An ENSEMBLE Technique for Linking Software Features to Code Components

By
Din Muhammad
Azharuddin

Computer Science and Engineering Discipline
Khulna University
Khulna - 9208, Bangladesh

December, 2023

An ENSEMBLE Technique for Linking Software Features to Code Components

Din Muhammad
Student ID: 190237
E-mail: 190237@ku.ac.bd

Azharuddin
Student ID: 190238
E-mail: 190238@ku.ac.bd

Computer Science and Engineering Discipline
Khulna University
Khulna - 9208, Bangladesh

December, 2023

December, 2023
Computer Science and Engineering Discipline
Khulna University, Khulna

The undersigned hereby certify that Din Muhammad and Azharuddin of the Computer Science and Engineering Discipline, Khulna University, Khulna have successfully completed the thesis entitled ”An ENSEMBLE Technique for Linking Software Features to Code Components ” in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering (CSE) at Khulna University, Khulna.

Thesis Supervisor

Dr. Amit Kumar Mondal
Associate Professor
Computer Science and Engineering Discipline
Khulna University, Khulna

Second Examiner

Dr. Manishankar Mondal
Associate Professor
Computer Science and Engineering Discipline
Khulna University, Khulna

Head of the Discipline

Professor Dr Abu Shamim Mohammad Arif
Professor
Computer Science and Engineering Discipline
Khulna University, Khulna

December, 2023
Computer Science and Engineering Discipline
Khulna University, Khulna

We undersigned hereby declare that this thesis is a presentation of our original research work. Wherever contributions of others are involved, every effort is made to indicate this clearly, with due reference to the literature, and acknowledgment of collaborative research and discussions. The work was done under the guidance of Dr. Amit Kumar Mondal, at the Computer Science and Engineering Discipline, Khulna University, Khulna.

Din Muhammad

Azharuddin

Acknowledgment

First of all, we would like to thank Almighty Allah for giving us enough mental and physical strength to complete our thesis properly. We would like to express our sincere gratitude to our supervisor Dr. Amit Kumar Mondal for his cooperation, suggestion, guidance, and continuous encouragement throughout the course of the study. We are highly grateful to our second examiner Dr. Manishankar Mondal for reviewing this report and giving us suggestions to improve it.

Besides our supervisor, we would like to acknowledge our honorable teachers of Computer Science and Engineering Discipline for their encouraging support and discussion. We also thank our parents for their encouragement, support, and attention. We are also thankful to our classmates for their moral support which helped us to accomplish our thesis.

Dedication

To our parents and our family. Both our parents give enough inspiration and encouragement to complete our thesis work.

Abstract

In the realm of software development and maintenance, the manual linking of software feature documents to code components is a critical yet challenging task for developers. This linkage is essential for various purposes such as implementing new features, documentation, tracking, and test case design. Additionally, it plays a pivotal role in third-party inspections to ensure compliance with regulations governing different software products. However, the manual linking process is fraught with challenges, including errors and time constraints. To address these challenges, previous studies have proposed automated techniques. Despite their potential benefits, these techniques often encounter issues related to accuracy, cost, and explainability. In response to these limitations, our work is dedicated to addressing three crucial dimensions: accuracy, cost-effectiveness, and overall performance. Our proposed solution involves an innovative ensemble approach designed to simultaneously improve accuracy and reduce costs. Through a series of extensive experiments conducted on two distinct software projects, we observed a noteworthy performance enhancement when linking software features to three fundamental architectural abstractions of code components: modules, classes, and methods. Comparing our ENSEMBLE approach to baseline lightweight techniques such as the Vector Space Model (VSM), Latent Semantic Indexing (LSI), and A Contextual Thematic Approach for Linking Features to Muilti-level Software Architectural Component (FSECAM), our approach demonstrated superior results. The evaluation, encompassing datasets from 18 cases across the two projects, showcased higher precision rates, recall, and F1 scores, particularly in the context of Modules and Classes within proprietary projects. Out of the 18 cases evaluated, our proposed ENSEMBLE approach exhibited superior results in 14 cases, underscoring its effectiveness in enhancing the linking of software feature documents to code components. This research contributes valuable insights into the realm of automated techniques for linking software documentation to code, addressing key challenges, and paving the way for more accurate, cost-effective, and efficient practices in software development and maintenance.

Table of Contents

List of Figures

List of Tables

Chapter I

Introduction

1.1 Background

Software is everywhere in our daily life. We cannot imagine a single day without using any kind of software service. We use software for communication, social media, entertainment, education, transportation and health care, banking and finance, travel and navigation, e-commerce and home automation, etc. all make our daily lives easy.[1] [2] We use many software features in our lives such as authentication and authorization in web applications, official documentation, security and privacy, and so on[3].

