Our approach can also be classified as a form of deep learning. In our methodology, it can be inferred that each rule generated can be interpreted as encapsulating certain bits of information. Subsequently, our majority voting mechanism combines these pertinent fragments at a more comprehensive level. The present study asserts that our proposed methodology possesses considerable importance and originality. Moreover, it demonstrates exceptional efficacy in addressing scenarios wherein some keywords may be linked to various meanings or when one issue report encompasses multiple distinct problems.

3.3 Summary

In this chapter, we present a novel approach to generating credible rules for classification tasks. We have defined measures for discovering rules and other essential notions in this study.

A ggeneral-to-specificsolution and the difficulties of implementing it have been defined. Furthermore, the concept of consequent restricted rules has been defined as a way to lower the general solution's complexity.

The majority vote approach was ultimately implemented for classifying issue reports. Our method is intentionally designed to return an 'unable to classify' outcome in cases where the number of rules fired for each category is equal. This decision was made to avoid arbitrary classification when evidence for multiple

categories is balanced, thus prioritising classification accuracy and reducing the risk of misclassification.

This approach differs from existing classification methods, such as Naive Bayes and Decision Tree algorithms, in its reliance on association rule mining to extract both frequent and less frequent (credible) patterns, rather than focusing solely on dominant, highly supported cases. Unlike conventional algorithms that prioritize highly supported patterns, our method intentionally incorporates lower-support rules to capture nuanced relationships in the data that may be missed by purely frequency-based classifiers. Furthermore, unlike standard rule-based methods that require manually defined rules, this approach leverages a data-driven rule generation process, making it adaptable to the dynamic nature of software issue reports without extensive manual input.

Although the algorithms presented in this chapter offer the advantage of capturing both dominant and credible patterns, they are limited since the algorithms presented treat every rule fired during classification as equally important, rather than weighing each rule based on its support count or strength level. This approach means that all rules, regardless of their frequency in the dataset, contribute equally to the classification decision. Consequently, lower-support rules, which may represent less common but significant patterns, can have the same influence on classification outcomes as higher-support (stronger) rules. This can lead to cases where the classifier is overly sensitive to less representative patterns, reducing overall accuracy and potentially causing misclassification.

This limitation shapes the scope of Chapter 4, where we explore approaches to address it by introducing rule-weighting mechanisms that factor in each rule's support count. These modifications aim to improve classification reliability by prioritising rules with higher support counts. They ensure that the classifier can better differentiate between common and uncommon patterns while reducing the risk of misclassification in ambiguous cases.

The subsequent chapter presents various heuristics developed to mitigate the constraints identified in our initial proposed methodology, which employed the majority vote technique.

Chapter 4

Support Based Voting Classification

In the preceding chapter, we presented our proposed methodology for classifying software issue reports using a method based on association rules mining, specifically inspired by the concept of majority vote. Based on our preliminary assessment, our approach will classify any report that receives an equal number of votes from the rules as "unclassifiable" or "unknown" in subsequent analysis.

This phenomenon occurs as a result of the approach we engineered, in which each rule that is fired is considered independently, without consideration to its level of *strength* determined by its *support* measure and *confidence* measure which determines rule *accuracy*. This situation is considered undesirable since it may result in a significant proportion of reports being classed as "unknown."

In certain instances, the classification of certain items as "unknown" may be left to human judgement. However, when a large number of items are classified as "unknown" the process might become burdensome.

As elaborated in Section 3.1.3, *Support* indicates the strength of a given rule. Hence, it is essential to treat every rule with different level of "weight" depending on the level its support. Therefore, this chapter presents an enhance version of *Algorithm 2 - Restricted Consequent Rules Generation* explained in Section 3.2.2 by including support count as part of the rules formation. Later, we introduced

a modified classification algorithm based on *Algorithm 3 - Majority Vote Classification* presented in Section 3.2.3 by incorporating support as part of decision factor and count them collectively during classification task.

4.1 Support Count in rules formation

This section presents the modified algorithm, which includes support count in the generated rules. The algorithm is as follows.

Algorithm 4 *Finding Consequent Restricted Rules with Support count*

input: report vectors $\mathbb{R}$ and class values C
the user defined thresholds $minSupp$, $minConf$

output: a set of consequent restricted rules $\mathcal{R}$ and the $SupportCount$ of every rule

1. $\mathcal{R} \leftarrow \emptyset$;
2. **for** each c_j in C **do**
3. $T_{c_j} \leftarrow \textsc{Select}(\mathbb{R}, c_j])$
4. $L_1 \leftarrow \{\ v \mid minSupp \leq |v|, v$ in the domain of $T_{c_j}\}$
5. **for** $(i = 1,\ L_i \neq \emptyset,\ i{+}{+})$ **do**
6. **for** each $t_1, t_2, \ldots, t_i$ in L_i **do**
7. **if** conf$((t_1, t_2, \ldots, t_i \rightarrow c_j) \geq minConf$
8. $\mathcal{R} \leftarrow \mathcal{R} \cup t_1, t_2, \ldots, t_i \rightarrow c_j$
9. $L_{i+1} \leftarrow \textsc{Generate}(L_i)$
10. **return** $\mathcal{R}, SupportCount$

4.1.1 Explanation of the Modified Algorithm

This algorithm targets to discover a set of consequent restricted rules, $\mathcal{R}$, based on given report vectors, $\mathbb{R}$, class values, C, and user-defined thresholds, $minSupp$

and $minConf$.

Inputs:

- Report vectors, $\mathbb{R}$, representing the dataset.
- Class values, C.
- User-defined thresholds, $minSupp$ and $minConf$, representing the minimum support and the minimum confidence, respectively.

Outputs:

- A set of consequent restricted rules, $\mathcal{R}$.
- The $SupportCount$.

Steps:

1. **Initialization:** The set $\mathcal{R}$ is initialized to an empty set.
2. **Iteration over Class Values:** For each class value c_j in C:
   (a) **Select Reports:** $T_{c_j} \leftarrow$ reports from $\mathbb{R}$ where the class is c_j.
   (b) **Initialize Level-1 Set:** L_1 is initialized with items that have a support $\geq minSupp$ in the domain of T_{c_j}.
   (c) **Generation of Itemsets:** For each level i, where $L_i \neq \emptyset$, increment i and do:
      i. **Iteration over Itemsets:** For each itemset in L_i, represented by $t_1, t_2, \ldots, t_i$:
      ii. **Check Confidence:** If the confidence of $(t_1, t_2, \ldots, t_i \rightarrow c_j)$ is $\geq minConf$, then:

iii. **Update Rule Set:** $\mathcal{R}$ is updated by adding the new rule $t_1, t_2, \ldots, t_i \rightarrow c_j$.

(d) **Generate Next Level Itemsets:** L_{i+1} is generated from L_i.

3. **Return Results:** Return the set $\mathcal{R}$ and the *SupportCount*.

Conclusion:

The algorithm generates rules by iteratively increasing itemset levels and checking them against user-defined thresholds for *support* and *confidence*. Only the rules that satisfy these thresholds are added to $\mathcal{R}$. Finally, $\mathcal{R}$ and *SupportCount* are returned.

Implementing the Algorithm 4 with the same set of vectorised reports in Table 3.1 from Section 3.2, will derive the following set of rules. Please note that we set the minimum support to 2 and minimum confidence to 50%. The rules are represented in the form of $r : t_1, t_2 \ldots, t_k \rightarrow c_j, SupportCount$. Where *SupportCount* is a numerical value indicate the strength of a given rule.

r_1: $compile \rightarrow bug, 3$

r_2: $except \rightarrow bug, 4$

r_3: $compile, except \rightarrow bug, 2$

r_4: $error \rightarrow non\text{-}bug, 2$

r_5: $support \rightarrow non\text{-}bug, 4$

r_6: $declare \rightarrow non\text{-}bug, , 2$

r_7: $allow \rightarrow non\text{-}bug, 2$

r_8: $allow, support \rightarrow non\text{-}bug, 2$

r_9: $declare, support \rightarrow non\text{-}bug, 2$

r_{10}: $error, support \rightarrow non\text{-}bug, 2$

r_{11}: $error, support, declare \rightarrow non\text{-}bug, 2$

4.2 Support based Voting

Support based voting incorporate support count of every rule generated in *Algorithm 4*. The support count refers to the number of times a particular rule or pattern appears within the dataset, indicating its frequency. In this context, a higher support count means the rule is based on a pattern that occurs frequently across issue reports, while a lower support count indicates a less common but potentially significant pattern.

Hence, this section presents the modification of *Algorithm 3 - Majority Vote Classification Algorithm* where we count the vote based on support count a given rule as opposed to treat the rule individually. The modified algorithm is as follows.

4.2.1 Explanation of Algorithm 5 - Using Support as Weight

This algorithm classifies elements of a given vectorized issue report V into one of three categories: "b" (bug), "nb" (non-bug), or "unknown", based on a set of provided rules $\mathcal{R}$. Each rule r in $\mathcal{R}$ consists of a set of elements and an associated category ("b" or "nb"), and also has a support value indicating the strength or frequency of the rule.

1. **Input:**
   - $\mathcal{R}$: A set of rules, where each rule $r : r_1, r_2, \ldots, r_m \rightarrow c_j$ is associated with a category c_j ("b" or "nb") and has a support value.
   - V: A vectorized issue report, represented as a list $V = \langle t_1, t_2, \ldots, t_k \rangle$ where each t is a transaction.
2. **Initialization:** For each transaction t in V, initialize counters $cBug$ and $cNonBug$ to 0. These counters will be used to accumulate the support of rules that are covered by t.

Algorithm 5 *Using Support as Weight*

input: a set of restricted consequent credible rules $\mathcal{R}$ with *SupportCount* s value and a vectorised issue report $V = \langle t_1, t_2, \ldots, t_k \rangle$

output: the category of V

1. **for** each $t \in V$ **do**
2. Initialize $cBug$ and $cNonBug$ to 0
3. **for** each rule $r : r_1, r_2, \ldots, r_m \rightarrow c_j$ in $\mathcal{R}$ **do**
4. **if** $r - \{c_j\} \subseteq t$
5. **if** c_j is "b"
6. $cBug = cBug+$ *SupportCount* s
7. **else**
8. $cNonBug = cNonBug+$ *SupportCount* s
9. **if** $cBug = cNonBug$
10. Append "unknown" to t
11. **else if** $cNonBug > cBug$
12. Append "nb" to t
13. **else**
14. Append "b" to t
15. **return** V

3. **Processing Each Rule in $\mathcal{R}$:** For each rule $r : r_1, r_2, \ldots, r_m \rightarrow c_j$ in $\mathcal{R}$, check whether the left-hand side of r (i.e., $r - \{c_j\}$) is a subset of the transaction t. If it is, and if the associated category c_j of the rule is "b", then add the support of the rule to $cBug$. If the associated category c_j of the rule is not "b" (i.e., it is "nb"), then add the support of the rule to $cNonBug$.

4. **Classifying Each Transaction:** After processing all the rules for a given transaction t, compare the accumulated support in $cBug$ and $cNonBug$. If

$cBug = cNonBug$, then the classification is inconclusive, and "unknown" is appended to t. If $cNonBug > cBug$, then "nb" is appended to t, classifying it as a non-bug. If $cBug > cNonBug$, then "b" is appended to t, classifying it as a bug.

5. **Output:** The algorithm returns the modified vectorized issue report V, where each transaction t has been appended with its classification ("b", "nb", or "unknown").

Example: Suppose we have a vectorized issue report $V = \langle t_1, t_2 \rangle$ and a set of rules $\mathcal{R} = \{r_1 : \{a, b\} \rightarrow \text{"b"}, r_2 : \{c\} \rightarrow \text{"nb"}\}$ with the support of r_1 as 2 and the support of r_2 as 1. If $t_1 = \{a, b, c\}$ and $t_2 = \{c\}$, then for t_1, $cBug$ would be 2, and $cNonBug$ would be 1, classifying it as a bug. For t_2, only $cNonBug$ would be incremented, classifying it as a non-bug. The returned V would be $V = \langle \text{"b"}, \text{"nb"} \rangle$.

Using the previous example explained in Section 3.2.3 we attempt to classify the following reports using *Algorithm 5*.

$$
\begin{aligned}
V_1&: \quad \langle compile, ? \rangle \\
V_2&: \quad \langle compile, except, error, ? \rangle \\
V_3&: \quad \langle compile, declare, ? \rangle
\end{aligned}
$$

From our previous attempt using *Majority Vote Algorithm* for classifying the aforementioned reports, we managed to classify V_1 as *bug*, V_2 as *bug* and V_3 as "unable to classify" or "unknown".

When classifying V_1 using the *Augmented based Vote algorithm* only one rule will be fired which is r_1. With total support of 4, we classify V_1 as *bug*. As for V_2, the following rules will be fired and used for classification of V_2.

r_1: $compile \rightarrow bug, 3$

r_2: $except \rightarrow bug, 4$

r_3: $compile, except \rightarrow bug, 2$

r_4: $error \rightarrow non\text{-}bug, 2$

From the rules fired, r_1, r_2, and r_3 are all *bug* rules with with support count of 3, 4, and 2 respectively. On the contrary, r_4 constitutes 2 support count. Hence, $Count_{bug} = 9$ and $Count_{non\text{-}bug} = 2$. Our *support based vote* will select *bug* as the category for this report.

Finally for V_3, rules r_1, and r_6 will fire, so we end up with $Count_{bug} = 3$ and $Count_{non\text{-}bug} = 2$. Hence, V_3 will be classified as *bug* where the same report was classified as "unknown" using our *Algorithm 3 - Majority Vote* in the previous chapter. since we have a tie between two categories.

4.3 Summary

This chapter has presented modified algorithms which were initially presented in Chapter 3. The first modified algorithms include *support* count as part of the output for generating restricted consequent rules described in Section 4.1. Later, these rules then serve as input for classification by accumulating the support count explained in Section 4.2.

The approach presented in this chapter differs from conventional classification methods, such as Naive Bayes, Decision Trees, and Support Vector Machines (SVM), in several key ways. Unlike traditional methods that rely primarily on highly supported patterns, this approach incorporates both dominant and credible rules. This inclusion allows the classifier to handle more complex cases accurately, as it can recognise patterns that occur less frequently but still have strong predictive value. Additionally, while Naive Bayes and Decision Trees require well-defined, pre-selected features, our method's use of association rule mining with

a support-based approach enables flexible, adaptive rule generation directly from the dataset's unique patterns.

Furthermore, this approach offers greater interpretability through its use of association rules, which are human-readable and provide clear insight into the classification logic. This contrasts with the less transparent decision boundaries found in methods like SVMs. By combining both dominant and credible patterns and integrating a support-based mechanism, this method seeks to balance classification accuracy with interpretability, making it especially applicable for issue report classification where understanding the basis of decisions is essential.

The following chapter evaluates the impact of implementing our initial proposed approach described in Chapter 3 and 4 through series of experiments using benchmark datasets.

Chapter 5

Experiments and Results

This chapter presents a sequence of empirical investigations conducted to assess the efficacy of the methodology proposed herein for resolving the issue discussed in this thesis. The experimental design scrutinizes the impact of rule generation with varying thresholds of *minSupport* and *minConfidence*, as elaborated in Chapters 3 and 4.

The following section describes the experimental data we used. Later, we elaborate in details the steps taken to pre-process our data by applying feature selection. We proceed by conducting a series of experiments to assess the suggested method's quality, by comparing our method to those used by Terdchanakul et al [77] and Pingclasai et al [62] in their investigations.

5.1 Data Description

We used datasets from Herzig et al [25] in our experiments. These datasets were extracted from an Issue Tracking System[1], and comprise of three Open Source Software (OSS) projects, *Http-Client*, *Jackrabbit* and *Lucene*. In total, there are 5590 issue reports and their distribution are shown in Table 5.1.

We adopted the same strategy practiced by Terdchanakul et al [77] and Pingclasai et al [62] by selecting 90% of oldest issue reports as training set and used

[1]https://www.atlassian.com/software/jira

Table 5.1: Dataset

Project	# of Reports	# of Bugs	# of Other Request
Http-Client	745	305	440
Jackrabbit	2402	938	1464
Lucene	2443	697	1746
Cross Project	5590	1940	3651

the remaining 10% newest issue reports for testing. We chose to use this hold-out testing strategy as our dataset is relatively large. The distribution between training and testing set are illustrated in Table 5.2 and 5.3.

Table 5.2: Issue Reports used for training and testing bug reports

Project/Type	# of Bugs Report	# Training Bugs	# Test Bugs
Http-Client	305	275	30
Jackrabbit	938	844	94
Lucene	697	627	70
Cross Project	1940	1746	194

Table 5.3: Issue Reports used for training and testing other reports

Project/Type	# of Other Reports	# Training Other	# Test Other
Http-Client	440	396	44
Jackrabbit	1464	1318	146
Lucene	1746	1571	175
Cross Project	3650	3285	365

In addition to *Hold-Out* evaluation [78, 70], we also used *10-fold Cross Validation* to make sure every single data point in the dataset is tested. Cross-validation was implemented by splitting the data set into k folds, where each fold respects the temporal sequence of the issue reports. This ensured that reports from earlier

periods were used in training, while later reports were reserved for testing, simulating real-world scenarios where classifiers encounter new data chronologically [37, 7].

5.2 Data Preparation

The initial step involves data preprocessing. The significance of this matter lies in the fact that software issue reports are typically composed in unstructured text format and necessitate vectorisation prior to being processed by our proposed solution. There exist a multitude of natural language processing (NLP) techniques that can be utilised for the purpose of vectorising issue reports.. This approach is in line with earlier research [4, 86, 59, 19], we employ the NLTK toolkit [47] and Scikit-learn package [61].

Without loss of generality, we assume that a software issue report R is represented as a sequence of terms separated by spaces and a manually annotated category: $R = \langle t_1\ t_2\ \ldots\ t_n, C\rangle$, where term $t_i, 1 \leq i \leq n$. The terms t_i in R include words extracted from the text of the issue reports, which can also include **code-related terms** such as variable names, function calls, and code fragments. Furthermore, numbers and special characters attached to words (e.g., `variable1`, `func_call()`) are retained as part of the tokenised terms, reflecting their importance in identifying patterns and C is the category. For example, the following represents an issue report that has 9 terms and a category `bug`:

$R = \langle$ *I got errors and warning when executed classException() method,* **bug**$\rangle$

Not all terms in a report are relevant or appropriate for developing our classifier. Consequently, data cleaning and preprocessing are imperative to ensure the quality and suitability of the data. The subsequent subsections will provide a comprehensive explanation of the procedures used to preprocess the data for the proposed methodology.

5.2.1 Text Normalisation

Text normalisation is a process of wrangling, cleaning, and standardising textual data into a form that could be consumed by machine learning (ML) algorithms as input. Text normalisation involves a variety of tasks such as tokenisation, case conversion, spelling correction, stopwords and special character removal, stemming, and lemmatization. To illustrate, the stopwords process is carried out in the preparation of our data; we use the example given in Figure 5.1.

Item	Content
Project	HttpClient
Bug ID	HTTPCLIENT-8
Title	`LogSource.setLevel` incorrectly uses entrySet

Description	When I call `LogSource.setLevel`, I'd get the following exception:
	
	`java.lang.ClassCastException: java.util.HashMap$Entry at org.apache.commons.httpclient.log.LogSource.setLevel (LogSource.java:158)`
	
	The Calling code is:
	
	`LogSource.setLevel (Log.OFF);`
	
	The error's (I believe) that you should get the value set from the map, not the entry set (in LogSource):
	
	`static public void setLevel(int level) {`
	`  Iterator it = _logs.entrySet().iterator();` <-- should be `_logs.values()`
	`  while(it.hasNext()) {`
	`  Log log = (Log)(it.next());`
	`  log.setLevel(level);`
	`  }`
	`}`

Figure 5.1: Example Issue Report Before Text Normalization

We focus on the *Title* and *Description* elements of the report as they contain most useful information. However, both contain terms that are unsuitable for our method to process, for example, special characters, words written in camelCase (e.g. *setLevel*, *LogSource* and *ClassCastException*), code snippet, words with little semantic content for our intended analysis (e.g. *The*, *from*, *that*, *you* etc), and words written in contraction form (e.g. *I'd* and *error's*). So they need to be dealt with in text normalization.

The splitting of reports into individual words was carried out to simplify the repre-

sentation of text data, allowing efficient pattern detection and feature extraction. Maintaining longer entities, such as phrases or complete sentences, would increase dimensionality and computational complexity without necessarily improving classification accuracy, given the unstructured nature of issue reports.[84, 30, 70]

Figure 5.2 shows the steps we take in our work to normalise the free text in issue reports. In the following we will introduce these steps conceptually first, and then describe a Python implementation of these steps later. We will use the issue report in Figure 5.1 to illustrate how an issue report is transformed using the proposed text normalization process.

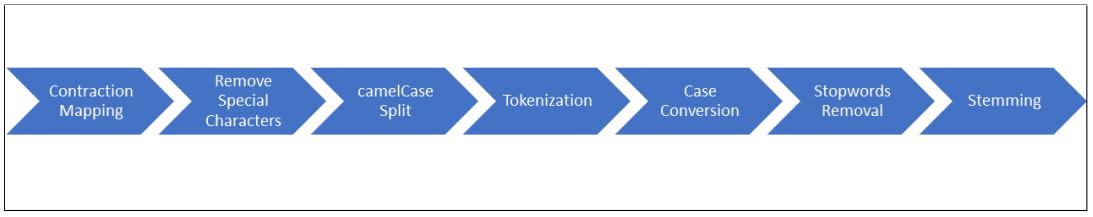

Figure 5.2: Text Normalization Framework

Contraction Mapping

Contractions are created by removing certain letters from words. Most of the English contractions are created by eliminating some vowels from a word, for example, *are not* is shortened to *aren't* and *shall not* to *shan't* where an apostrophe is used to denote the contraction and some letters are removed. Various forms of contraction exist and they pose problems for text analytics because we have an apostrophe character in a word, and as we have two words represented by a contraction, tokenization will not be done correct, that is words like *won't* will be treated as single token instead of two (*will not*).

Therefore, contractions need to be pre-processed before other text processing being carried out. To do so, we have used a contraction mapping program in Python to remove contractions. Using this tool, we can transform *I'd* and *error's*

found in our example issue report given in Figure 5.1 into *'I' 'would'* and *'error' 'is'* as required.

Removing Special Characters

The next task to perform is to remove special characters. Quite often and in any text corpora, symbols, numbers and punctuation occur in sentences. It is useful and necessary to remove these special characters, since they do not much use for text analysis. As can be seen in Figure 5.1, our example report contains a lot of unnecessary symbols. This is because code snippets and log messages are typically present in a software issue report, hence symbols such as $, *semicolon*, *braces*, *parentheses* and *arithmetic* are commonly found in our dataset.

We use a regular expression package `RegEx` available in Python to remove all these special characters. In addition to symbols, we also remove numbers found in our data since they do not provide any useful information either to our proposed method.

camelCase Split

One special characteristic of our text data is that they can include code snippets which are typically written in `Java`. It is customary that method names in Java are written in the form of camelCase, for example, `setLevel`, `ClassCastException` and `HashMapEntry` in our example. These method names need to split in order to obtain more meaningful terms from the reports. We have implemented a camelCase splitting tool using the regular expression package `RegEx` available in Python, which when applied to our example report, the following camelCase expressions were successfully converted into normal text.

- **LogSource** converted into `Log` and `Source`.

- **setLevel** converted into `set` and `Level`.
- **ClassCastException** converted into `Class`, `Cast` and `Exception`.
- **hasNext** converted into `has` and `Next`.
- **entrySet** converted into `entry` and `Set`.

Tokenisation

Tokenisation is a step that splits longer strings of text into smaller pieces, or tokens. Larger chunks of text can be tokenised into sentences, sentences can be tokenised into words, etc. Text analysis is normally performed on a piece of text has been appropriately tokenized. Tokenisation is also referred to as text segmentation or lexical analysis. While segmentation is sometimes used to refer to the breakdown of a large chunk of text into pieces larger than words (e.g. paragraphs or sentences), tokenisation is reserved for the breakdown process that will result in words. We have chosen to conduct tokenisation after dealing with other text normalisation issues described in the previous subsections, and we have used a package available in Scikit-learn [61] to tokenise issue reports in our study.

Case Conversion

Once the text is tokenized we perform case conversions. This is done in order to make text analysis easier, for example, when performing word matching. We have chosen to convert all text to lowercase, similar to most research in text analytics.