The basic activity of software development and maintenance are requirement gathering and analysis it invokes understanding and basic needs of software user. After establishing the requirement design phase starts. It includes the blueprint or architecture of the system. It also includes structure, interface, class, module, method, function, etc. The next step of software development and maintenance is implementation. It involves the remake phase. In this phase, the developer converts the design to code. Then the next step is software testing. Software testing is very crucial that ensure the satisfaction of the user. This phase adds the test case, bug fixing, and validating against the expected result. After completing the testing the next step is deployment. This phase mainly focuses on the environment of the user's system. This includes installing the software of the user's system.[4]

Once the software is deployed, then the maintenance phase arises. This step involves bug fixing, performance optimization, security updates, and most importantly enhancement to meet changing user needs. Software development and maintenance include the next phase is support and documentation.[5] This phase invokes providing support to the end-user, addressing their queries and issues, and maintaining user manuals, technical guidelines, and release notes. The last step of software development and maintenance is software updates and enhancement. Over time the software may need to update and enhance the new feature of the existing software system. This phase includes improving performance, changing the technology environment, analyzing user feedback then updating features[6]. For changing requirements and enhancing features, the development team is required to discover the code component associated with the requirements. That's why linking features to code components is very crucial for the above development phase [7].

Feature location for program comprehension, whether performed automatically or manually, necessitates the association of multiple facets to enhance accuracy and rationale.

Various approaches exist for linking features to code components, falling into categories such as manual, static information retrieval, dynamic, and deep learning techniques. Despite the progress in automated techniques, software projects encounter ongoing challenges including bugs, security issues, frequent deployment, rising change costs, difficulties in pooling expertise, and the need for proper documentation to search for urgent information.[8] Simultaneously, due to socio-economic changes, the workforce exhibits less interest in complex and stressful tasks. Performing supportive tasks mentioned in the previous paragraph several weeks after implementation can significantly increase mental stress. Reliable tools are essential to accomplish these tasks while minimizing the associated stress. Moreover, when a developer departs from a project or company, they are often required to document their design and implementation parts. This proves challenging, as re-analyzing several years of work may not be straightforward, and the company may be uncertain about the completeness of the documentation. For instance, our discussions with two software companies revealed that they resorted to this practice due to a lack of reliable and low-cost tools. The challenge also extends to onboarding new members of a development team. In summary, existing feature linking techniques face three critical challenges: (i) accuracy, (ii) costs, and (iii) explainable properties. Conversely, limited effort has been directed toward linking features with multi-abstraction levels of code components. [9].

However, the manual process of software development and maintenance has a lot of limitations. Such as manual processes maintained by a human when human mistakes then occur an error. Manual processes are often not scalable when dealing with a complex or large software system then there is a lack of scalability. Manual processes are generally slower and they consume more time. This process is susceptible to inconsistency. Manual processes often lack proper mechanisms for traceability and auditability. It may hinder code reusability. This process may restrict effective collaboration among team members, especially when working on distributed teams or in different time zones. The cost of a manual process is high because this process leads to increased costs in terms of time, effort, and resources. In summary, the overall efficiency of the manual process is low. Also, there are a lot of automated tools for software development and maintenance but these tools have some limitations as gathering and managing requirements is a crucial step in software development, but it can be a complex and error-prone process. Code generation tools are available but there are limitations in terms of the complexity of code they can generate A lot of software testing tools exist but this accuracy is not high in complex software systems[10]. Automated tools for software maintenance and evaluation such as bug detection, code refactoring, and impact analysis are continuously improving but it's very challenging for complex codebases. Applying natural language processing techniques such as code to document generation, code summarizations, and code generations from natural language requirements is not more accurate, still improving these tools[11].

Experiments with developers reveal that automated tools lack a rational explanation for outcomes, making it challenging for practitioners to understand decision-making processes. The difficulty arises from the inadequacy of existing techniques to identify relevant properties, impacting developers' confidence in the results. Deep learning models,