Stopwords Removal

Stopwords are words that carry little semantic meaning. Having these words in text data will create bias in analysis since the frequency of their appearance in a

corpus can be very high. Therefore, stopwords are usually removed during text pre-processing so that only words that are significant to and provide context for the intended analysis are retained. In our example, for instance, words like *I*, *call*, *is*, *believe*, *is*, *it* and *not* will be removed from the data. However, it is useful to note that we should not over-removing stopwords since words like *can* and *cannot* can imply two different meanings depending on the context of sentences. For our experiments, we create our own list of stopwords since we do not want to reduce too many words that will impact the performance our classifier later. The stopword exception list was curated by manually reviewing high-frequency terms in the dataset to identify domain-specific words critical for classification, such as *error*, *bug*, and *fix*. These terms were retained despite their frequent occurrence, as they carried significant semantic weight. The final list of exceptions included technical terms such as *function*, *module*, and *patch*, which were relevant for the classification of software issues. The rest of the stopwords list were taken from the Natural Language Tool Kit [47].

Stemming

Understanding stemming necessitates an understanding of what word stems are. This requires understanding of morphemes, the smallest independent unit in any natural language. Morphemes are composed of stems and affixes. Affixes, like prefixes and suffixes, are units that are attached to a word stem to change its meaning or create a new word entirely. Word stems are also known as a word's `base form`, and they can be used to create new words by attaching affixes to them in a process known as *inflection*. The opposite of this is known as *stemming*, which is the process of obtaining the base form of a word from its inflected form.

Take a look at the word *FOLLOW*. We can add affixes to it to create new words like *FOLLOWS*, *FOLLOWED* and *FOLLOWING*. The word stem in this case is the base word FOLLOW. We can get back to the base form by stemming any of

its three inflected forms, as shown in Figure 5.3.

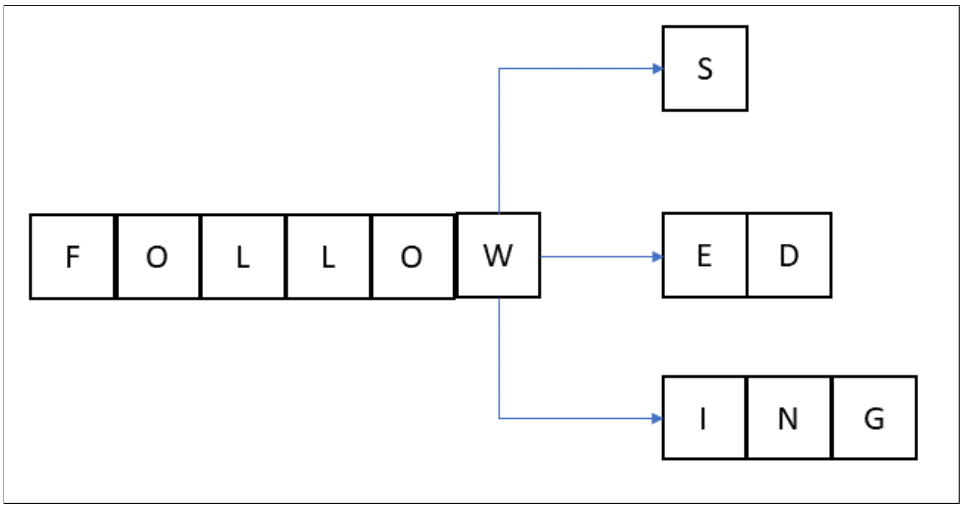

Figure 5.3: Word stem and inflections

The diagram depicts how the word stem is represented in all of its inflections because it serves as the foundation upon which each inflection is built using affixes. Stemming helps standardise words to their base stem regardless of inflections, which aids many applications such as text classification and clustering, where frequency of word occurrence is important. We used Porter stemmer as our stemming technique for our dataset, which has been widely used in previous studies [4, 77, 59, 14, 43].

Summary of Text Normalization

In this section we have described our text normalisation process. We summarise this process in Algorithm 6, which has been implemented in Python.

Applying our text normalisation algorithm to the example issue report given in Figure 5.1, we obtain the following vector:

⟨ *apach, believ, call, cast, class, code, common, descript, entri, error, except, follow, get, hash, httpclient, incorrectli, int, iter, java, lang, level, log, map, next, not, off, org, public, set, should, sourc, static, use, util, valu, void* ⟩

Algorithm 6 *Text Normalisation*

input: a set of reports $\mathcal{D} = \langle R_1, R_2, \ldots, R_k \rangle$ where $R_i = \langle w_1, w_2, \ldots, w_k \rangle$

output: a set of vectorised issue report $V = \langle T_1, T_2, \ldots, T_k \rangle$

1. $V \leftarrow \emptyset$
2. **for** each report R in $\mathcal{D}$ **do**
3. **for** each term w in R **do**
4. **if** w is a *contracted word* **then**
5. $w \leftarrow$ contraction mapping(w)
6. **if** w is a *special character* **then**
7. $R \leftarrow$ remove(w, R)
8. **if** w is a *camelCase* word **then**
9. $w \leftarrow$ split(w)
10. $T \leftarrow$ tokenize(R)
11. **for** each token t in T **do**
12. **if** t is a *stopword* **then**
13. $T \leftarrow$ remove(t, T)
14. $t \leftarrow$ stemming(t)
15. $V = V \cup T$
16. **return** V

5.2.2 Feature Selection

Once the issue reports have been pre-processed by text normalisation described in Section 5.2.1, we carry out dimension reduction or feature extraction on the resulting vectors. That is, we select a fixed number of terms (features) to represent the vectors and use the reduced data as a training set.

Feature selection was necessary to eliminate irrelevant or redundant terms from the dataset, ensuring that the classification model focused on the most informative

features. This step reduces noise in the data, enhances model interpretability, and minimises the risk of overfitting, particularly when working with high-dimensional text data.[24, 46, 80, 21]

We define a training set as the subset of issue reports used to generate rules and train the classification model. This dataset provides labelled examples that enable the model to learn patterns and associations relevant to the classification task.

In our experiments, we have considered three feature selection methods.

Selection by frequency

With this method, we choose terms based on the frequency of their occurrence in the given set of reports or corpus. That is, we set a frequency threshold and discard any terms whose occurring frequency are less than that threshold, and represent the dimension-reduced vectors as shown below.

Table 5.4: Frequency Table

ID	add	error	exception	import	modify	null	remove	warn	Category
1	0	2	3	0	0	3	1	1	bug
2	0	1	1	0	0	3	0	0	bug
3	0	3	4	1	0	0	0	0	bug
4	2	0	0	1	1	0	4	2	non-bug
5	4	0	0	1	3	0	0	2	non-bug
6	3	0	1	0	3	0	3	0	non-bug

Note, if we want to select the four most frequent terms based on the frequency table above, we will end up selecting `add, exception, modify` and `remove`. This dimension reduction should help efficiency without compromising classification

accuracy, as it is well established that not all terms, especially those occurring not frequently, will have little impact on filtration [73, 58, 54].

Selection by TF-IDF

Feature selection based on terms frequency will lead to some potential threats. Some terms might occur frequently but less significant and it will overshadow other terms in the feature set. Especially terms that do not occur frequently enough, but might be more interesting and effective to be identified as feature. Therefore, we employ TF-IDF (*term frequency-inverse document frequency*) as introduced by Salton et al [68]. It is a numerical statistic which reflects how important a feature is to a document in a corpus and widely employed to deal with text analysis and information retrieval [82].

Let R represents a collection of issue reports, and $d_r \in R$ is an issue in R. The term frequency $tf(t_i, d_r)$, is the occurrences of term t_i appeared in document d_r. However, in some cases a term would appear more frequently in long documents compared to a shorter one. Thus, TF is normalised as [50]:

$$TF = \frac{f(t_i, d_r)}{\sum t, d_r} \tag{5.1}$$

where

- f_{w_D} is the frequency of term w in document D
- $\sum t_i, d_r$ is the total of terms in document d_r

Hence, once Equation(5.1) applied to Table 5.4, the following Term Frequency table will be derived.

Next, let R_{t_i} represents a set of issue reports in R that contain the term t_i. Table 5.6 shows the total number of documents contained term t_i

Table 5.5: Term Frequency Table

ID	add	error	exception	import	modify	null	remove	warn	Category
1	0	0.2	0.3	0	0	0.3	0.1	0.1	bug
2	0	0.2	0.2	0	0	0.6	0	0	bug
3	0	0.375	0.5	0.125	0	0	0	0	bug
4	0.2	0	0	0.1	0.1	0	0.4	0.2	non-bug
5	0.4	0	0	0.1	0.3	0	0	0.2	non-bug
6	0.3	0	0.1	0	0.3	0	0.3	0	non-bug

Table 5.6: Document Frequency Table

terms	add	error	exception	import	modify	null	remove	warn
DF	3	3	4	3	3	2	3	3

and D denotes the total number of documents in R which is 6. Hence, the inverse document frequency idf of term t_i in R is calculated as

$$idf(d,t) = log\left[\frac{(1+D)}{(1+df(d,t))}\right] + 1 \tag{5.2}$$

Note that, the constant 1 is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions [50, 8]. When Equation(5.2) is applied on Table 5.6, the following result will derive as depicted in Table 5.7

Table 5.7: Inverse Document Frequency Table

terms	add	error	exception	import	modify	null	remove	warn
IDF	1.5596	1.5596	1.3365	1.5596	1.5596	1.8473	1.5596	1.5596

The complete equation of TF-IDF is depicted in Equation(5.3) below.

$$TF - IDF(t_i, d_c) = tf(t_i, d_c) \times log\left[\frac{(1+D)}{(1+df(d,t))}\right] + 1 \tag{5.3}$$

Once applied, the following table will derive:

Table 5.8: TF-IDF Table (Without Normalisation)

ID	add	error	exception	import	modify	null	remove	warn	Category
1	0	0.3119	0.4009	0	0	0.5542	0.1560	0.1560	bug
2	0	0.3119	0.2673	0	0	1.1084	0	0	bug
3	0	0.5849	0.6682	0.1950	0	0	0	0	bug
4	0.3119	0	0	0.1560	0.1560	0	0.6238	0.3119	non-bug
5	0.6238	0	0	0.1560	0.4679	0	0	0.3119	non-bug
6	0.4679	0	0.1336	0	0.4679	0	0.4679	0	non-bug

As suggested by [8] an Euclidean normalisation is needed to smoothen the weighted score. Hence, the following equation must be applied for every report in the corpus.

$$v_{norm} = \frac{v}{\|v\|} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}} \tag{5.4}$$

First, we will calculate the Euclidean distance using this equation $\sqrt{v_1^2} + v_2^2 + \ldots + v_n^2$ and the following normalisation for each document will derive.

Table 5.9: Normalisation by Euclidean Distribution

Rep	1	2	3	4	5	6
L2 Value	0.7835	1.1821	0.9092	0.7953	0.8542	0.8213

Next, we will divide every weighted score in Table 5.8 with the Euclidean normalisation value as in Table 5.9. Our final TF-IDF vector table will be as follow:

Table 5.10: TF-IDF Table

ID	add	error	exception	import	modify	null	remove	warn	Category
1	0	0.3981	0.5118	0	0	0.7074	0.1991	0.1991	bug
2	0	0.2639	0.2261	0	0	0.9377	0	0	bug
3	0	0.6433	0.7350	0.2144	0	0	0	0	bug
4	0.3922	0	0	0.1961	0.1961	0	0.7845	0.3922	non-bug
5	0.7303	0	0	0.1826	0.5477	0	0	0.3651	non-bug
6	0.5697	0	0.1627	0	0.5697	0	0.5697	0	non-bug

Chi-square(X^2)

Chi2 is widely used on text data [52]. It is a measure for modelling the dependency between the features and the classes. More formally, for a given vectorised issue report $\mathbb{R} = \{T_1, T_2, \ldots, T_m\}$, we estimate the following quantity of each term $T_1, T_2 \ldots, T_m$ and rank them by their score. Chi2 weighs the score with the following Equation(5.5):

$$X^2(D, t, c) = \sum_{e_t \in [0,1]} \sum_{e_c \in [0,1]} \frac{(N_{e_t e_c})(E_{e_t e_c})^2}{(E_{e_t e_c})} \tag{5.5}$$

where

- D is an issue report, t terms appear in the issue report, and c is the category of the report.
- N is the observed frequency and E the expected frequency.
- e_t takes the value 1 if the document contains term t and 0 otherwise.
- e_c takes the value 1 if the document is in class c and 0 otherwise.

For each term t_i, a corresponding of low Chi2 score indicates the null hypothesis H0 of independence. Which means, the document class has no influence over the term's frequency, thus should be rejected. Whilst should the score is high, it indicates the occurrence of the term and class are dependent. In this case, we should select the term as feature for classification.

5.3 Performance Measures

To evaluate our proposed method, we apply the standard `Precision`, `Recall`, `F` measures, where TP stands for True Positive, FP False Positive, FN False Negative and TN True Negative:

$$Precision = \frac{TP}{TP + FP} \tag{5.6}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.7}$$

$$F = 2 \times \frac{Recall \times Precision}{Recall + Precision} \tag{5.8}$$

While these measures are commonly used in evaluating classification of software issue reports [4, 5], their application in our study needs to adjusted. This is due to a unique feature of our method: when a tie is produced following the vote, our method has an option to report the case as *unclassified* or its class is *unknown*. We will explain how we adapt these standard measure to take unknown cases into account in the following sections when we analyse the experiment results.

5.4 Evaluation Approach

We conduct two sets of tests to evaluate our proposed method:

1. `Hold-out Evaluation.` With this method, a dataset is randomly split into two parts: the *Training* set and the *Testing* set. We vary the *Training* and *Testing* ratio in our experiments.

2. `Cross-Validation Evaluation.` With this method, a dataset is randomly partitioned into k equal parts, one used as the *Testing* set and the remaining parts as the *Training* set. The testing is then repeated k times, with each of the k parts used as the *Testing* set exactly once, and the k results will be averaged to produce one single estimation.

In evaluating our approach using these two methods, we use the following datasets:

1. `Individual Project.` We test our algorithm using individual project data only. That is, we train on data from one project and test the algorithm on the same project.

2. `Cross Projects.` We test our algorithm using a mixture of data from all three projects. As each project is likely to use some specific terms, using cross-project data would test the robustness of our method.

5.5 Classification Experiment using Method 1

This Section presents the results of a series of experiments to evaluate the performance of the developed methods in a classification task. For the purpose of this study it is a *binary* classification to determine whether or not a given issue report is a *bug* or *non-bug*. The dataset described in Section 5.1 are used for predictions.

This section presents the results of our experiments . We will vary the results by using *Precision*, *Recall* and *F-Score* as the main measurement. The results outlined the performance of our approach based on different algorithm compared with the improvements of its own by tailoring the original algorithm.

After that, we compare our method with the state of the arts to show some useful points of how our approach can be useful.

5.5.1 The Effect of Minimal Rules Credibility

Description

As elaborated in Chapter 3 under Section 3.2.1, our method highly reliant on varying the measure of *support* counts and *confidence* percentage, to produce set

of credible rules to use for classifying issue reports. The user defines threshold values for each one of these measures which must be met by all rules. The experiments presented in this section study the effect of varying threshold value for given *minSupport* and *minConfidence* values. The best pair of values for the *minSupport* and *minConfidence* thresholds have been experimentally determined.

In order to examine how the different values of *minSupport* affect the prediction accuracy of the generated rules a series of prediction experiments are performed for each dataset described in Section 5.1. To evaluate the performance of our method we used the following variation.

1. Minimum support - Ranging from 3% to 10%

2. Minimum confidence - Ranging from 40% - 100%

As for feature engineering we are using *Term Frequency Inverse Document Frequency* and apply *Chi-square(X^2)* statistical test to select only 150 most significance features found in the date set. The reason being, when mining the rules using apriori algorithm [2, 31, 1], more features will lead more computational time.

Based on the aforementioned variation we ended up having 64 results for each project and 256 in total for all projects in our dataset. Hence, the following section presents the results that illustrate stability of our initial proposed method discussed in Section 3.2.2 and 3.2.3 for minimal credibility. We include the complete experiment results in Appendix section.

In section 5.6, we presents the results using the *minSupport* and *minConfidence* settings in this section for the second method discussed in Section 4.1 and 4.2.

Hold-out Evaluation Results

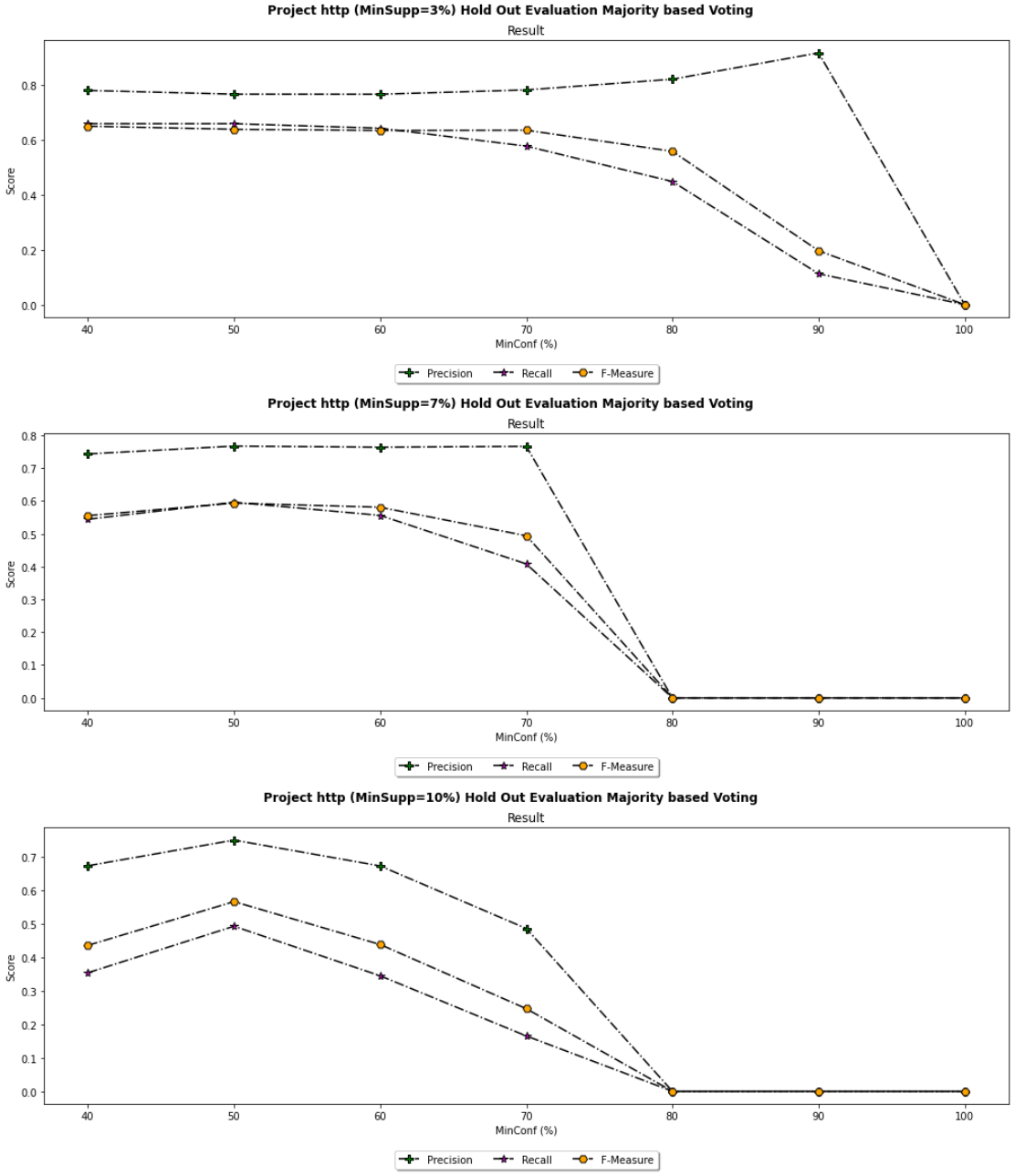

Figure 5.4: Http Project - Hold Out

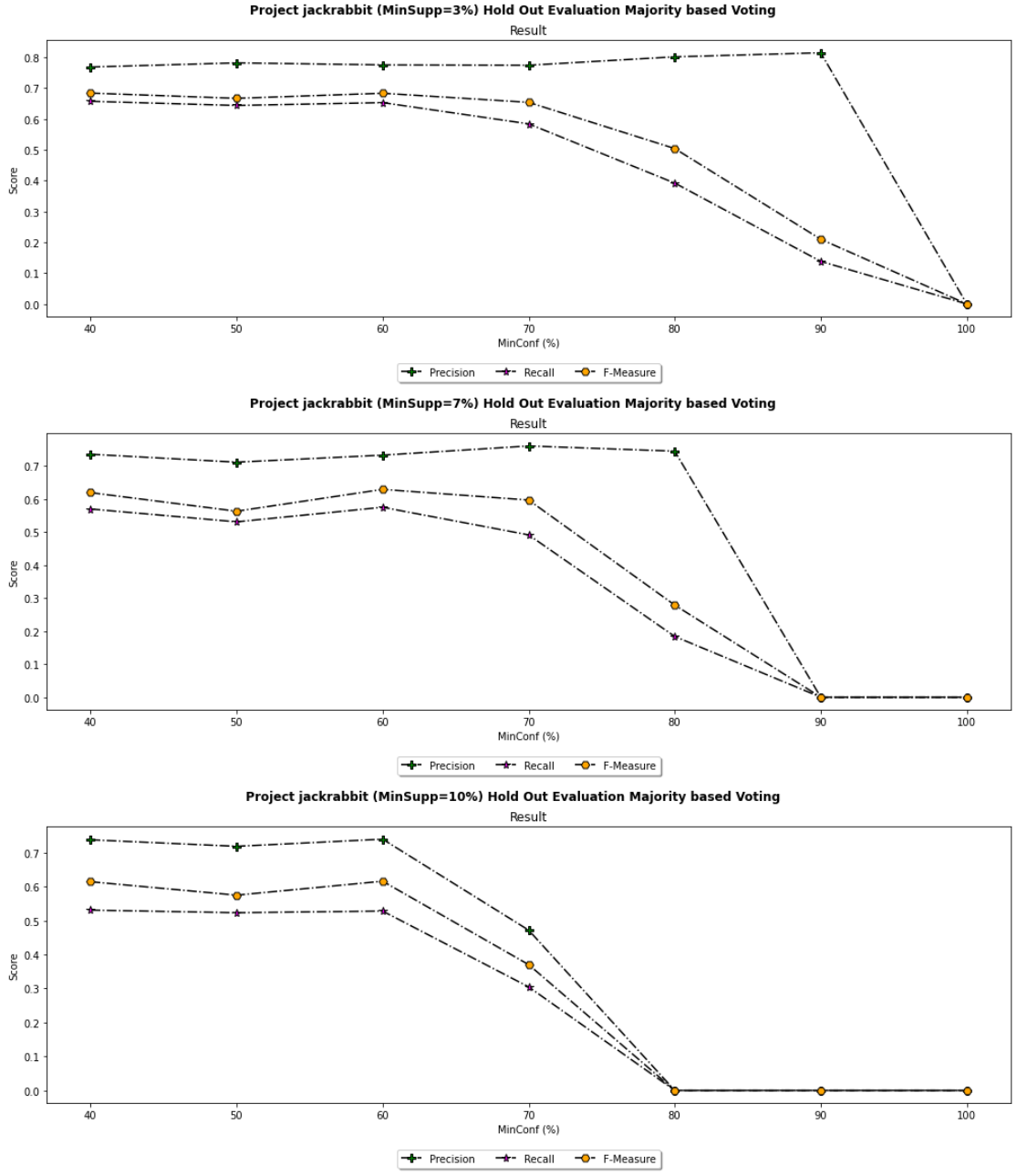

Figure 5.5: Jackrabbit Project - Hold Out

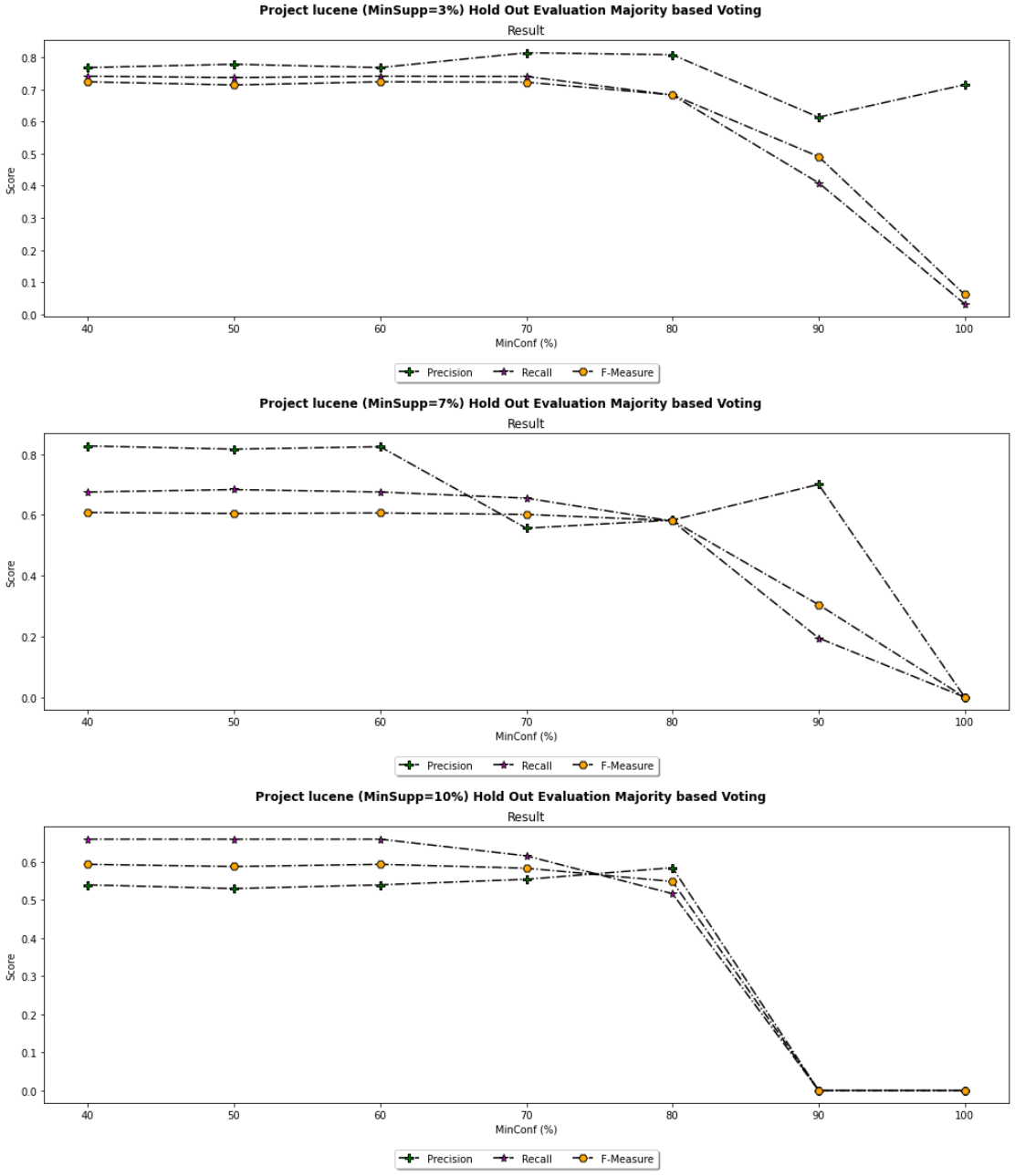

Figure 5.6: Lucene Project - Hold Out

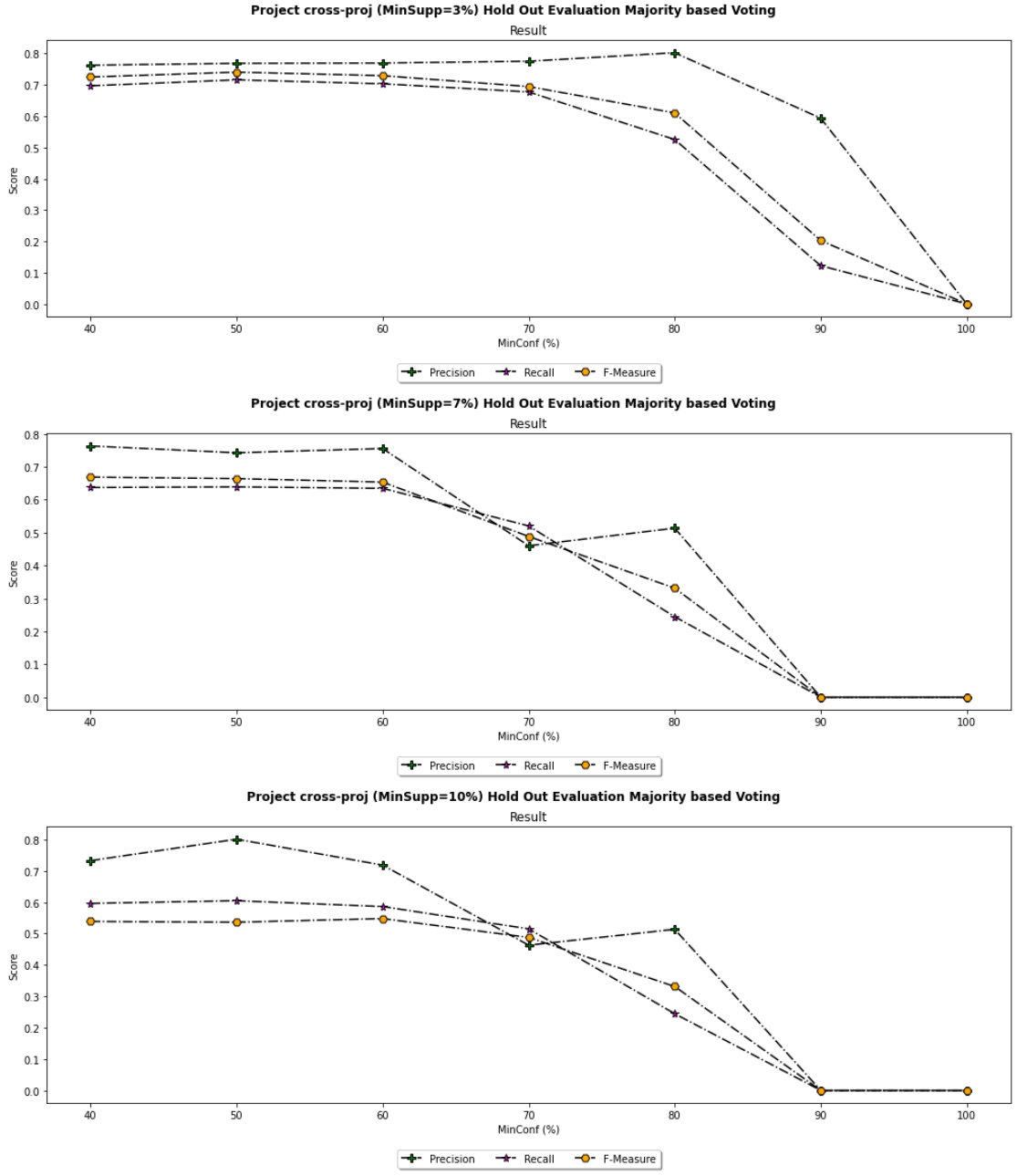

Figure 5.7: Cross Project - Hold Out

Experiment Results 10-Fold Cross Validation

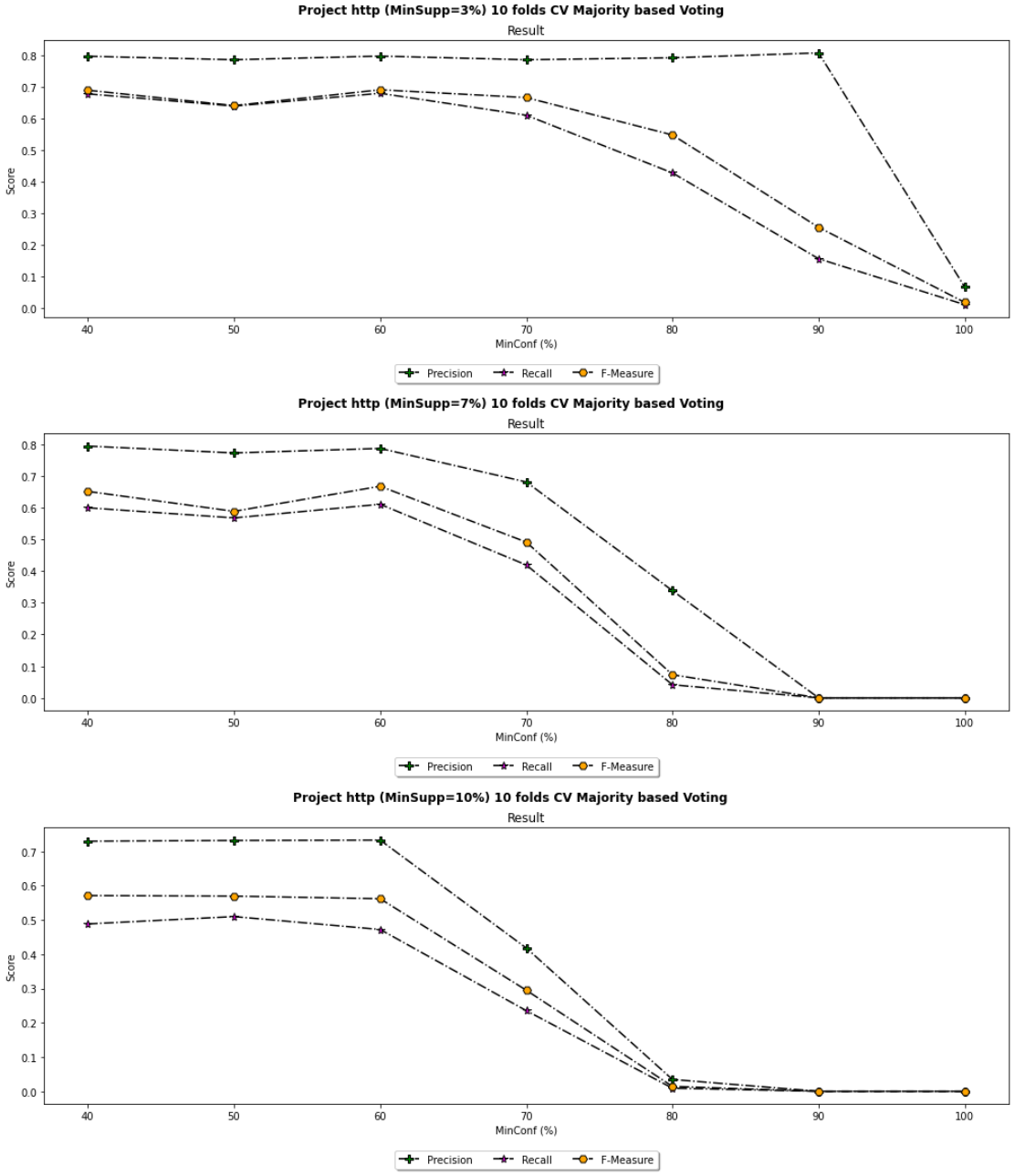

Figure 5.8: Http Project - Cross Validation

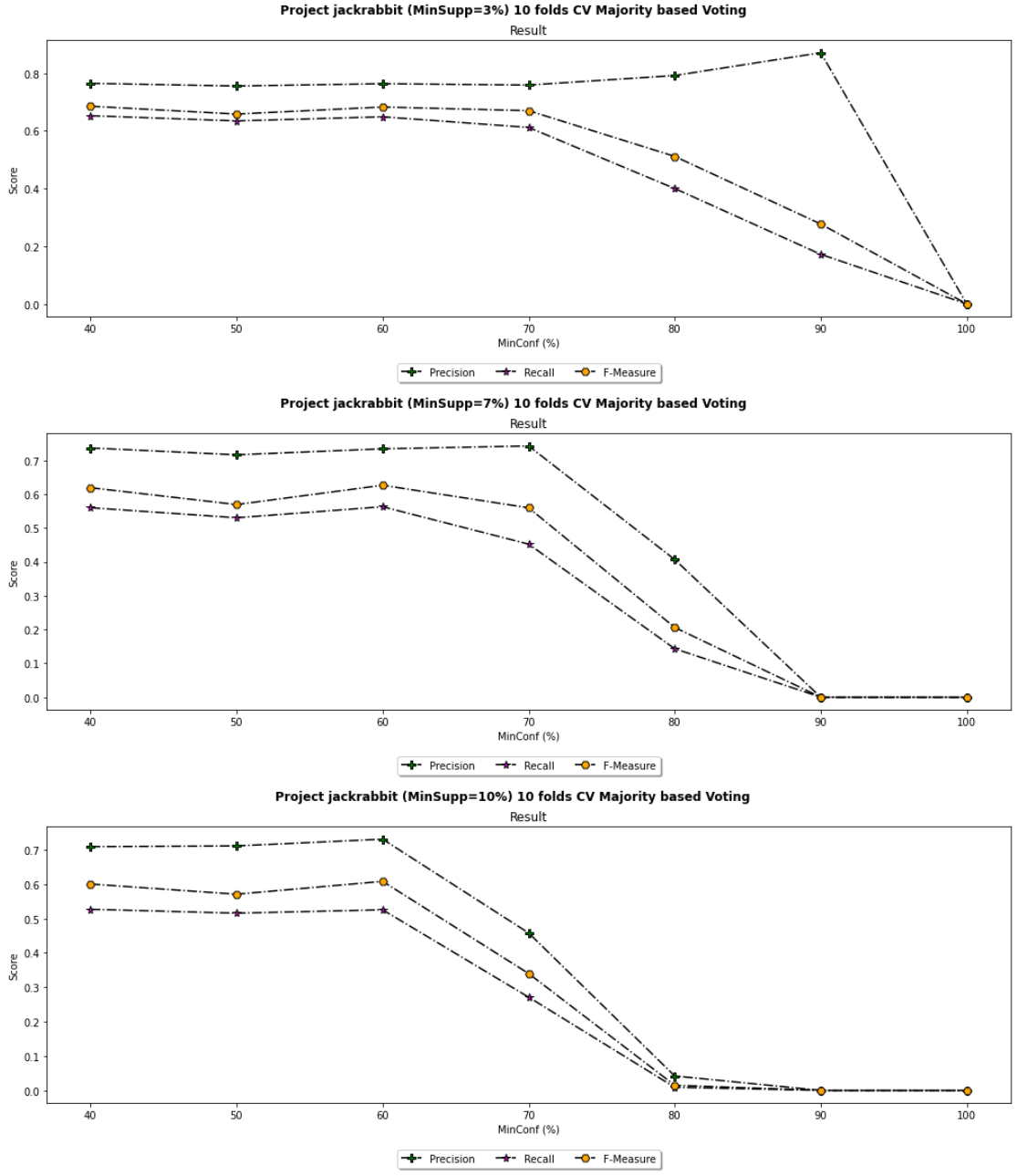

Figure 5.9: Jackrabbit Project - Cross Validation

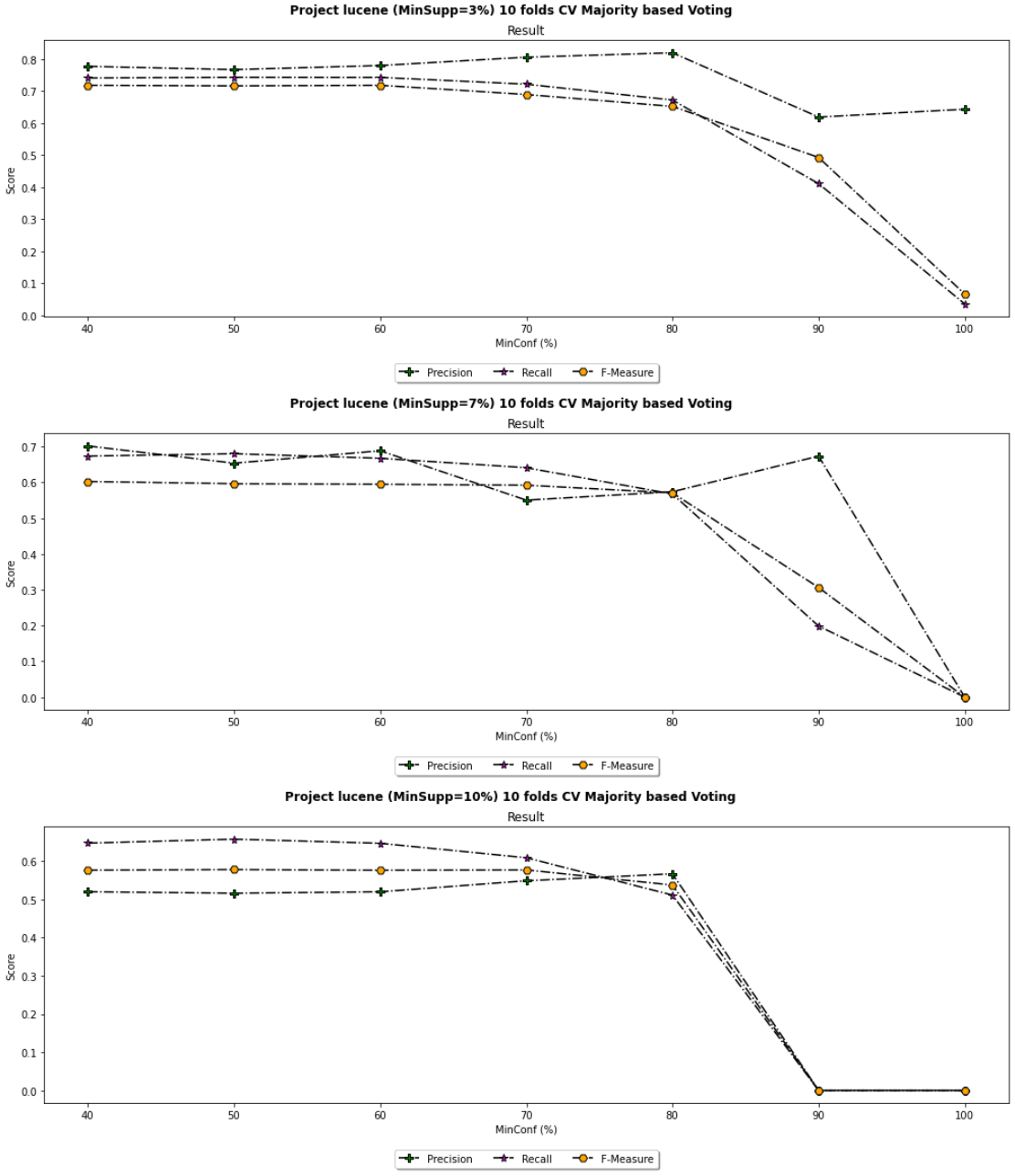

Figure 5.10: Lucene Project - Cross Validation

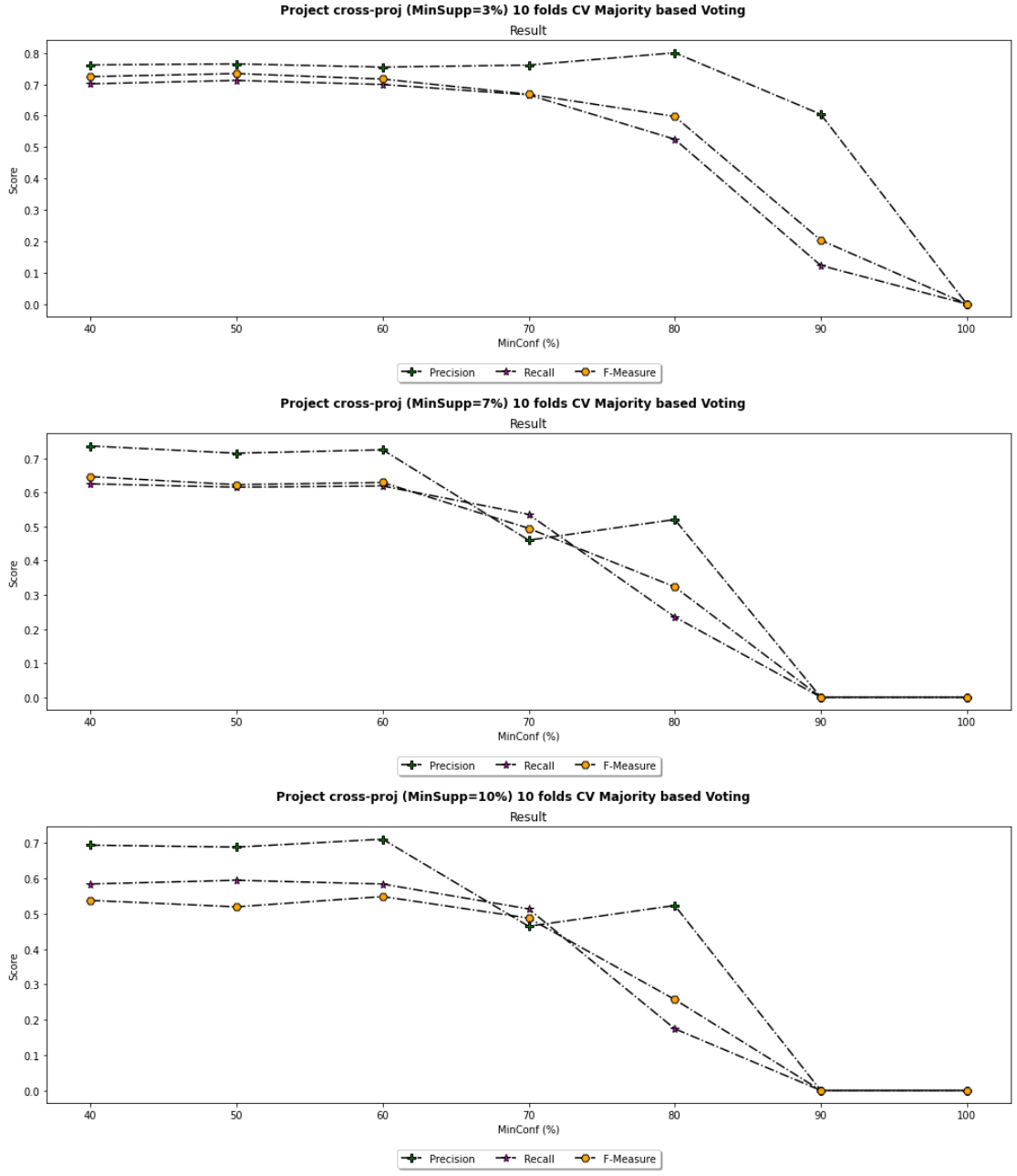

Figure 5.11: Cross Project - Cross Validation

5.5.2 Discussion

As illustrated in Figures 5.4, 5.5, 5.6, and 5.7 illustrate the result based on *Hold Out evaluation* while Figures 5.8, 5.9, 5.10, and 5.11 illustrate result for *Cross*

Validation evaluation. It is apparent for both types of evaluation when increasing the *minConfidence* threshold reduces the number of rules that can be generated. For some data sets this means that the prediction accuracy is reduced due to the reduced number of generated rules that can be used for prediction. We varied *minSupport* to 3% as the lower bound, 7% as the mid bound and 10% as the higher bound to investigate the stability of minimal credibility.

It can be observed that the classifier perform quite stably when the *minSupport* is set to 3% across three projects. The F-Measure rate dropped significantly when the *minConf* enter 80%. This is obvious since more *accurate* rules are sought and less rules will be discovered.

5.6 Classification Experiment - Method 2

This Section presents the results of a series of experiments to evaluate the performance of the developed methods in a classification task by using our enhanced method described in Section 4 and 5.

5.6.1 Minimal Credibility Thresholds

As presented in Section 5.5, we decided with *minSupport* count of 3% as the minimal credibility threshold since this number promised stability in our method in term of *precision*, *recall*, and *fscore*. The experiments are divided into two evaluation known as *Hold Out* and *Cross Validation* following the same strategy applied in Section 5.5. Unlike the preceding section, every rules discovered in this experiment will be treated differently depending on their support value.

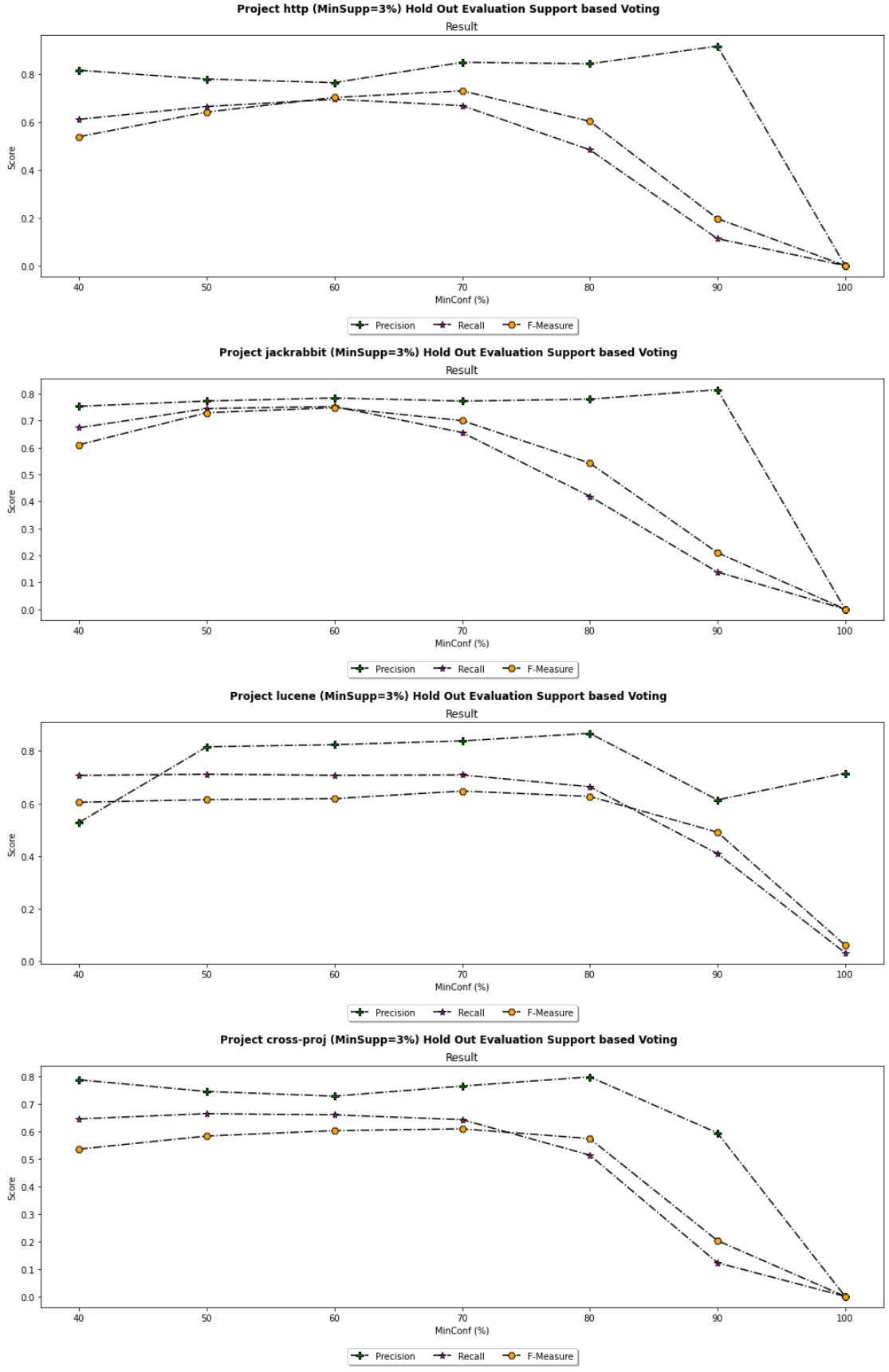

Figure 5.12: Method 2 Hold Out Evaluation

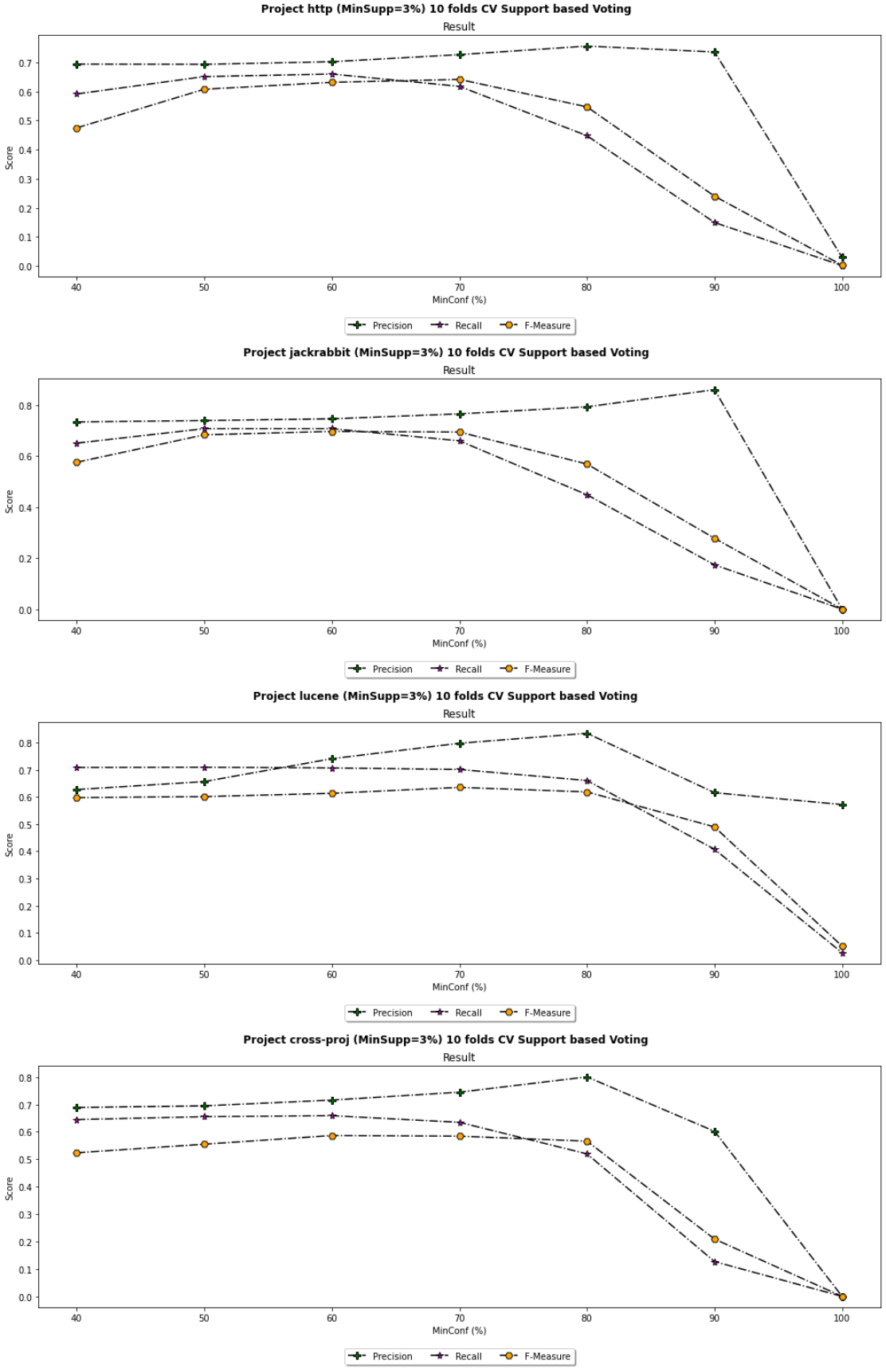

Figure 5.13: Method 2 Cross Validation

5.6.2 Discussion

As can be seen in Figures 5.12, and 5.13, raising the *minConf* threshold results in a reduction in the possible number of rules to be generated. This indicates that the accuracy of the prediction is going to suffer as a result of the decreased number of generated rules that may be used to the classification process.

It is plain to see that the results of Method 2 are not significantly different from those of Method 1. We hypothesise that the occurrence of this phenomena is related to the fact that we count occurrences of "unknown" classification as being erroneous. Because of this, our classifier's performance is substantially impacted due to the fact that when higher thresholds are applied, more correct rules will be sought after, which will result in fewer rules being generated. When that event occurred, none of the rules that are specifically designed to classify reports containing fewer than five vectorised phrases were activated. As a result, the reports are going to be categorised as "unknown".

5.7 Experiment Summary - Method 1 & 2

Concluding our experiments with Methods 1 and 2, we posit that the classifier's performance was notably influenced by adjustments in *minSup* and *minConf* parameters. The efficacy of the classifier in both methodologies was influenced upon the quantity of generated rules and the frequency of "unknown" classifications.

A noteworthy pattern observed across both methods was that elevating the threshold led to an increase in "unknown" classifications and a subsequent reduction in accuracy.

5.8 Comparative Analysis

This section presents the performance of our proposed method with the state of the art solutions. Here, we present different variations that illustrate our result. We introduce two different variations. The first one to include "unknown" as a *correct* classification and to include "unknown" as the majority class in the classification.

The baseline methods for comparison included the studies that have been conducted by Natthakul et al. [62] and Pannavat [77], which rely on predefined features and frequency-based patterns. These were chosen to evaluate the effectiveness of the proposed approach in capturing less frequent but meaningful patterns that traditional methods may overlook.

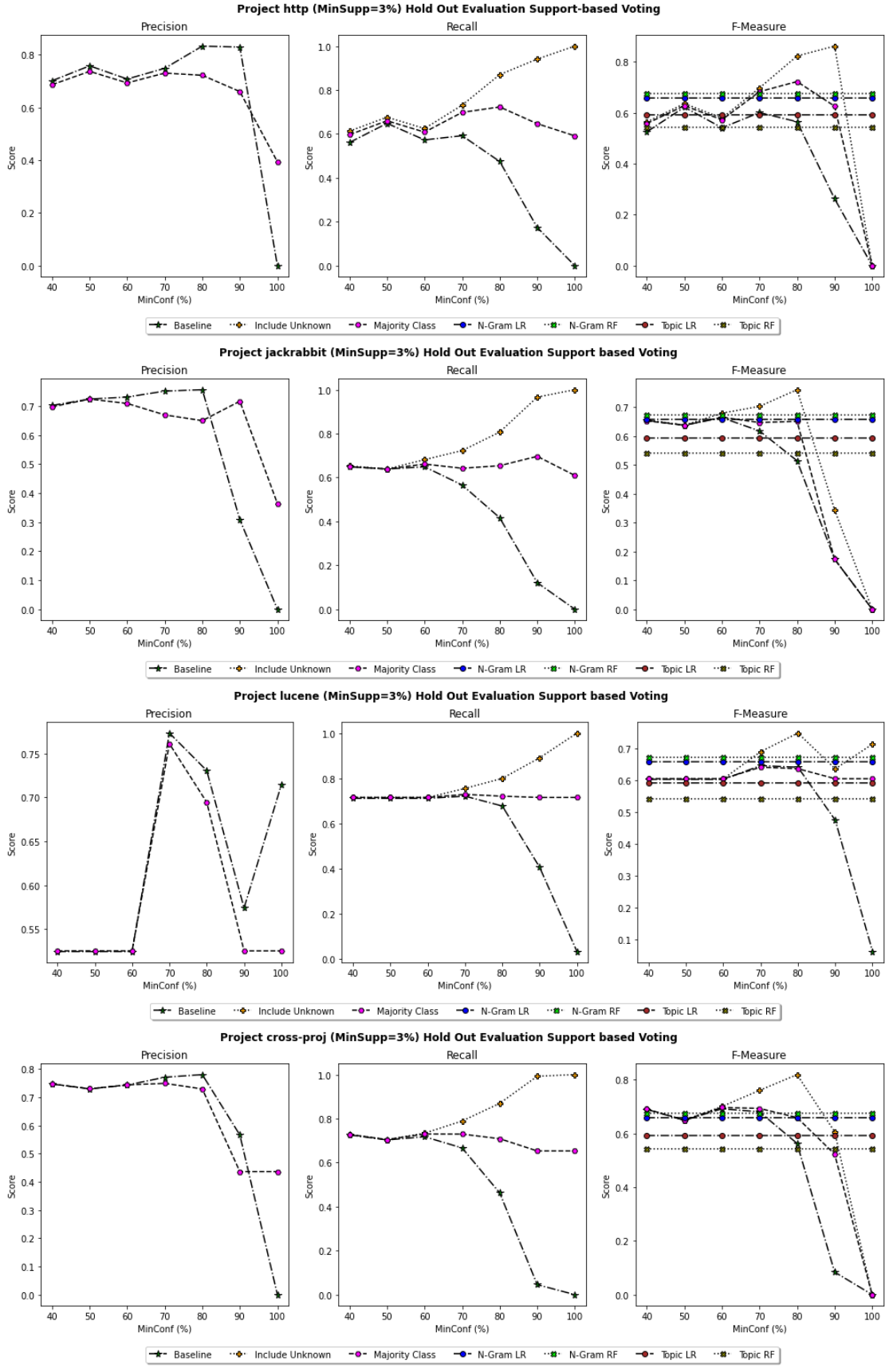

Figure 5.14: Method 2 Hold Out vs Others

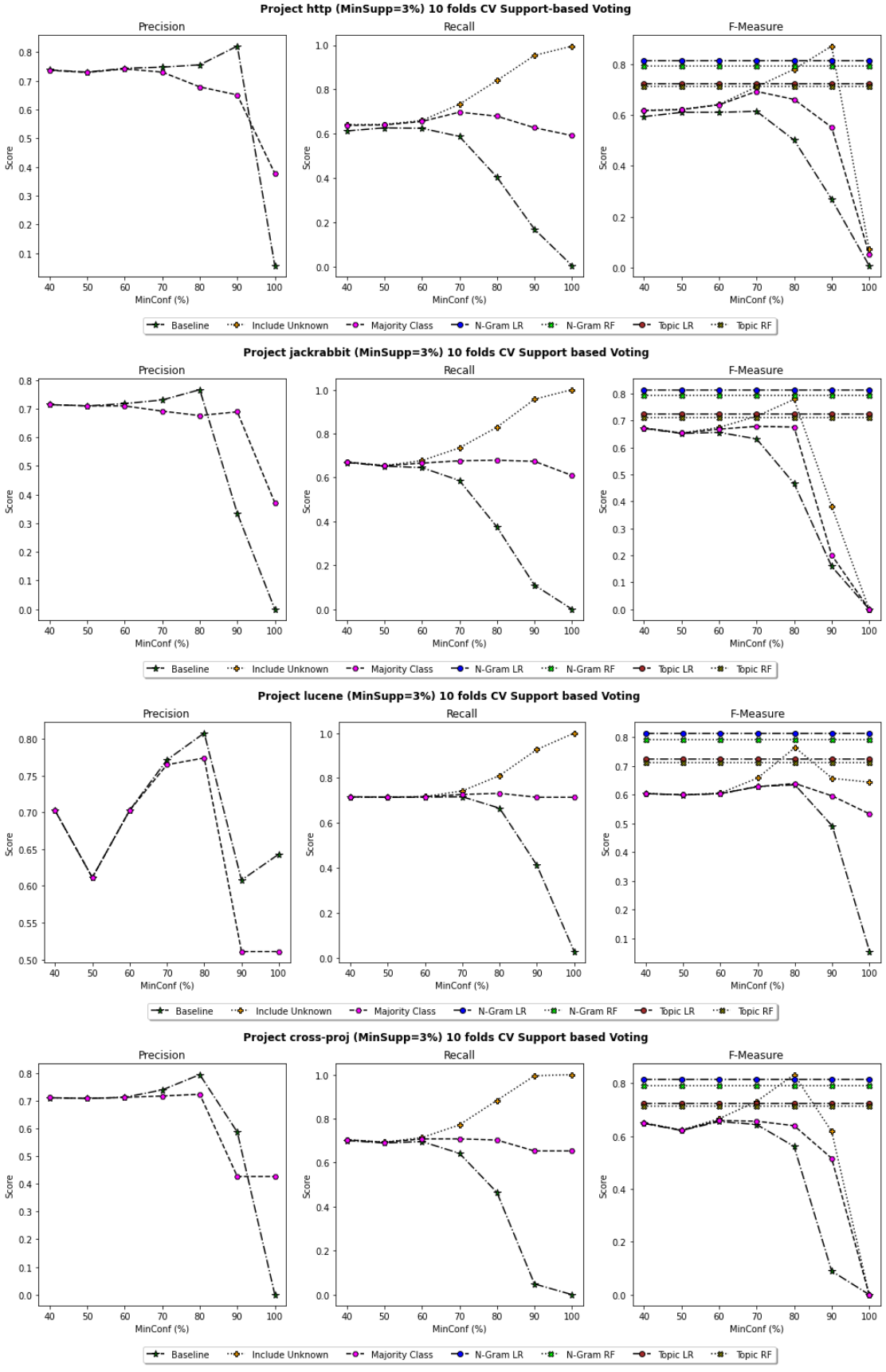

Figure 5.15: Method 2 Cross Validation vs Others

5.8.1 Discussion

The findings depicted in Figures 5.14 and 5.15 exhibit a similar trend as discussed in the preceding section. When the values of *minSupp* and *minConfidence* are increased, a noticeable decrease in the prediction accuracy of our baseline approach is noticed. It is important to acknowledge that in this context, the term "Baseline" pertains to Method 2 as outlined in Section 5. The performance of the state-of-the-art methods remains consistent as they do not depend on adjusting the values of *minSupp* and *minConfidence* during the classification of issue reports.

Irrespective of whether Cross Validation or Hold-Out evaluation was employed, our classifier consistently demonstrated a comparable level of F-Score, ranging from 60% to 70%, across all projects. When the minimum support is set to 3% and the minimum confidence is set at 60%. The thresholds presented in this section are derived from the experimental results obtained in Section 5.5.

A noteworthy pattern becomes evident when reports labelled as "unknown" are categorised as correctly classified, since it demonstrates a substantial level of accuracy across three projects. In contrast, the performance of the F-Score metric experienced a notable decline when reports labelled as "unknown" were regarded as inaccurate classifications.

In summary, our technique exhibited limited prediction accuracy due to our treatment of "unknown" cases as erroneous, hence impacting the performance of our classifier. We argue that reports classified as "unknown" should be approached separately due to the presence of uncertainty, necessitating more clarification on how to address them.

5.9 Unable to classify

Our classification method, as previously stated, is based on majority votes. In this case, we may run into an issue voted as a "tie". That is, when the number of rules fired is equal, we may have the same number of votes in both categories (`bug` and `non-bug`). In Section 5.9.1, we report the total number of reports classified as "unknown" based on experiment using Method 1 and 2 discussed in Chapter 3 and 4 respectively. The reports are presented based on individual projects by comparing both Method and how many "unknown" reports discovered as we increase the *minSupport* and *minConfidence* thresholds. In section 5.9.3, we present the possible causes of an unknown classification by extracting some of the reports that have been classified as "unknwon" using our method. We studied the characteristics of the reports and present the findings accordingly.

5.9.1 Unknown Classification

This section reports the number of "unknown" classification discovered in our experiment using Method 1 which was elaborated in Chapter 3. The data presented in the table were extracted based on *Hold-Out* evaluation where we are training 90% of the data and left the remaining 10% for testing purposes for Http Client Project only.

		Minimum Support							
		3%	4%	5%	6%	7%	8%	9%	10%
Minumum Confidence	40%	1	2	3	3	6	9	9	14
	50%	2	3	4	2	5	4	5	11
	60%	1	1	2	3	4	9	15	15
	70%	3	5	9	10	15	18	20	28
	80%	7	10	15	17	30	40	42	44

Table 5.11: Http-Client "Unknown" Classification - Method 1

Discussion

It is evident from the tables presented that as we increased the *minSupport* and *minConfidence* thresholds, more reports will be classified as "unknown". This is because the number of rules fired, will be decreasing as higher demand of support and confidence are required.

Please note that ratio of "unknown" report in project individually are varied since the *training* and *testing* proportions are different. For example, for Http-Client project the total number of issue report are 741. Whereas for Lucene and Jackrabbit, both are having 2443 and 2402 respectively.

For Http-Client using Method 1, we observed that the number of "unknown" reports fairly stable when we set the *minConfidence* to 60%. We believe when 60% is used as *minConfidence*, the method still can produce quite number of rules of classification task. And by setting the threshold at this value we are deriving set of credible rules by maintaining the their *strength* and *accuracy* level.

5.9.2 Possible Causes of Unknown

These possible causes are only our initial conjecture and further analysis is required to have more concrete evidence.

1. `Short Report`:

   Definition: any vectorised report that is less than 3 i.e $\mathbb{R} = \langle t_1, t_2, t_3 \rangle$, with all the stop words have been eliminated. These vectorised reports will only contain all the useful keywords to be used for classification. Initially, our conjecture said that short report is one of the cause of unknown when no rules are fired and ended up with *zero-zero* phenomenon where zero rules counted for bud and also for non-bug. Below, we manage to extract two reports that are deemed as *short report*.

Table 5.12: Short Report

Item	Content
Project	Lucene
Issue ID	Lucene-238
Category	Non Bug
Title	[PATCH] import cleanup
Description	This patch just removes useless imports so you get less warnings in Eclipse.

Item	Content
Project	Lucene
Issue ID	Lucene-240
Category	Non Bug
Title	bug form doesn't list latest version
Description	

As be be seen from the given reports, the amount of useful keywords that can be extracted are quite limited. When these report went through pre-processing stage as described in Section 5.2.1, these two reports were ended with few keywords and might not be able to be classified with generated rules the *minSupport* and *minConfidence* are increased.

2. `Missing keywords`:

   Definition: any report contained useful keywords but cannot be classified since no rules generated contain such keywords. We assumed some of the reports might be ended up being classified as *unknown* but do contain useful keywords which would be very helpful to classify them into specific category. For example, let us assume the following the vectorised report below which has been classified as *unknown*.

   $V_1 : \langle$ *appear, bottom, check, code, comment, complet, document, fals, give, go, happen, javadoc, like, list, mention, not, often, otherwis, page, print, read, rest, without, nb* $\rangle$

from the vectorised report, let us assume the following rules were fired during classification phase.

r_1: $check \rightarrow bug$

r_2: $code \rightarrow bug$

r_3: $not \rightarrow bug$

r_4: $read \rightarrow bug$

r_5: $code, not \rightarrow bug$

r_6: $not, read \rightarrow bug$

r_7: $code \rightarrow non\text{-}bug$

r_8: $document \rightarrow non\text{-}bug$

r_9: $like \rightarrow non\text{-}bug$

r_{10}: $not \rightarrow non\text{-}bug$

r_{11}: $code, not \rightarrow non\text{-}bug$

r_{12}: $not, like \rightarrow non\text{-}bug$

based on the rules listed, there are six rules classify the report as *bug* and another six rules identify the report as *non-bug*. Having analysed the vectorised report and rules fired, there are some meaningful keywords (*mostly verbs*) found in the vectorised report that could be useful for classification. For example keywords like *comment, print, list, mention, read, and false*. These keyword may appear below the threshold of *minSupp* and *minConf* which ended up not counted as rules. This is how the situation of missing important keywords in the rules but appeared in the unseen dataset.

3. `Contain code snippet etc`

   Definition: Report contained non-textual information i.e code snippet, stack trace, url, directory etc. These elements will affect the classifier as will also be pre-processed and treated as vectorised reports. In our initital algorithm, we treated code snippet as normal text. We transform any code snippet found in the report by using camelCase split. Any words in the form of

camelCase found in the report will split accordingly. However, this did not help to produce good or relevant features to generate rules. Since normal text and code snippet should not be treated equally. Figure 5.16 depicts an issue report that heavily contain code snippet.

Figure 5.16: Issue Report mostly with Code Snippet

4. `Multi-issue`:

Definition: One single report that contained multiple issues addressed by the reporter. To show how this would actually happen, in Table 5.13 below, we give a report that was classified as unknown by our method. Here we can see that the report was actually addressing two issues. The first half, highlighted in red, outlined a problem that the reporter has encountered, and therefore the report could potentially be classified as a `bug` report. In the second half, highlighted in blue, the reporter mainly offered a suggestion, so the report could be considered as a `non-bug` one. Lastly, text highlighted in olive suggests that the reporter gave an opinion whihc also can be considered as a `non-bug` category. Looking into the rules we generated, the presence of keywords such as `problem`, `warning` and `error` typically lead to `bug` category, and the keywords such as `change`, `option` and `would` in the second half typically lead to `non-bug`, hence a tie in the end. Obviously in this given example it may reasonable for the report to be classified as a bug report overall, and the suggestion contained in the second half would simply be useful by the software engineer who addresses the issue as a possible solution. But in other cases, it may be desirable to leave the report unclassified and leave its category to a human decision.

Table 5.13: Multiple Issues in one Report

Item	Content
Project	Jackrabbit
Issue ID	JCR-9
Category	Bug
Title	Version.isSame(Object) not working
Description	Version interface is implemented (on the frontend) by the VersionImpl class (extending NodeWrapper), which delegates to an internal NodeImpl class, which in turn extends ItemImpl. Say you have : Node node = // at Version 1.0 Version version = // retrieved as 1.0 for the node Version baseVersion = node.getBaseVersion() You now expect baseVersion.isSame(version) even if baseVersion != version This fails, because VersionImpl delegates the isSame call to its delegatee, thus above call becomes ((VersionImpl) baseVersion).delegatee.isSame(version) where this method is implemented by the ItemImpl class from which the delegatee NodeImpl extends. That latter implementation ItemImpl.isSame() only returns true if the other is an ItemImpl, too. But this is not the case because VersionImpl is a Version, NodeWrapper, Node but not an ItemImpl. Probably the best solution would be for NodeImpl.isSame() to check whether the otherItem is a NodeWrapper and use ((NodeWrapper) otherItem).delegatee as the otherItem for the delegatee call. On another track: ItemImpl.isSame() should probably do a fast check whether the otherItem is actually the same instance to prevent type checks...

Item	Content
Project	Jackrabbit
Issue ID	JCR-14
Category	Bug
Title	{XML—Object}PersistenceManager.destroy(*) may fail
Description	The destroy methods of the ObjectPersistenceManager class try to delete their files without checking for their existence. This may result in a FileSystemException being thrown because according to the specification of FileSystem.deleteFile() a FileSystemException is thrown "if this path does not denote a file or if another error occurs." While the Jackrabbit LocalFileSystem implementation silently ignores a request to delete a non-existing file, our internal implementation of the interface throws a FileSystemException in this case, which cause destroy to fail. I suggest all destroy methods should be extended to first check for the existence of the file to prevent from being thrown. Note: This not only applies to ObjectPersistenceManager but also to XMLPersistenceManager.

5. `Overlapping Rules`

   Definition:

   (a) During classification phase, most of the rules fired are overlapping ended up the majority vote to be a tie. This is because when weak rules but not credible are used for classification.

   (b) i.e $R1 : \langle t_1, c, supp \rangle$ where t_1 is the term exist in the rule set. C represent the class value and supp is the occurrence of such rules in the entire training dataset.

   (c) $R1 : \langle error, b, 30 \rangle$ and $R2 : \langle error, nb, 4 \rangle$ where $R1$ has stronger *support* which is 30 as compared to $R2$ which only has 4 as its *support*. Given a vectorised issue report as follow $V1 : \langle error, null, ? \rangle$. If this is happening both $R1$ and $R2$ will be fired and will end up as a tie although $R2$ is considered as weak rule and possibly not useful for classification.

It is revealed that 15% of the "unknown" class labels were attributed to short vectors with fewer than three tokens, while 25% were due to reports missing key

classification terms. Code snippets accounted for 30% of the 'unknown' classifications, as they lacked sufficient linguistic context for accurate categorisation.

5.9.3 Advantages and Disadvantages

In this section we argue the advantages and disadvantages that an "unknown" classification can contribute to the body of knowledge.

Advantages

Incorporating "unknown" as one of the category in binary or multi-class classification will open a new avenue in software issue reports classification. Below are some of the benefits.

Handling Ambiguity: Some issues may not clearly be a bug or a non-bug based on the provided information. An "unknown" class allows for the representation of such ambiguous cases, preventing misclassification.

Improving Model Accuracy: Forcing ambiguous or unclear issues into "bug" or "non-bug" categories could lead to inaccurate classifications and, consequently, a less effective model. An "unknown" class can act as a safety net, improving overall model accuracy. It avoids the model from making a definitive conclusion on issues where there is not enough information or context, reducing the risks associated with acting on incorrect classifications.

Enhanced Prioritisation and Triage: Issues classified as "unknown" can be flagged for manual review, allowing human experts to decide whether it's a bug or not, ensuring more accurate categorization and appropriate prioritization and assignment.

Training Data Quality: The "unknown" class can be beneficial when dealing with noisy or incomplete training data. It helps to segregate instances where the model has low confidence due to inadequate or conflicting information, thus maintaining the integrity of the training data.

Enhancing Learning: An "unknown" class can encourage further investigation and learning. Insights derived from reviewing "unknown" cases could contribute to refining and improving the classification model over time.

User Trust and Model Interpretability: The provision of an "unknown" classification can enhance user trust as it transparently reflects the model's uncertainty, rather than forcing a possibly incorrect binary classification.

In conclusion, an "unknown" class in software issue classification can act as a mechanism to manage uncertainty and ambiguity, improving the robustness and reliability of the classification system.

Disadvantages

This section presents the disadvatages of incorporating "unknwown" as the additional class in *binary* or *multi-class* classification.

Increased Complexity: Adding an additional class can complicate the model, making it more challenging to develop, maintain, and interpret. The added complexity might require more sophisticated methods to accurately classify instances.

Resource Allocation: Issues classified as "unknown" will likely need manual review and resolution, which can be resource-intensive and may slow down the overall resolution process.

Inconclusive Results: Having a considerable number of issues falling into the "unknown" category may lead to inconclusive or non-actionable insights, which might not be helpful for stakeholders looking for clear answers.

Imprecision: The "unknown" class can sometimes become a catch-all category for instances that are difficult to classify, leading to less precision in the model's outputs.

Training Challenges: Training a model with an "unknown" class can be challenging, as it may require labeled examples of "unknown" instances, which might not be readily available.

Potential for Neglect: There is a risk that issues categorized as "unknown" may be overlooked or deprioritized, potentially leading to significant issues being left unaddressed.

User Frustration: Users might find an "unknown" classification unhelpful or frustrating, as it does not provide a clear resolution or direction, and may prolong the time to reach a conclusive decision.

Dilution of Focus: The model might end up allocating substantial focus and resources on resolving the ambiguity in "unknown" classes, potentially detracting from the primary goal of accurately classifying clear-cut instances.

Delay in Remediation: The introduction of an "unknown" class might slow down the process of addressing and remedying legitimate bugs as they are stuck in limbo awaiting classification.

Model Performance Metrics: Introduction of an "unknown" class could complicate the evaluation of model performance, as traditional binary classification metrics may no longer be straightforwardly applicable.

Balancing the advantages and disadvantages is crucial. The introduction of an

"unknown" class can be beneficial in managing uncertainty but requires careful consideration and implementation to avoid the potential downsides.

5.10 Summary

This chapter introduced the data preparation stage, during which the data set was pre-processed before to conducting the experiments. Extensive experiments have been carried out to assess the efficacy of the developed algorithm for a classification task, utilising benchmark datasets sourced from three distinct open source software projects.

In the context of classification, the algorithm presented in this study demonstrated favourable outcomes that are equivalent to the approach put out by other researchers. Additionally, it exhibited slightly better results on specific datasets.

Furthermore, we engaged in a comprehensive discussion over the identification of a "unknown" phenomenon, attributing it to the inherent characteristics of our methodology. This was substantiated by presenting illustrative cases of issue reports extracted from the dataset.

Chapter 6

Conclusion & Future Works

6.1 Conclusion

Classifying software issue reports can present challenges, particularly when dealing with reports that include code snippets and are written in lengthy sentences. The process of manual classification often results in a significant expenditure of time and laborious. The overall contribution of the thesis has been centred around a methodology for classifying issues reported in the context of software maintenance. Our method was driven by the methodology of Classification Association Rule Mining. Instead of seeking dominant patterns in issue reports for the purpose of constructing a classifier, our approach involves identifying all credible patterns in order to develop a set of collectively robust rules for classifying issue reports. This practise is valuable and significant in situations where there are several semantic interpretations linked to particular phrases or when an issue report encompasses multiple problems. The experimental results demonstrate that our proposed methodology exhibits a promising level of accuracy in classifying issue reports obtained from diverse projects. The contribution of the thesis can be summarised as follow:

- **Majority Vote Classification**:

  We introduced a classification technique was implemented using the majority vote approach. Chapter 3 was dedicated to elaborate our initial approach in classifying software issue reports. The original hypothesis was based on the generation of "credible" rules through the manipulation of the *minSupport* and *minConfidence*.

  By lowering the threshold values, a substantial increase in the number of rules available for classification was seen. While many rules may be considered weak, they are nonetheless utilised jointly in constructing a proficient classifier as long as they occur with sufficient frequency in our complete dataset.

  The procedure will afterwards determine the number of rules that pertain to bugs and non-bugs. The class value for the report classification will be determined by selecting the rules that have the highest count representing a specific class. The methodology employed in this approach diverges significantly from the existing classifier models that were constructed using a "dominant" pattern. By employing such a methodology, certain rules that are deemed "credible" may be omitted and not utilised for the purpose of classification.

- **Support based Voting**:

  We conducted additional investigations into the potential for improving our technique by using support count as a component of the selection criterion for rule counting during classification. In Chapter 4, a comprehensive description is provided on improving our methodology, which is achieved through Algorithms 4 and 5. In contrast to our previous methods, which viewed each rule as equal regardless of its support count, the Support-based Voting strategy assigns a value to each rules based on its support. It utilises these values combined to classify an unseen issue report. This strat-

egy presents a more equitable voting technique by assigning varying levels of weight to each rule based on their respective "strength" or "support" count.

- **"Unknown" Classification** The methodology employed in our study is based on calculating the number of rules that have identified a particular report as either a bug or a non-bug. If there is a situation where both rules receive an equal number of votes, our method will classify the report as "unknown". Chapter 5, specifically Section 5.9, provides a comprehensive analysis of the classification finding pertaining to entities labelled as "unknown".

  We presented an analysis of the major characteristics of a report that is likely to be classified as "unknown". We used samples from issue reports within our dataset to support our findings. Based on our discussion, using the "unknown" classification can introduce a novel approach in the field of classification, particularly in scenarios where reports exhibit a significant degree of uncertainty. The reports often encompassed multiple issues addressed inside a single report or were heavily intertwined with code snippets, making pre-processing challenging. The "Unknown" classification differs from typical classifiers in that it does not require reports to be classed as a definitive class value. This will impact the overall precision of the model.

  Nevertheless, if the inclusion of unknown variables is to be integrated into the measurement process, it is imperative to be executed suitably. The excessive classification of reports as "unknown" does not serve as a reliable performance measure for our methodology. Therefore, in Section 6.2.1, we present guidelines on how to address the issue of "unknown" classification in order to preserve the efficacy of our approach.

6.2 Future Works

In Section 6.2.1 we present potential solutions to overcome with "unknown" classification should the number of reports classified as "unknown" are relatively high. Please note that we do not view "unknown" as an issue or new problem to solve. However, we view it as an opportunity to improve the performance of our method. We divide our potential solutions into two main ideas. One in which require modification of our existing method. Secondly, to include human in the loop as expert view when dealing with certain types of "unknown" report based on the findings presented in Section 5.9.3.

In Section 6.2.2 we provide some pointers on new research directions based on our proposed method.

6.2.1 Possible solutions to Unknown Cases

As stated earlier, an unforeseen outcome of our methodology was observed when the total number of votes reached a state of equality that leads to "unknown" classification. Upon thoroughly analysing the issue at hand, we propose a range of potential solutions, considering the possible causes outlined in Section 5.9.3. Solution 1 and 2 below require modification of our method. Whereas Solution 3 requires human expert to classify the "unknown" report manually and update the classifier for future classification.

Solution #1: Missing Keywords & Short Reports

Missing keywords occurred in our interpretation when a report contained desirable keywords but was unable to be classified due to the absence of rules fired during the classification phase. To anticipate and handle such an event, it is recommended to implement a reclassification procedure. This approach is inspired

by the semantic approach to find any existing rules that are comparable to the keywords in the unknown report by way of calculating the cosine similarity between each keyword and the set of all possible rules. The procedure is presented below.

1. Scan the vectorised report find any keywords that have no match with rules fired during classification.
   (a) Find any keywords that fall under the following category *noun*, *verb* and *adjective*.
2. The identified keyword will then be categorised into verb taxonomy as designed in WordNet.
3. Scan all rules that fall under the same categorization group of identified keywords.
4. Match all keywords discovered with rules that have been grouped accordingly.
   (a) For each keyword find the match rules in the same category.
   (b) Calculate the cosine similarity between keyword and rule.
   (c) repeat until all keywords and rules have been covered and calculated.
5. Rank all the matching keywords and rules and find the highest value.
   (a) Select the rules that have closest to the keyword which indicate strong semantic similarity.
   (b) Use the rules to reclassify the Unknown report by adding the vote counting of the initial classification.
6. Present reclassification result.

Solution #2: Report Contained Code Snippet

One of the complications associated with the classification of issue reports is the presence of extensive code snippets within some of the reports. The presence of extraneous symbols frequently observed in a code snippet poses a hurdle during the pre-processing phase. Previously, we employed the camelCase split method to handle a code snippet. Nevertheless, this methodology ultimately resulted in the extraction of numerous specialised terminologies present inside the code snippet. Therefore, in order to facilitate future enhancements, it is suggested that every word written in camelCase style be replaced with the term *Code'* to signify that the report contains a code snippet.

By implementing this approach, the frequency of specialised terminology present in the code snippets is diminished and substituted with the phrase *Code'*. This development warrants additional exploration in the field of issue report classification, specifically in the incorporation of code snippets as a feature selection technique.

Based on our empirical observations, it has been noted that a significant proportion of reports including code snippets are predominantly classified as instances of software bugs. By employing this methodology, it is postulated that there will be an improvement in the efficacy of the classification process.

Solution #3: Multi Issues Report

One of the findings from our investigation of the attributes of reports categorised as "unknown" is that a singular report encompassed numerous topics being examined. Based on the example provided in Table 5.13, it is evident that the majority of reports that address many issues extensively employ lengthy sentence structures. Therefore, the identical number of votes across *bug* and *non-bug* instances presented a challenge to our classification algorithm.

To enhance future enhancements, it is advisable to refer to any reports that surpass a specific number of vectorised *terms* and have been classified as "unknown" for manual assessment and clarification. Once the human expert has verified that a report has multiple issues, it is necessary to update the classifier accordingly.

6.2.2 New Research Directions

Further research endeavours can encompass exploring the potential applicability of the findings presented in this thesis across many domains, as well as soliciting feedback from practical implementations.

- **Multi-Class Classification**:

  One potential avenue for investigating the feasibility of our approach is by doing a multi-class classification task. At present, we only employ the method for *binary* classification, which presents a constraint. In order to effectively apply the methodology for multi-class classification, it is advisable to select a substantial dataset, as the efficacy of our technique relies on the "credibility" of the rules derived via the process of rule mining.The presence of imbalanced data could cause a difficulty in effectively implementing our approach.

- **Feature Selection Technique**:

  Alternatively, this is also one potential avenue for exploration. Currently, we only employ *Term Frequency - Inverse Document Frequency* and Chi-square(X^2) as our feature selection. Although these two techniques have proven to be effective [67], it is recommended in explore the possibility of other techniques such as word embedding [55, 22, 81] to improve the feature selection in text classification.

- **Incorporation of Language Model**:

  One promising avenue for future research involves exploring the utility of

modern language models (e.g., BERT, GPT) for feature engineering [23] and automatic labelling [10, 17] tasks in software issue reports. These models excel at capturing contextual and semantic relationships within text, which could enhance feature extraction processes by identifying domain-specific terms and subtle linguistic patterns. Additionally, language models could assist in automating the labelling process by providing preliminary classifications or suggesting likely categories based on report content. This could significantly reduce the reliance on manual intervention and further streamline the issue classification process. Examining how language models perform in conjunction with the proposed approach could offer insights into the scalability and adaptability of intelligent bug tracking systems.

References

[1] Firas Abuzaid, Peter Kraft, Sahaana Suri, Edward Gan, Eric Xu, Atul Shenoy, Asvin Ananthanarayan, John Sheu, Erik Meijer, Xi Wu, Jeff Naughton, Peter Bailis, and Matei Zaharia. Diff: A relational interface for large-scale data explanation. *Proc. VLDB Endow.*, 12(4):419–432, dec 2018.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[3] Ibrahim Aljarah, Shadi Banitaan, Sameer Abufardeh, Wei Jin, and Saeed Salem. Selecting discriminating terms for bug assignment: A formal analysis. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 12:1–12:7, New York, NY, USA, 2011. ACM.

[4] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA, 2008. ACM.

[5] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[6] Alain April and Alain Abran. *Software maintenance management: evaluation and continuous improvement*, volume 67. John Wiley & Sons, 2012.

[7] Sylvain Arlot and Alain Celisse. A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4(none):40 – 79, 2010.

[8] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[9] G. Bavota. Mining unstructured data in software repositories: Current and future trends. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 1–12, March 2016.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

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

[12] Yguaratã Cerqueira Cavalcanti, Paulo Anselmo da Mota Silveira Neto, Daniel Lucrédio, Tassio Vale, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. The bug report duplication problem: an exploratory study. *Software Quality Journal*, 21(1):39–66, Mar 2013.

[13] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[14] Indu Chawla and Sandeep K. Singh. An automated approach for bug categorization using fuzzy logic. In *Proceedings of the 8th India Software Engineering Conference*, ISEC '15, pages 90–99, New York, NY, USA, 2015. ACM.

[15] J. Cleland-Huang, R. Settimi, X. Zou, and P. Solc. The detection and classification of non-functional requirements with application to early aspects. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 39–48, Sept 2006.

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

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[18] Davide Falessi, Bill Kidwell, Jane Huffman Hayes, and Forrest Shull. On failure classification: The impact of "getting it wrong". In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 512–515, New York, NY, USA, 2014. ACM.

[19] Q. Fan, Y. Yu, G. Yin, T. Wang, and H. Wang. Where is the road for issue reports classification based on text mining? In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 121–130, Nov 2017.

[20] IEEE for Software Engineering. Ieee standard for software maintenance. *IEEE Std 1219-1998*, pages i–, 1998.

[21] George Forman. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.*, 3(null):1289–1305, March 2003.

[22] J. Guo, J. Cheng, and J. Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 3–14, May 2017.

[23] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8342–8360, Online, July 2020. Association for Computational Linguistics.

[24] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3(null):1157–1182, March 2003.

[25] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013*

International Conference on Software Engineering, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[26] Kim Herzig and Andreas Zeller. *Mining Bug Data*, pages 131–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[27] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 30–39, May 2009.

[28] H. Hosseini, R. Nguyen, and M. W. Godfrey. A market-based bug allocation mechanism using predictive bug lifetimes. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 149–158, March 2012.

[29] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 111–120, New York, NY, USA, 2009. ACM.

[30] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd edition, 2024. Online manuscript released August 20, 2024.

[31] R. Uday Kiran, Sourabh Shrivastava, Philippe Fournier-Viger, Koji Zettsu, Masashi Toyoda, and Masaru Kitsuregawa. Discovering frequent spatial patterns in very large spatiotemporal databases. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '20, page 445–448, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Eric Knauss, Daniela Damian, Jane Cleland-Huang, and Remko Helms. Patterns of continuous requirements clarification. *Requirements Engineering*, 20(4):383–403, Nov 2015.

[33] Eric Knauss, Siv Houmb, Kurt Schneider, Shareeful Islam, and Jan Jürjens. *Supporting Requirements Engineers in Recognising Security Issues*, pages 4–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[34] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, December 2006.

[35] P. S. Kochhar, F. Thung, and D. Lo. Automatic fine-grained issue report reclassification. In *2014 19th International Conference on Engineering of Complex Computer Systems*, pages 126–135, Aug 2014.

[36] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 803–814, New York, NY, USA, 2014. ACM.

[37] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, page 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[38] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229, May 2017.

[39] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th In-*

ternational Conference on Software Engineering, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[40] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32, Nov 1997.

[41] Laura Lehtola, Marjo Kauppinen, and Sari Kujala. *Requirements Prioritization Challenges in Practice*, pages 497–508. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[42] Wenmin Li, Jiawei Han, and Jian Pei. Cmar: accurate and efficient classification based on multiple class-association rules. In *Proceedings 2001 IEEE International Conference on Data Mining*, pages 369–376, 2001.

[43] N. Limsettho, H. Hata, A. Monden, and K. Matsumoto. Automatic unsupervised bug report categorization. In *2014 6th International Workshop on Empirical Software Engineering in Practice*, pages 7–12, Nov 2014.

[44] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang. An empirical study on bug assignment automation using chinese bug data. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 451–455, Oct 2009.

[45] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, KDD'98, pages 80–86. AAAI Press, 1998.

[46] Huan Liu and Lei Yu. Toward integrating feature selection algorithms for classification and clustering. *IEEE Trans. on Knowl. and Data Eng.*, 17(4):491–502, April 2005.

[47] Edward Loper and Steven Bird. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1*, ETMTNLP '02, pages 63–70, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[48] G. A. Di Lucca, M. Di Penta, and S. Gradara. An approach to classify software maintenance requests. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 93–102, Oct 2002.

[49] Anas Mahmoud and Grant Williams. Detecting, classifying, and tracing non-functional software requirements. *Requirements Engineering*, 21(3):357–381, Sep 2016.

[50] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[51] Sharon McGee and Des Greer. Towards an understanding of the causes and effects of software requirements change: two case studies. *Requirements Engineering*, 17(2):133–155, Jun 2012.

[52] Phayung Meesad, Pudsadee Boonrawd, and Vatinee Nuipian. A chi-square-test for word importance differentiation in text classification. In *2011 International Conference on Information and Electronics Engineering*, volume 6, pages 110–114, 2011.

[53] T. Merten, M. Falis, P. Hübner, T. Quirchmayr, S. Bürsner, and B. Paech. Software feature request detection in issue tracking systems. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 166–175, Sept 2016.

[54] Mahmoud Mhashi, Roy Rada, Hafedh Mili, Geeng-Neng You, Akmal Zeb, and Antonis Michailidis. *Word Frequency Based Indexing and Authoring*, pages 131–148. Springer Netherlands, Dordrecht, 1992.

[55] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proc. Interspeech 2010*, pages 1045–1048, 2010.

[56] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, pages 120–, Washington, DC, USA, 2000. IEEE Computer Society.

[57] J. Nam and S. Kim. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463, Nov 2015.

[58] S Niharika, V Sneha Latha, and DR Lavanya. A survey on text categorization. *International Journal of Computer Trends and Technology*, 1(3):39–45, 2006.

[59] Nitish Pandey, Debarshi Kumar Sanyal, Abir Hudait, and Amitava Sen. Automated classification of software issue reports using machine learning techniques: an empirical study. *Innovations in Systems and Software Engineering*, Jul 2017.

[60] Sangameshwar Patil and B. Ravindran. Predicting software defect type using concept-based classification. *Empirical Software Engineering*, 25:1341–1378, Mac 2020.

[61] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[62] N. Pingclasai, H. Hata, and K. Matsumoto. Classifying bug reports to bugs and other requests using topic modeling. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 13–18, Dec 2013.

[63] Klaus Pohl and Chris Rupp. *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB Compliant.* Rocky Nook, 2nd edition, 2015.

[64] M. Riaz, J. King, J. Slankas, and L. Williams. Hidden in plain sight: Automatically identifying security requirements from natural language artifacts. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, pages 183–192, Aug 2014.

[65] Gema Rodriguez-Perez, Jesús M. Gonzalez-Barahona, Gregorio Robles, Dorealda Dalipaj, and Nelson Sekitoleko. *BugTracking: A Tool to Assist in the Identification of Bug Reports*, pages 192–198. Springer International Publishing, Cham, 2016.

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

[67] Furqan Rustam, Arif Mehmood, Muhammad Ahmad, Saleem Ullah, Dost Muhammad Khan, and Gyu Sang Choi. Classification of shopify app user reviews using novel multi text features. *IEEE Access*, 8:30234–30244, 2020.

[68] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., USA, 1986.

[69] Stephen R. Schach. *Object-Oriented and Classical Software Engineering Vol. 8.* McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2011.

[70] Hinrich Schütze, Christopher D. Manning, and Prabhakar Raghavan. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008.

[71] Carolyn B. Seaman. Software maintenance: Concepts and practice authored by penny grubb and armstrong a. takang world scientific, new jersey. copyright © 2003; 349 pages isbn 981-238-426-x (paperback) us$40. *J. Softw. Maint. Evol.*, 20(6):463–466, November 2008.

[72] L. Shi, Q. Wang, and M. Li. Learning from evolution history to predict future requirement changes. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 135–144, July 2013.

[73] Vandita Singh, Bhupendra Kumar, and Tushar Patnaik. Feature extraction techniques for handwritten text in various scripts: a survey. *International Journal of Soft Computing and Engineering*, 1(3):238–241, 2013.

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

[75] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 10th edition, 2015.

[76] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[77] P. Terdchanakul, H. Hata, P. Phannachitta, and K. Matsumoto. Bug or not? bug report classification using n-gram idf. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 534–538, Sep. 2017.

[78] Gary M. Weiss and Foster Provost. Learning when training data are costly: the effect of class distribution on tree induction. *J. Artif. Int. Res.*, 19(1):315–354, October 2003.

[79] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[80] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, page 412–420, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[81] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 404–415, May 2016.

[82] Bei Yu. *An Evaluation of Text Classification Methods for Literary Study*. PhD thesis, USA, 2006.

[83] Andreas Zeller. *Can We Trust Software Repositories?*, pages 209–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[84] ChengXiang Zhai and Sean Massung. *Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining*, volume 12. Association for Computing Machinery and Morgan & Claypool, 2016.

[85] Jie Zhang, XiaoYin Wang, Dan Hao, Bing Xie, Lu Zhang, and Hong Mei. A survey on bug-report analysis. *Science China Information Sciences*, 58(2):1–24, Feb 2015.

[86] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process*, 28(3):150–176, 2016. JSME-15-0091.R2.

[87] Davor Čubranić. Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, pages 92–97. KSI Press, 2004.

Appendix

Appendix

.1 Hold Out Evaluation

.1.1 Http Client Project

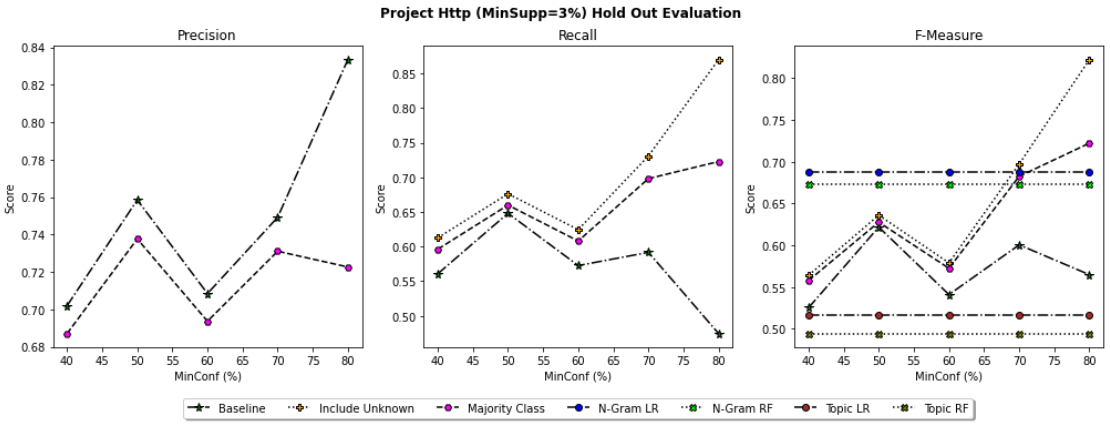

Figure 1: Http-Client Project

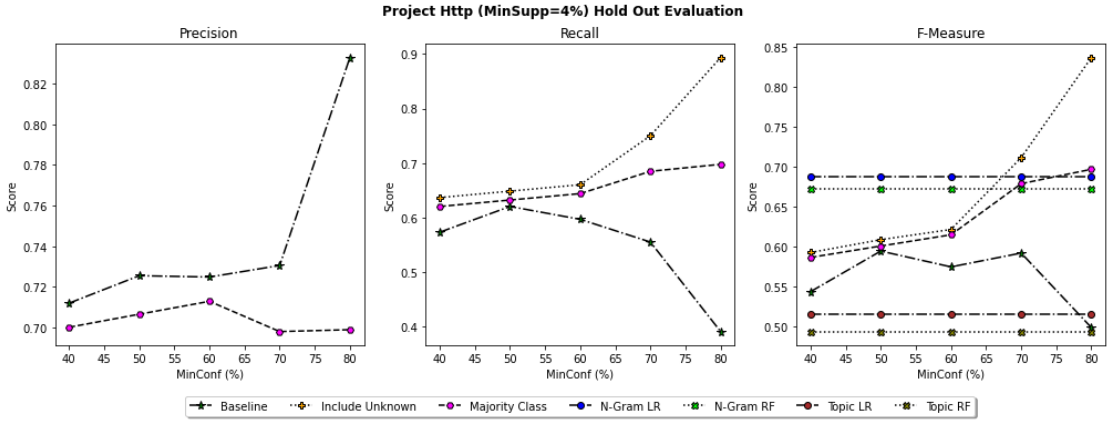

Figure 2: Http-Client Project

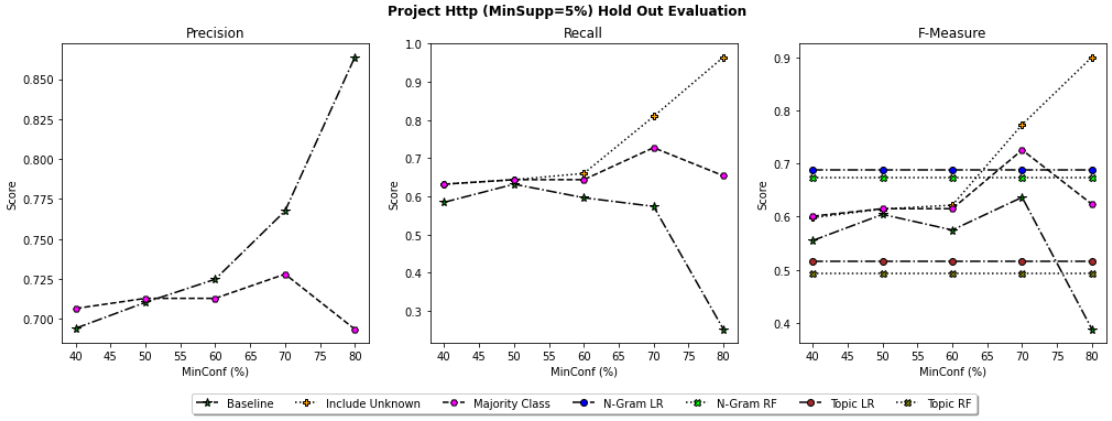

Figure 3: Http-Client Project

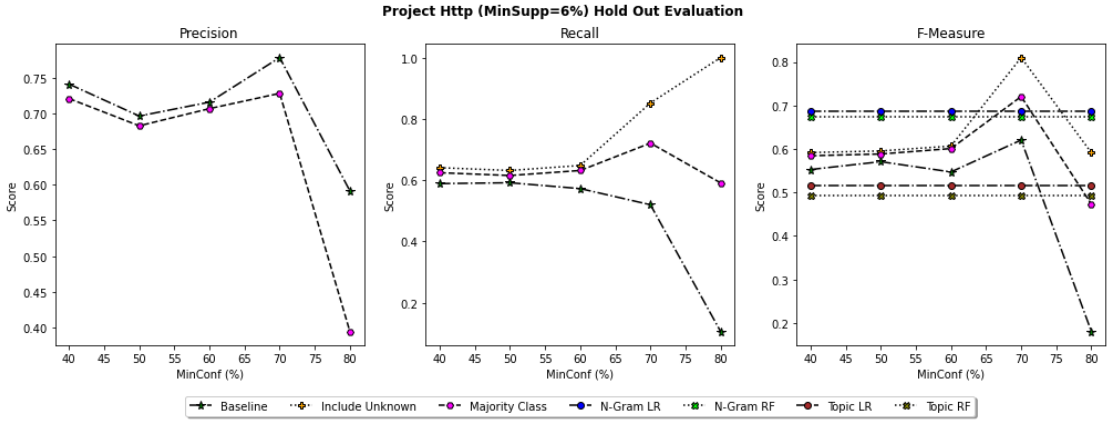

Figure 4: Http-Client Project

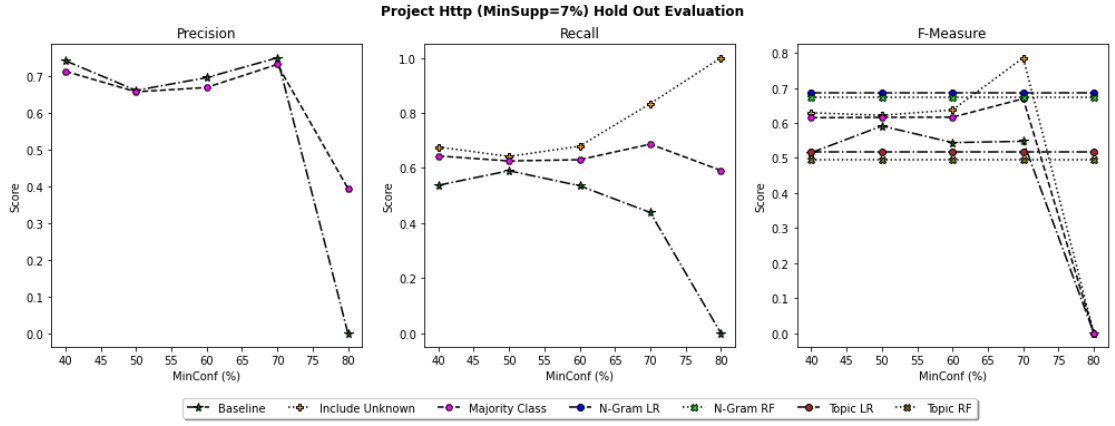

Figure 5: Http-Client Project

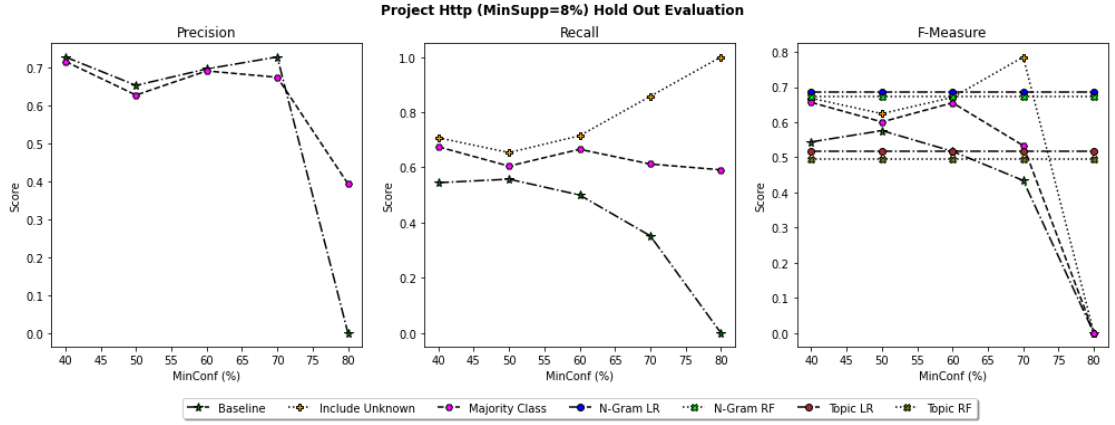

Figure 6: Http-Client Project

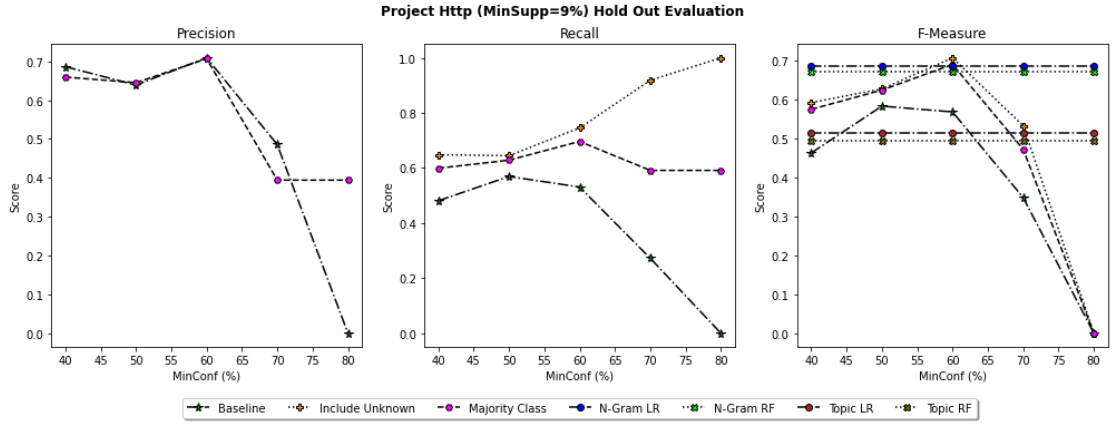

Figure 7: Http-Client Project

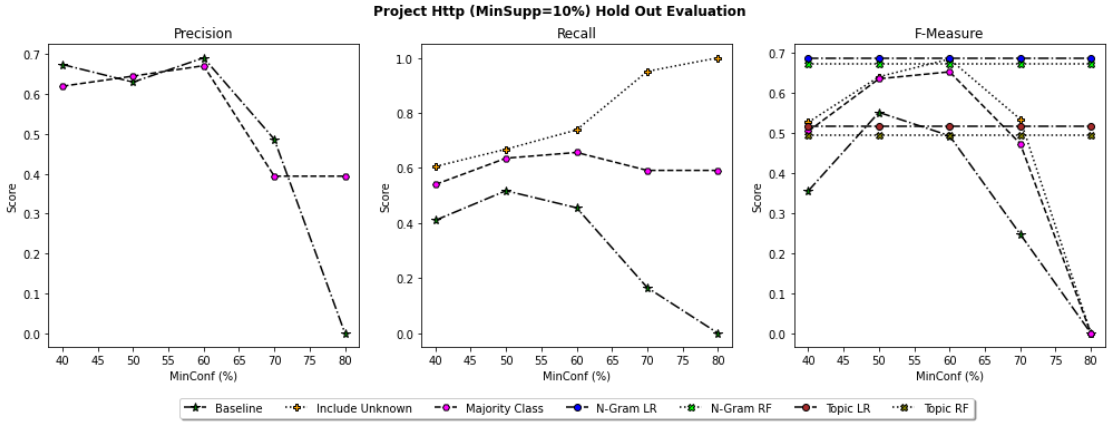

Figure 8: Http-Client Project

.1.2 Lucene Project

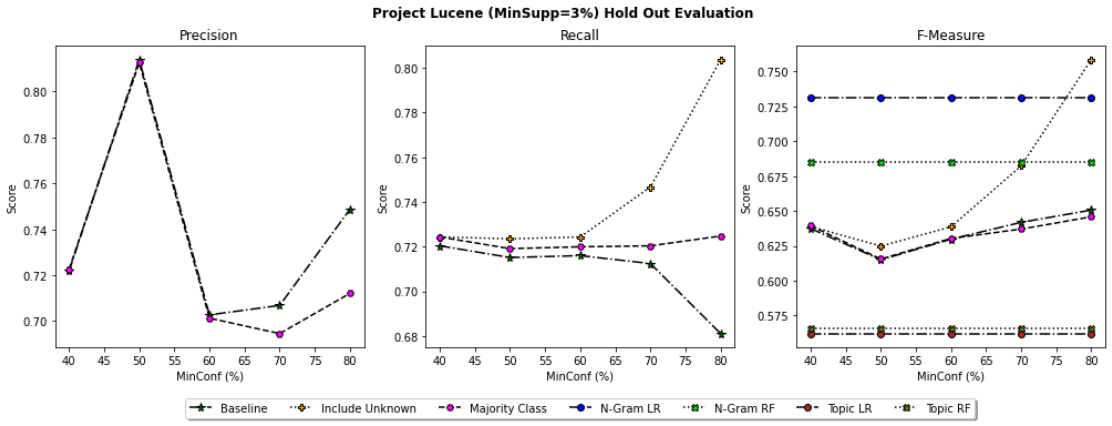

Figure 9: Lucene Project

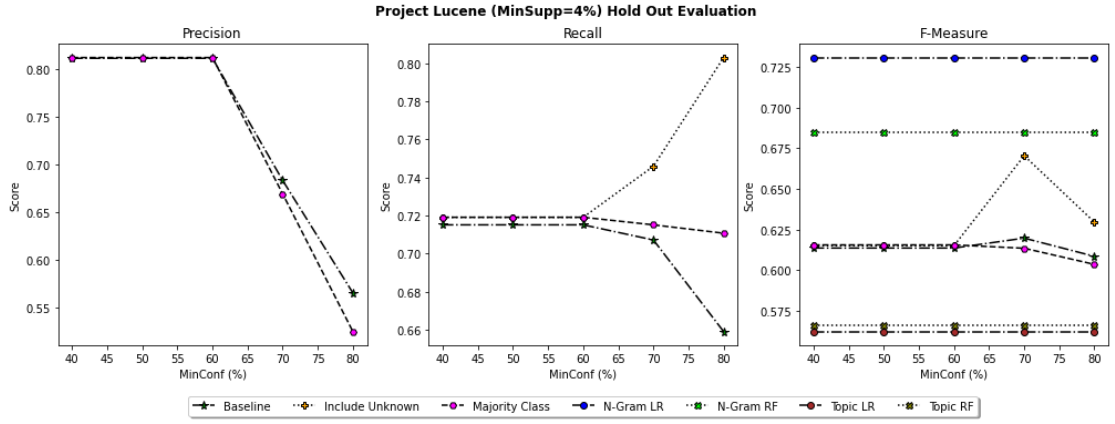

Figure 10: Lucene Project

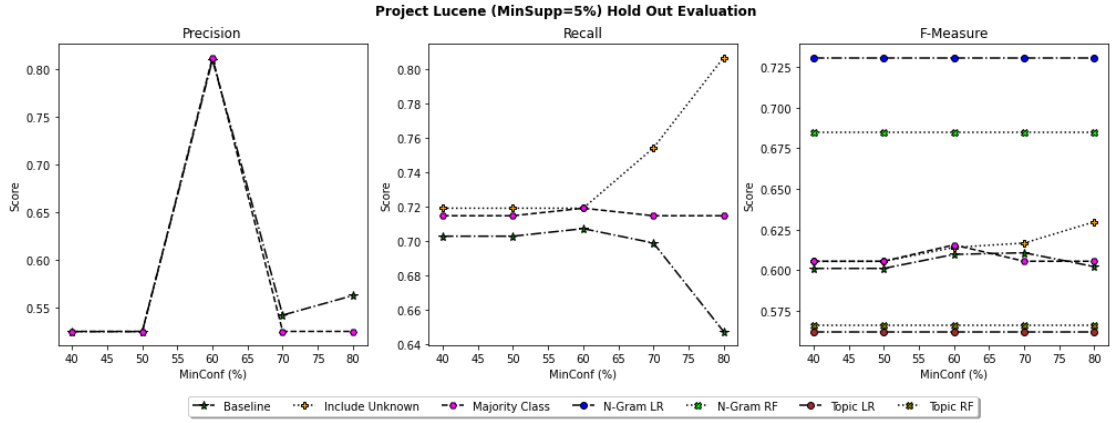

Figure 11: Lucene Project

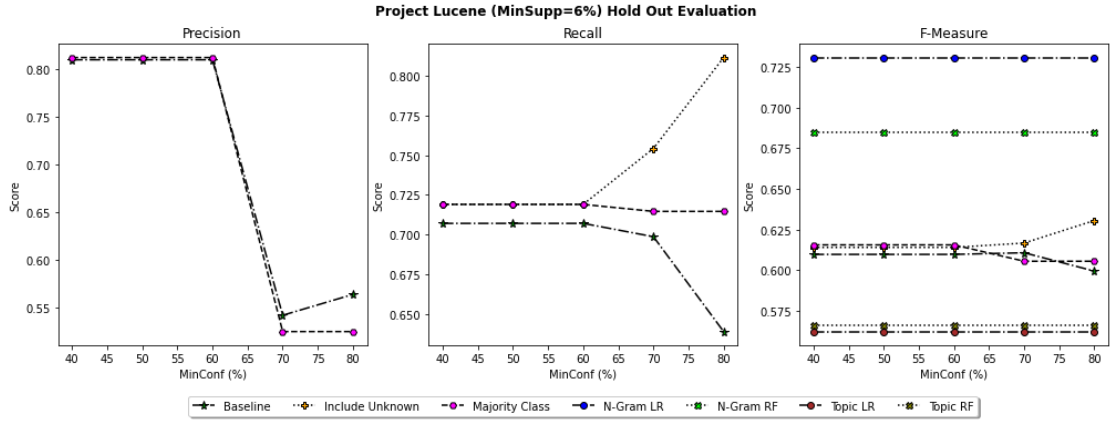

Figure 12: Lucene Project

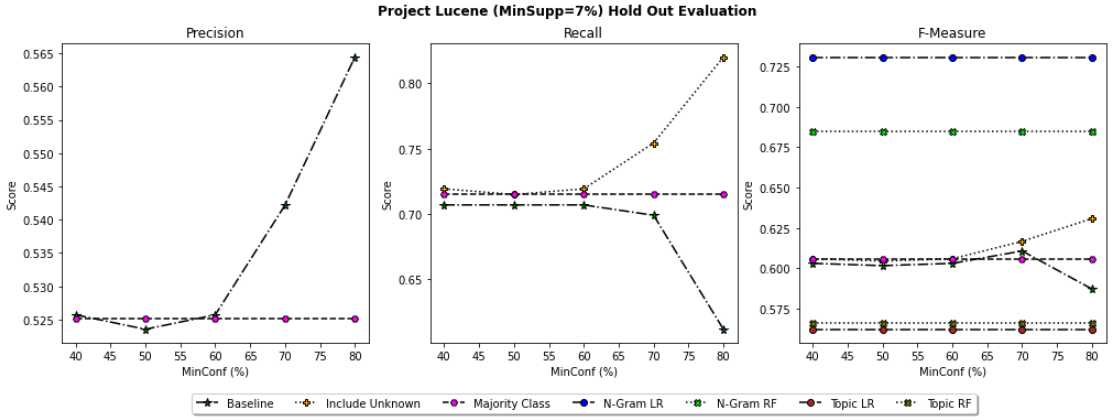

Figure 13: Lucene Project

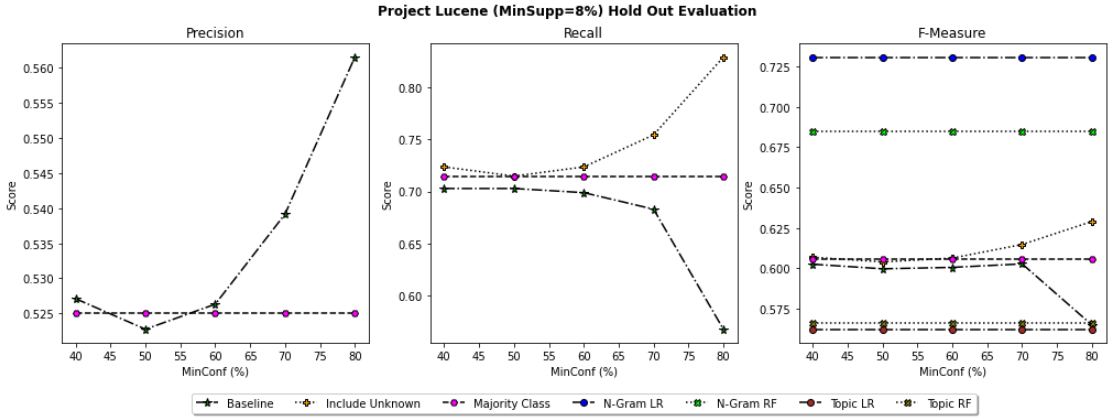

Figure 14: Lucene Project

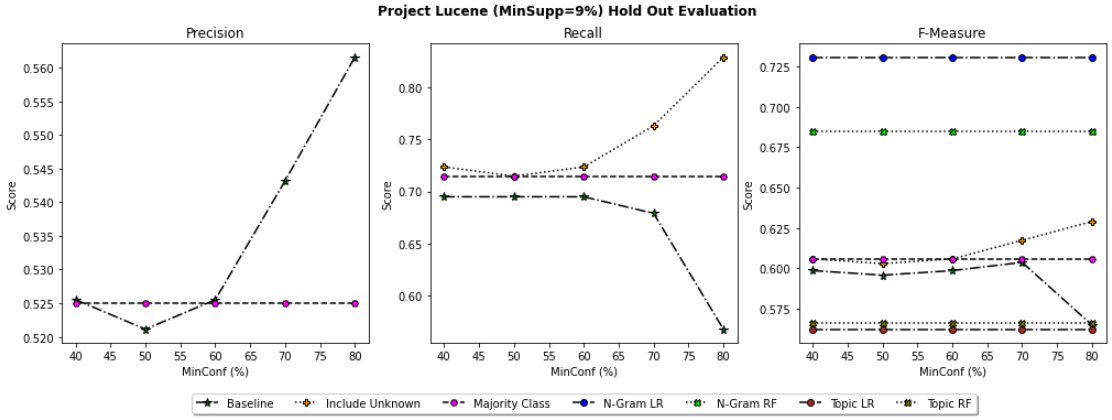

Figure 15: Lucene Project

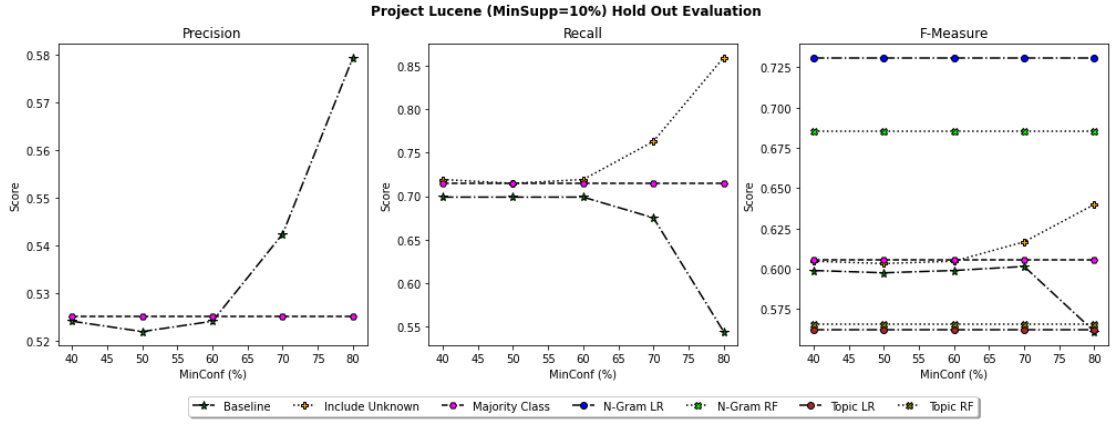

Figure 16: Lucene Project

.1.3 Jackrabbit Project

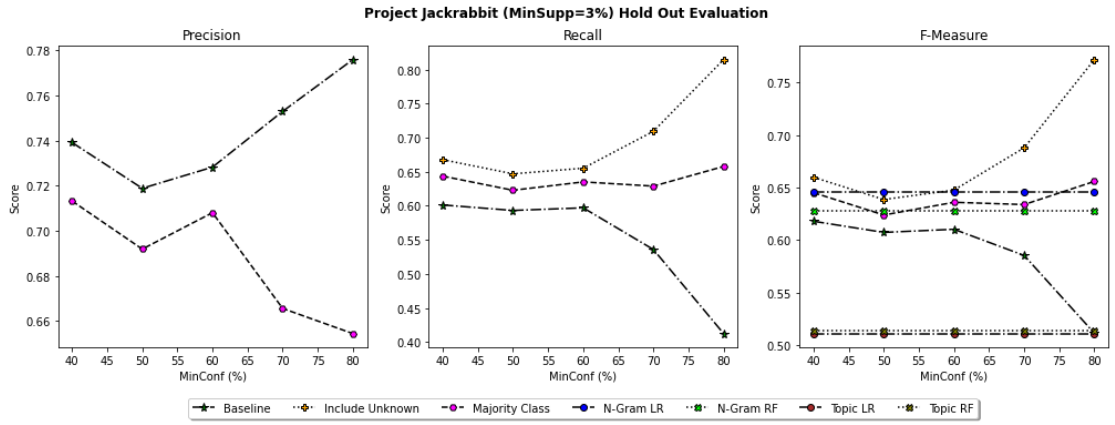

Figure 17: Jackrabbit Project

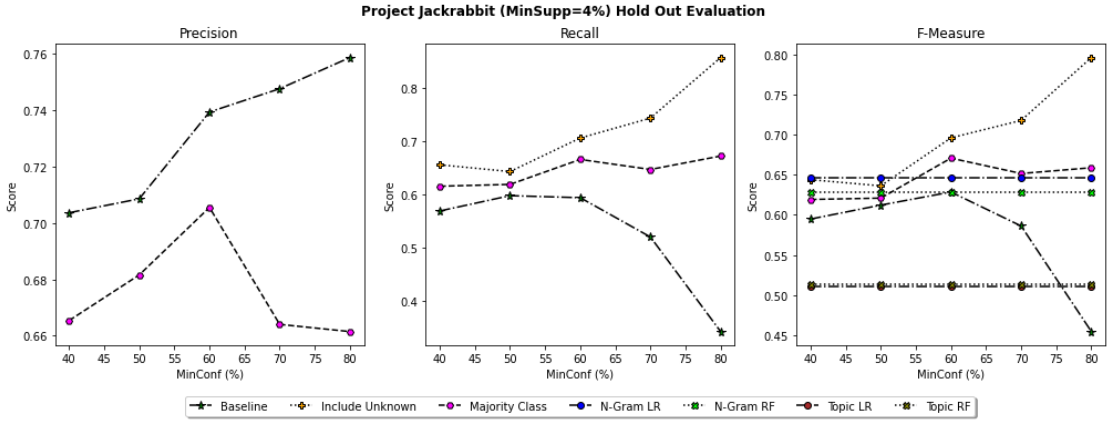

Figure 18: Jackrabbit Project

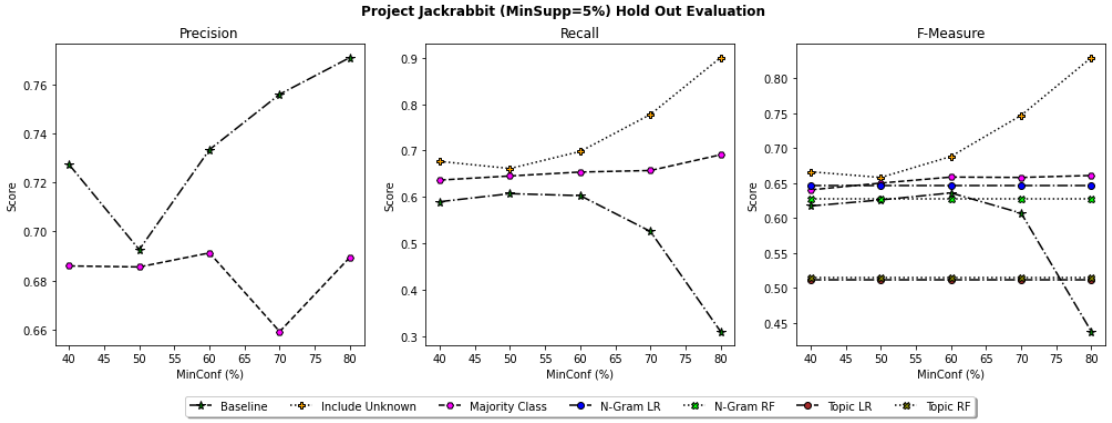

Figure 19: Jackrabbit Project

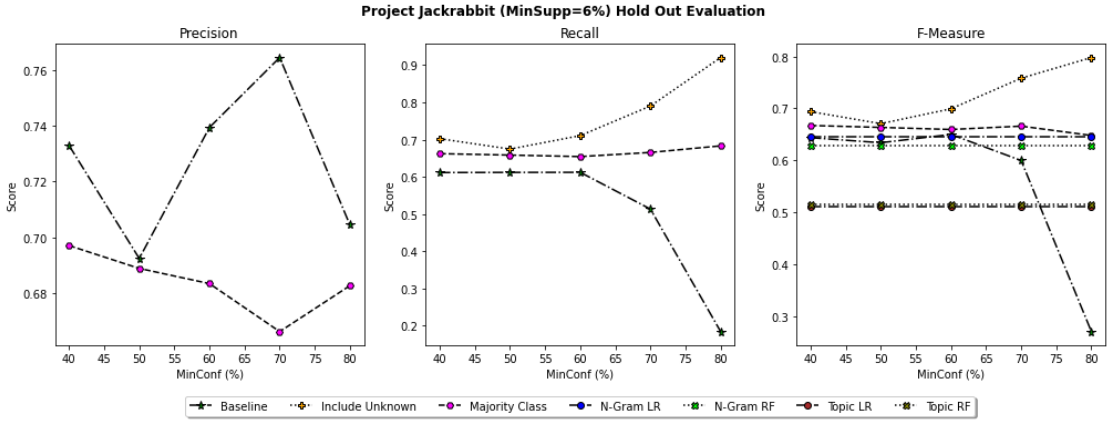

Figure 20: Jackrabbit Project

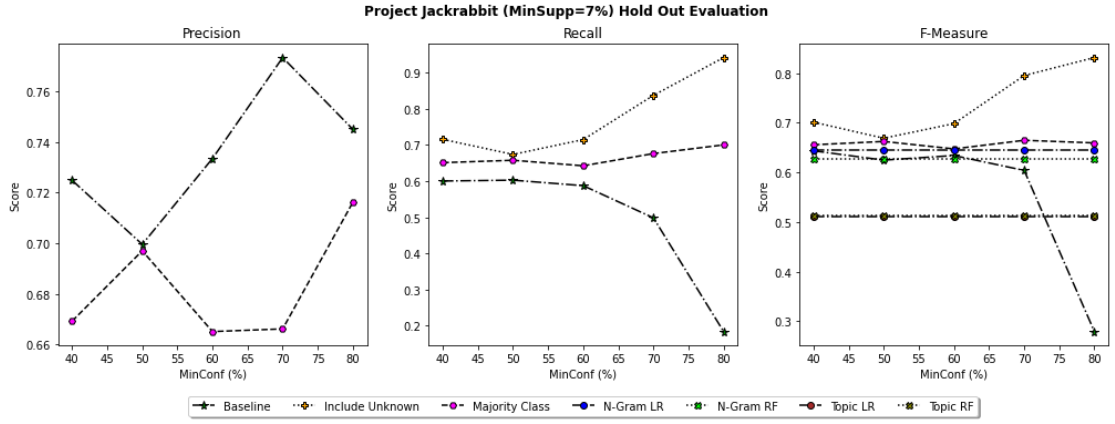

Figure 21: Jackrabbit Project

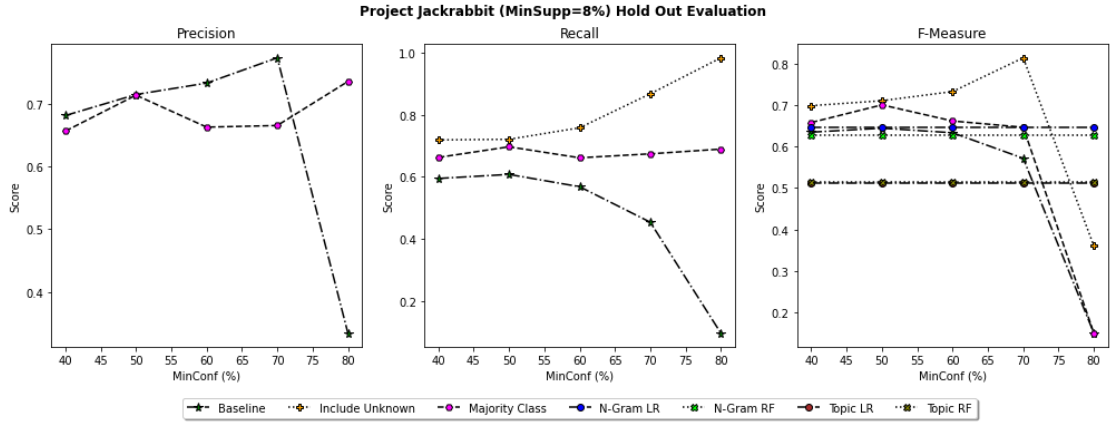

Figure 22: Jackrabbit Project

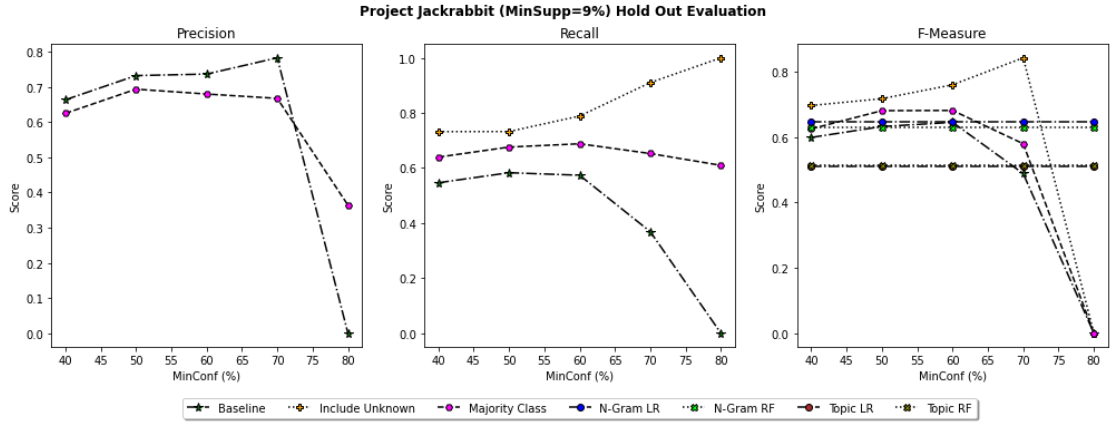

Figure 23: Jackrabbit Project

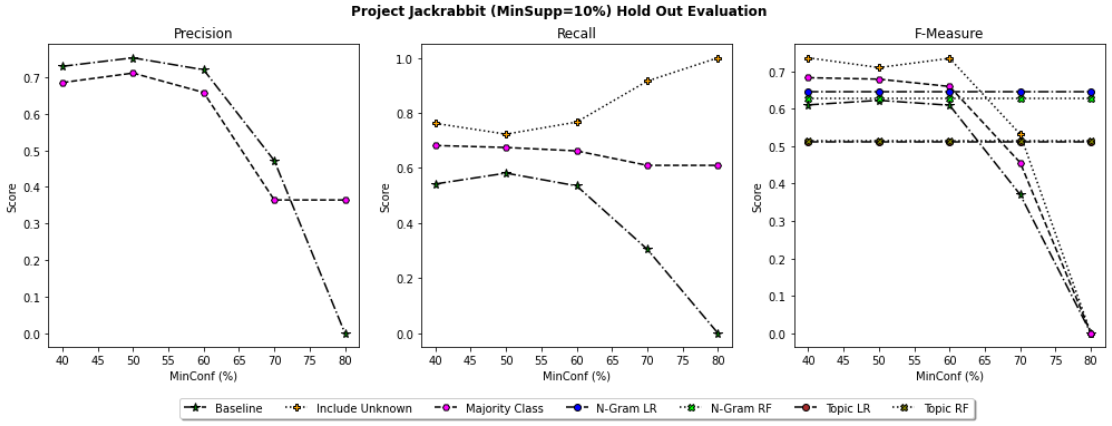

Figure 24: Jackrabbit Project

.2 Cross Validation

Http-Client Project

Experiment results using TF-IDF as feature selection

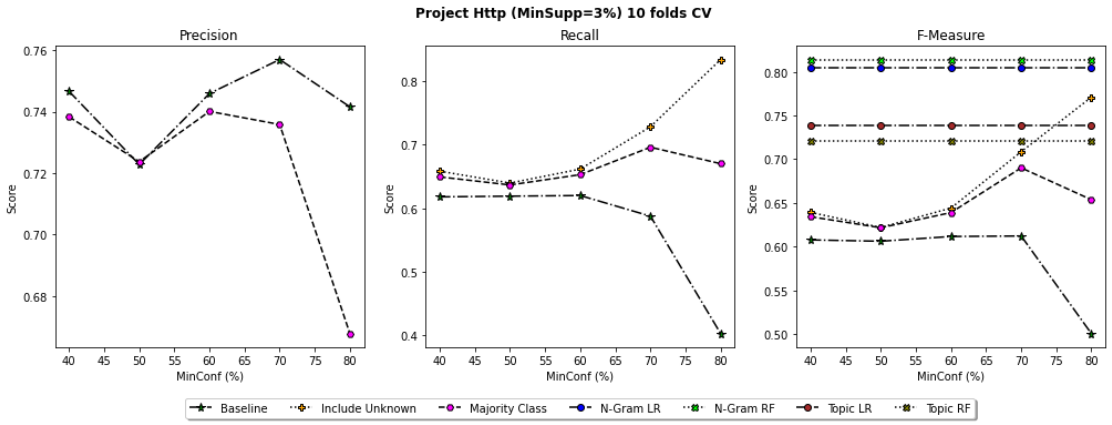

Figure 25: Http-Client Project

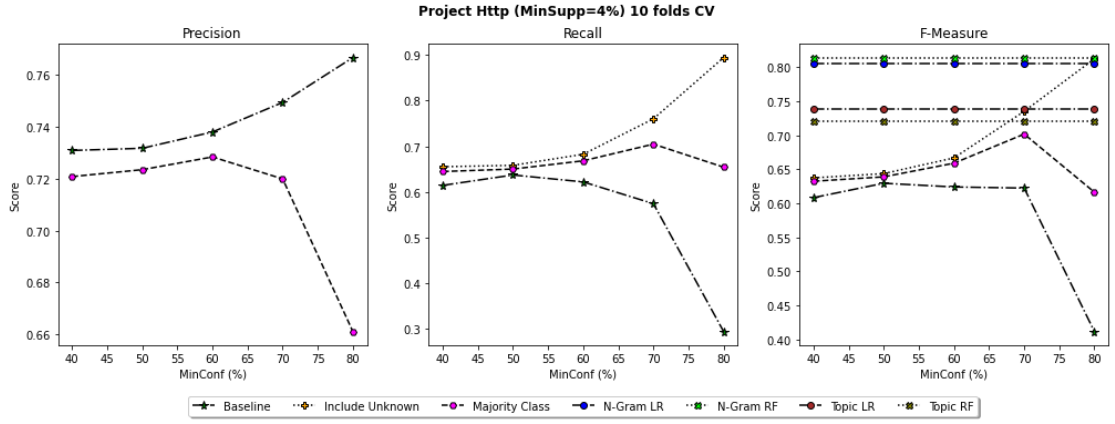

Figure 26: Http-Client Project

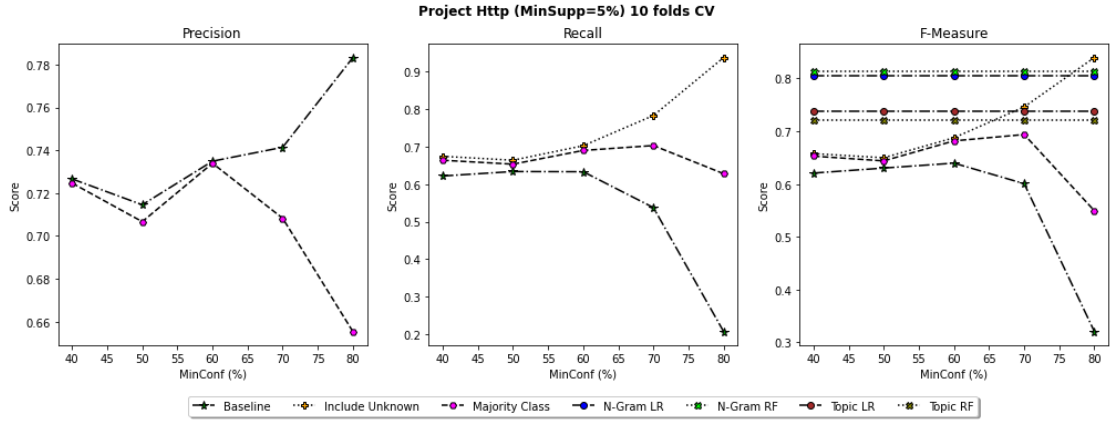

Figure 27: Http-Client Project

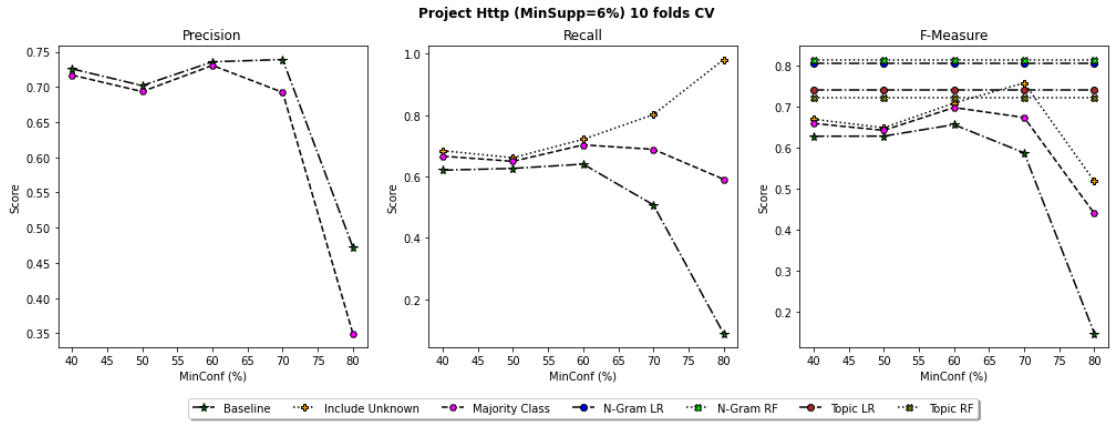

Figure 28: Http-Client Project

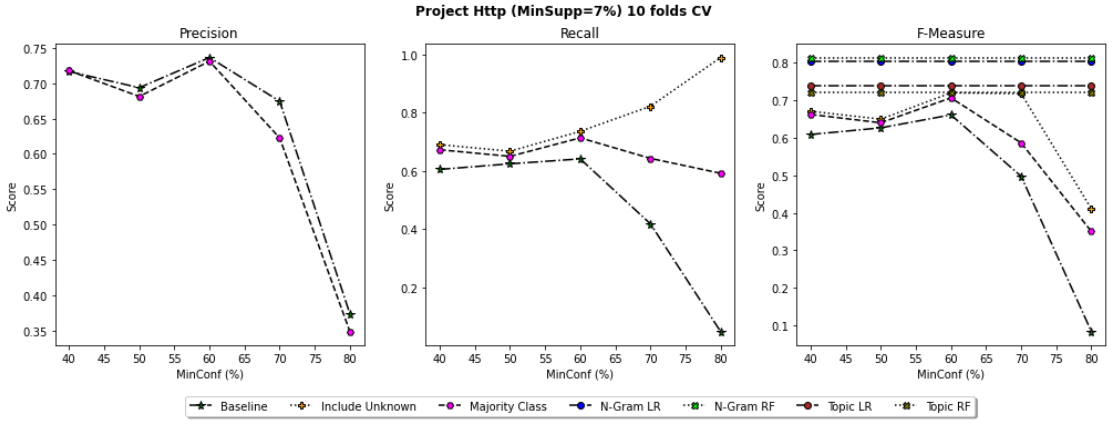

Figure 29: Http-Client Project

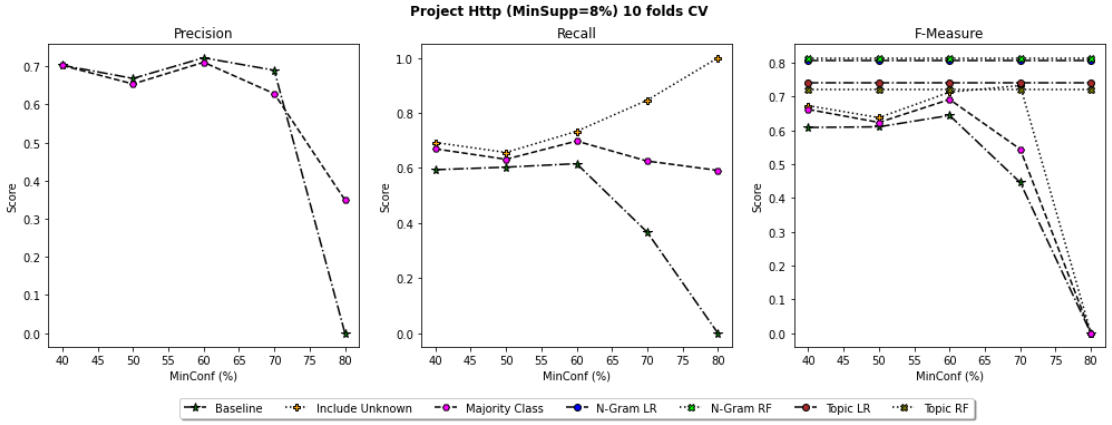

Figure 30: Http-Client Project

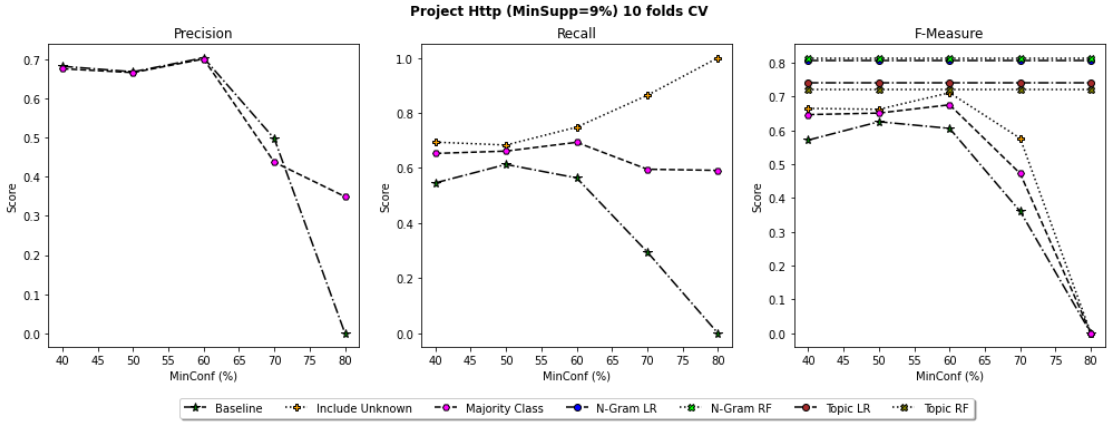

Figure 31: Http-Client Project

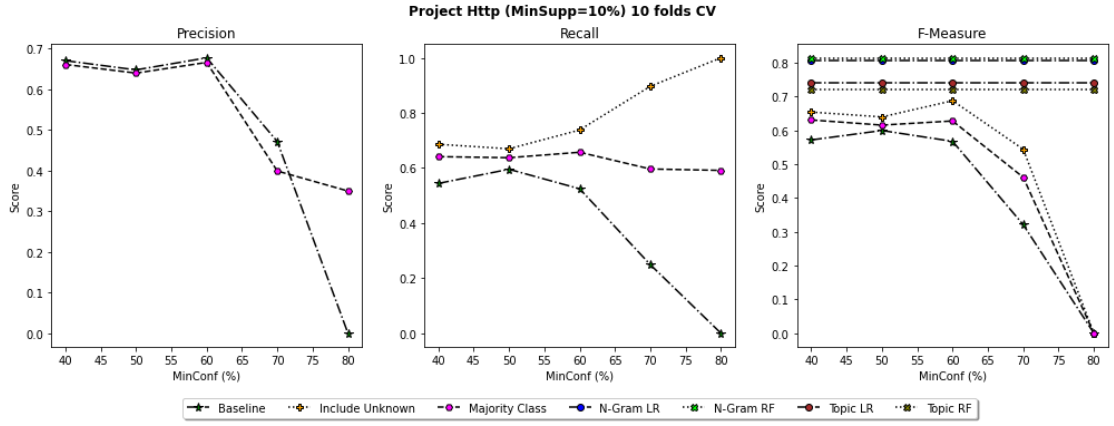

Figure 32: Http-Client Project

.2.1 Lucene Project

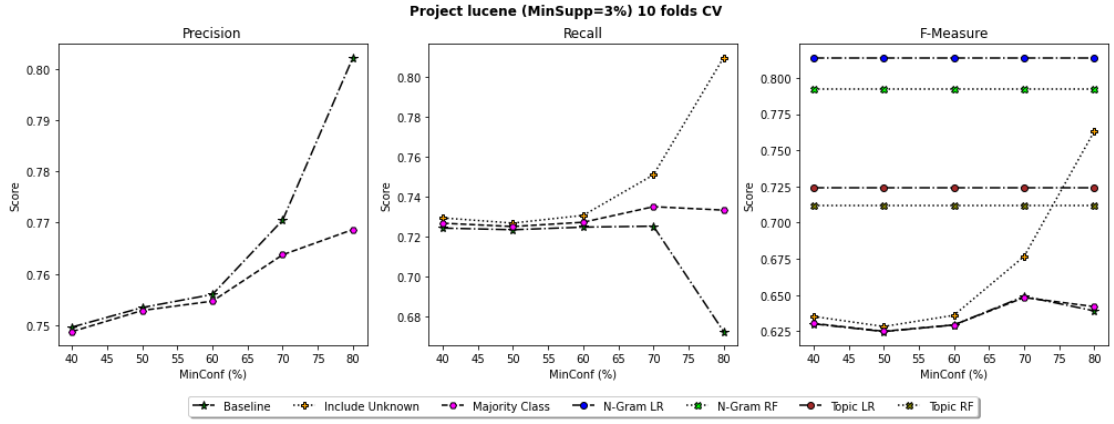

Figure 33: Lucene Project

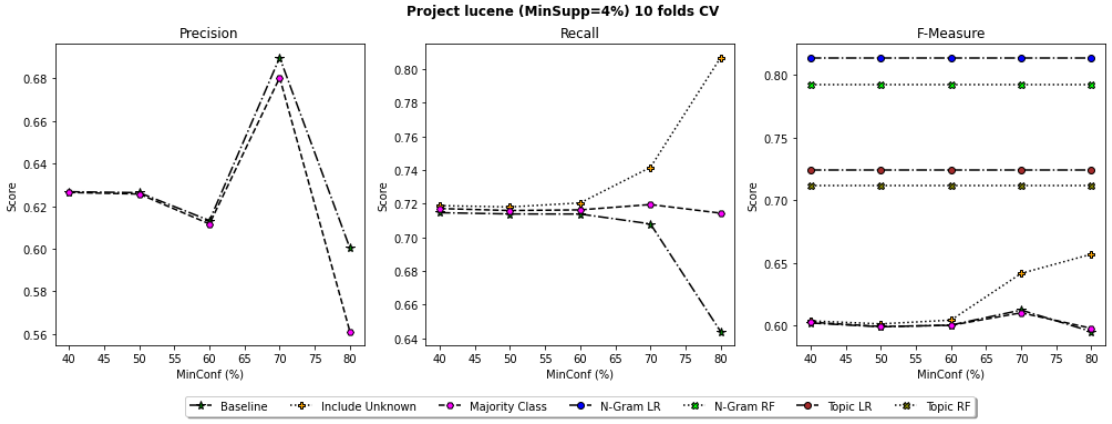

Figure 34: Lucene Project

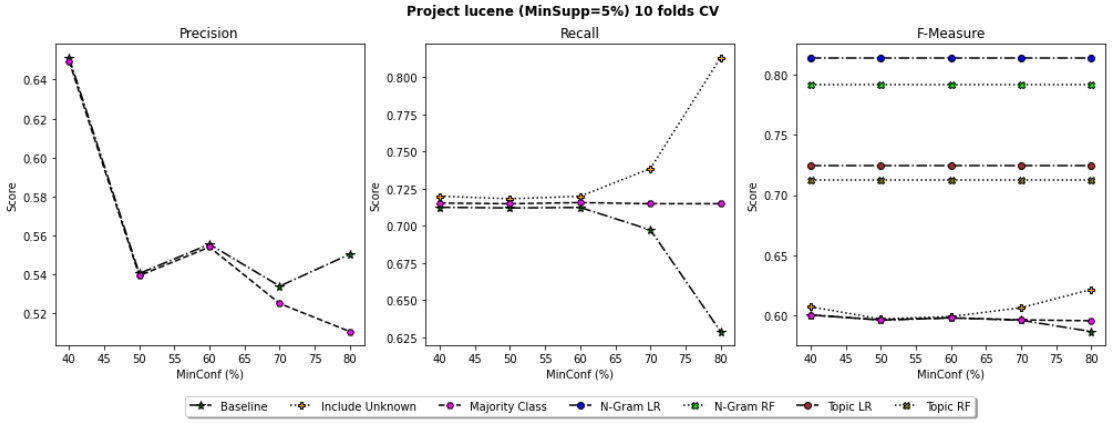

Figure 35: Lucene Project

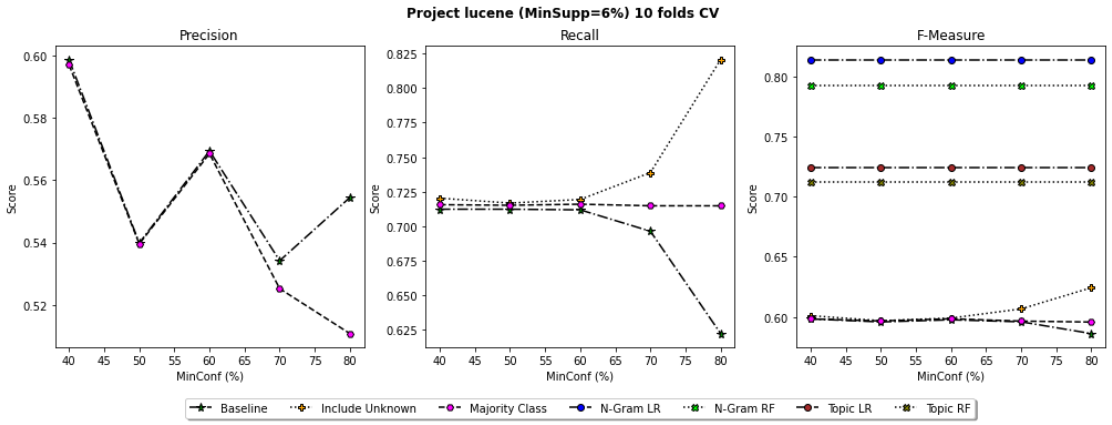

Figure 36: Lucene Project

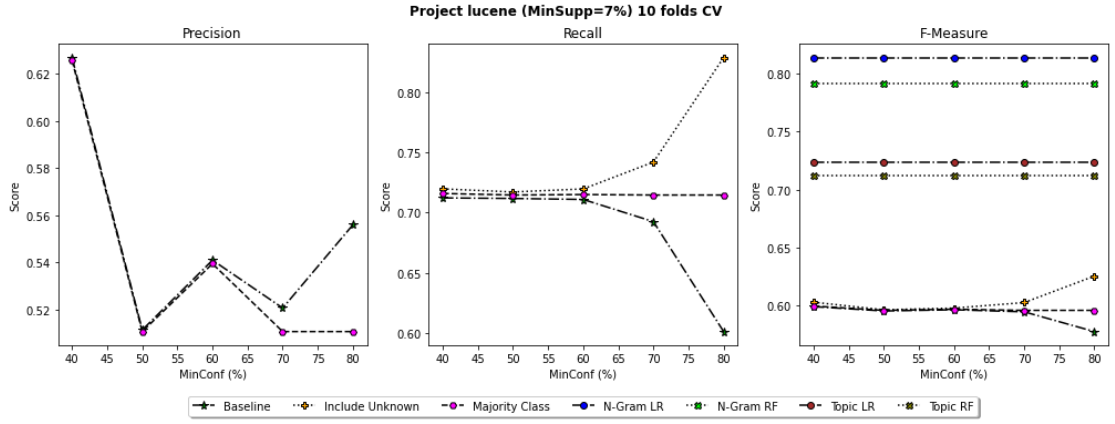

Figure 37: Lucene Project

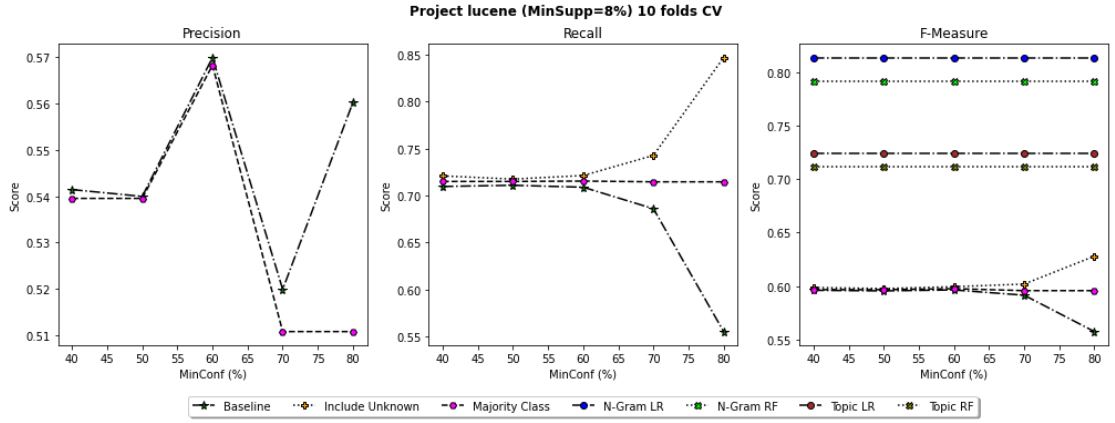

Figure 38: Lucene Project

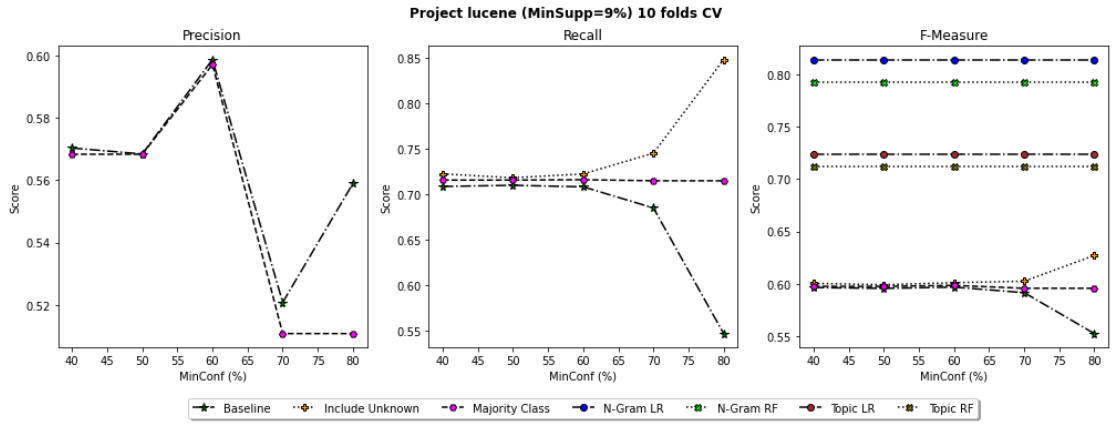

Figure 39: Lucene Project

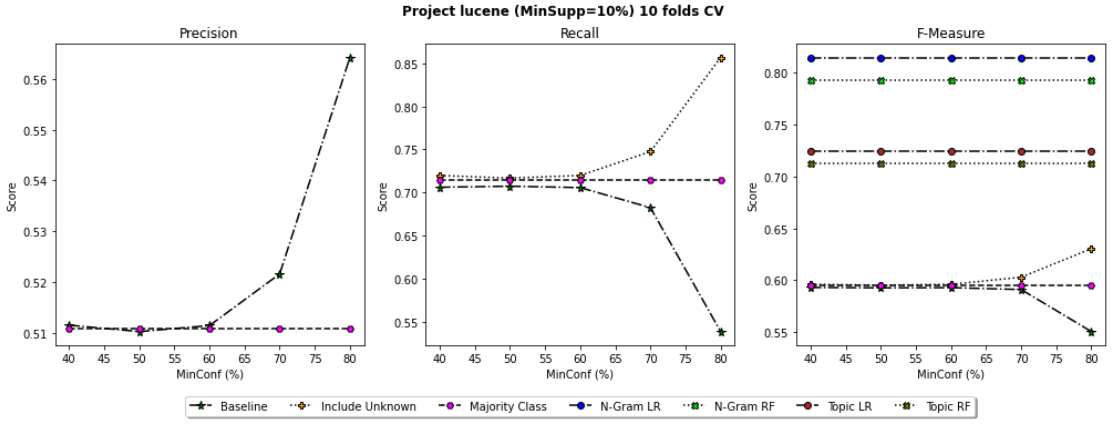

Figure 40: Lucene Project

.2.2 Jackrabbit Project

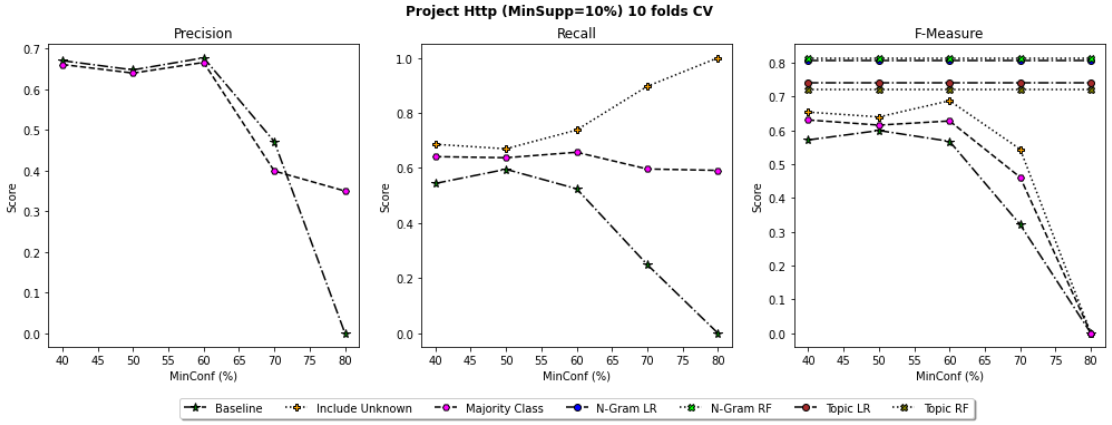

Figure 41: Jackrabbit Project

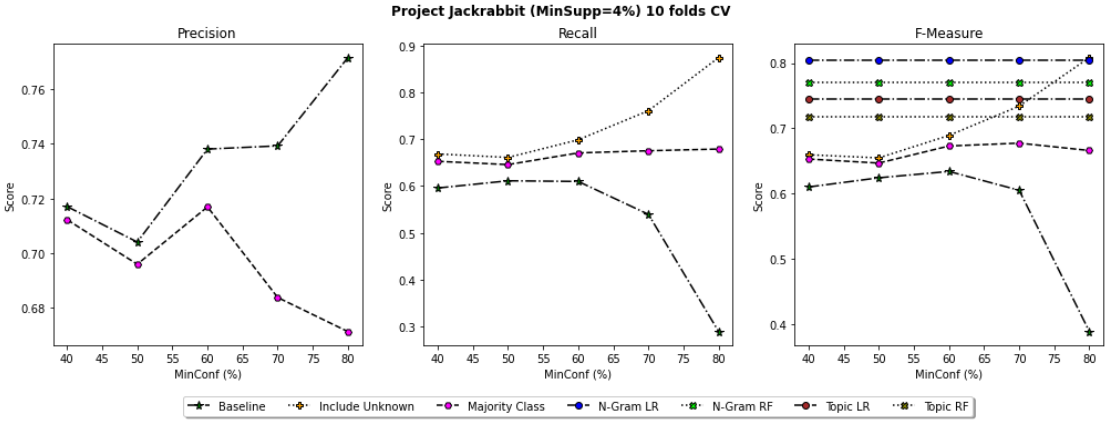

Figure 42: Jackrabbit Project

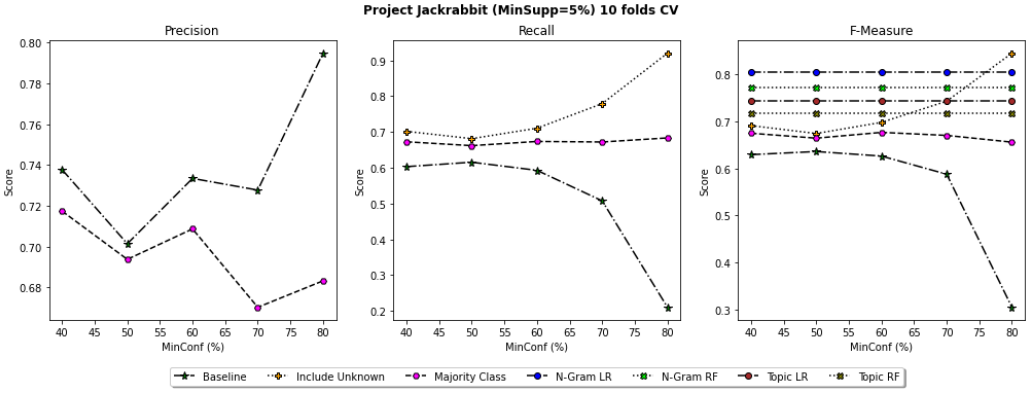

Figure 43: Jackrabbit Project

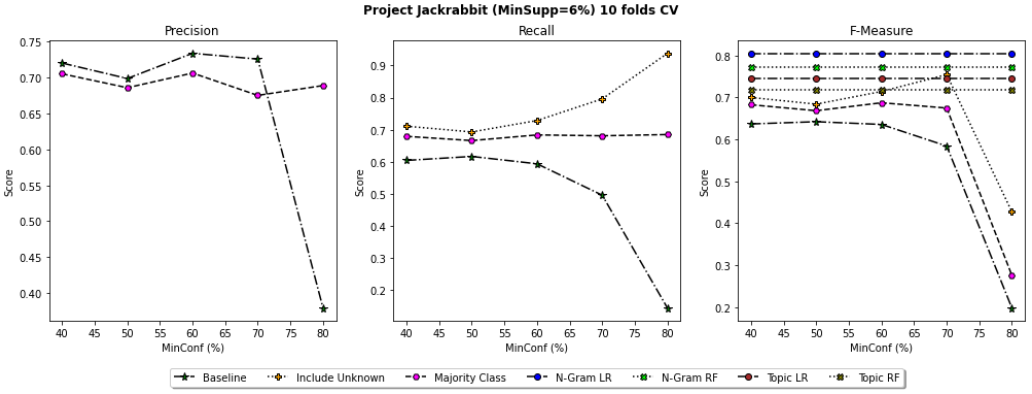

Figure 44: Jackrabbit Project

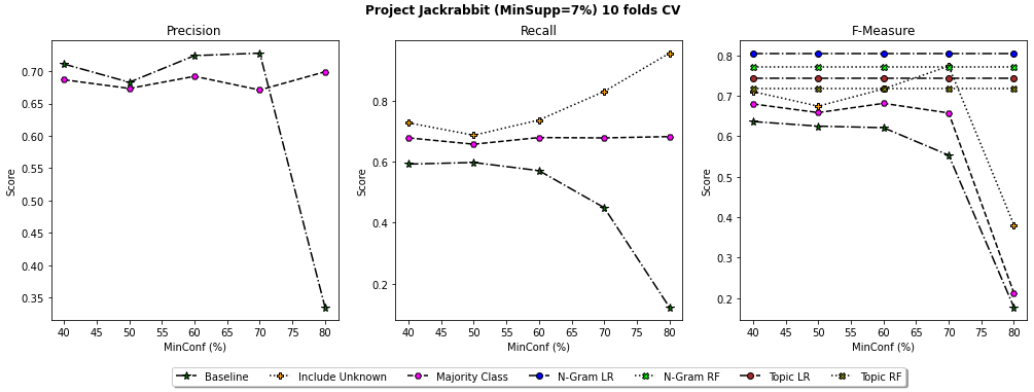

Figure 45: Jackrabbit Project

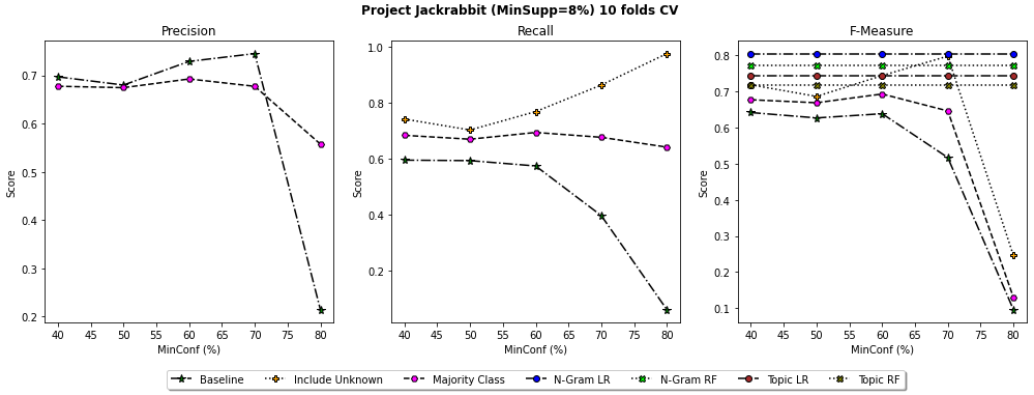

Figure 46: Jackrabbit Project

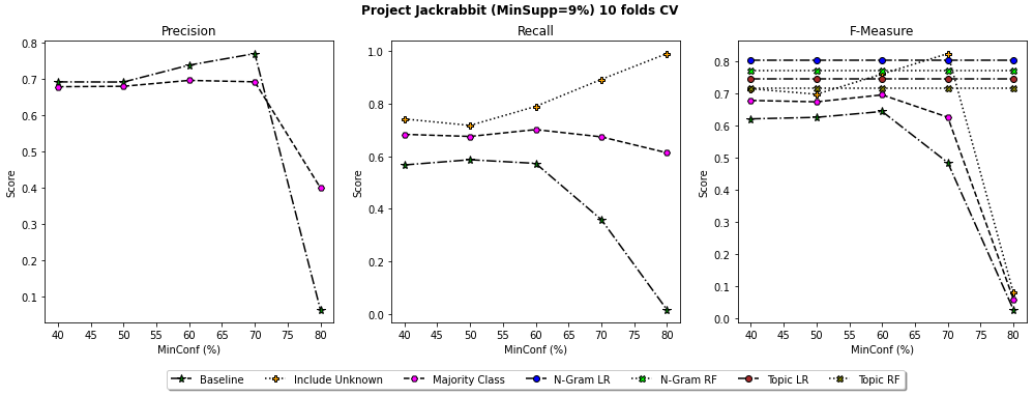

Figure 47: Jackrabbit Project

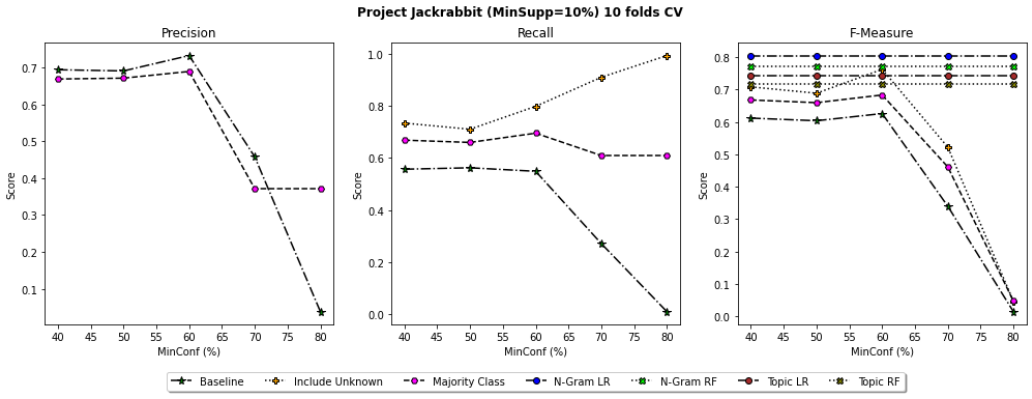

Figure 48: Jackrabbit Project