with large training data samples and numerous node parameters, pose challenges in terms of explainability, raising concerns about risky situations in safety-critical software inspection. Similarly, techniques like LSI and VSM employ hidden mathematical properties that are difficult to align with the subjective properties of components for an effective explanation., etc [12] [13].
Our hypothesis, inspired by Damevski et al. is that developers use only a few terms for code searches within a project. We observed that not every semantic meaning in a description is essential for representing a requirement in code components, leading to unnecessary efforts. We focus on relevant parts and reduce inconsistencies in accuracy, costs, and explainability. Our approach, ENSEMBLE, maps features/issues to code abstraction levels (modules, classes, methods) and works for both functional and non-functional requirements. Unlike query reformulation, our method shrinks queries, addressing limitations in existing techniques and aligning with experts' suggestions of incorporating better contexts, leveraging structures, and creating a natural language model. Our proposed ENSEMBLE approach shows promising precision rates in real-world usage, outperforming closely related approaches. In ideal cases, our method demonstrates precision, recall, and F1 score of Modules, Classes, and Methods of all 20 features on Project 1 and Project 2, our proposed ENSEMBLE approach shows better performance in 14 out of 18 cases. Notably, our technique processes only texts and static code parts without the need for compiling or building change reversions or extracting input data from third-party systems. Human experts consider it lightweight. We have rigorously analyzed the possible causes of the poor results and found some interesting findings, such as semantic inconsistencies in the implemented code of an issue, which will help both practitioners and researchers focus on this direction to resolve the concerns.

the rest of the paper is organized as follows. Section 2 provides background about the construction of features and architecture. Section 3 provides related work on our proposed methods. Section 4 presents the dataset and its description. Section 5 presents methodology that means the approaches for feature-to-code component mapping. Section 6 describes the performance measurement and experimental outcomes. Section 7 discusses the conclusion and discussion.

1.2 Motivation

Automated software development and maintenance aims to reduce manual effort and improve the efficiency of software development processes. By automating repetitive and time-consuming tasks, it can help overcome the limitations of manual processes. Manual processes are almost fully dependent on humans these processes are prone to human error, which can lead to bugs, inconsistencies, and other issues in software engineering. Automated process tools follow predefined rules and guidelines, these tools ensure a higher level of accuracy and reduce the risk of mistakes over manual process tools.

Moreover, there are a lot of approaches used in our related work. These have some limitations such as The Vector Space Model (VSM) approach has some limitations as when there are no linear relationships between software and code components then this

method fails to produce proper output. VSMs can become computationally expensive and memory-intensive when dealing with large datasets[14]. In the Latent Semantic Indexing (LSI) appeoach, if there is a significant amount of noise or outliers in the data, it can negatively impact the quality of the linking results. LSI might identify a code component as relevant based on its similarity to certain terms but fail to recognize its true relevance in the software feature context. FSECAM has limitations like trace overhead and dependency on test cases. Besides that, some are lacking like trace overhead and dependency on test cases. The component Dependency Graph (CDG) approach has some limitations such as incomplete dependency analysis, and dynamic dependencies. The ranking of the Components approach has some limitations such as a lack of consideration for external dependencies, and incomplete capture of dynamic behavior. The combination of Dynamic Analysis and Static Analysis approaches have some limitations such as incomplete coverage which means that the test suite or usage scenarios do not encompass all possible scenarios, potentially leading to missed feature-specific computational units. Increasing scenario coverage and gathering feedback from users can help mitigate this limitation. Besides that, some are lacking like scalability, precision, and false positives, dynamic and static analysis limitations. Traditional Heuristics have some limitations such as reliance on keywords and patterns, lack of flexibility, and sensitivity to noise.

Besides that, Regular activities in software development maintenance increase day by day in the world. Another thing that many researchers already researched this topic but this work has low accuracy and high cost. on the other hand still space for performance improvement as well as an increase in explainable pro parties. Moreover, creating the link between software features to code components is very useful for a software developer. It helps a software developer find feature locations in the code base. Finally, create a more accurate link between software features to code components.

We want to make it more efficient for real-world usage.

1.3 Overall contributions of our works

Overall contributions of our works are as follows:

- We are developing an automated technique for software features to code components.
- We explored outcomes and analyzed the challenges of requirements.
- we improved the accuracy, recall, and precision of creating link software features to code components.
- We compare existing techniques and our proposed techniques (ENSEMBLE) and show the results.
- Visualization of overall results is done through bar charts and box plots.

Chapter II

Background

2.1 Definition of Software Features, Requirements, and Construction

Software Feature: A software feature refers to a distinct and identifiable capability or functionality of a software system. It represents a specific task or behavior that the software can perform[15].
For example, features could include functionalists like spell-check, formatting options, document collaboration, or the ability to insert images.

Software Requirement: A software requirement is a documented description of what a software system or application should do or how it should behave. It represents a specific need, constraint, or expectation that the software must meet[16].
For example, a functional requirement for an e-commerce website could be the ability for users to add products to a shopping cart, while a nonfunctional requirement could be that the website should load within three seconds for optimal user experience.

Software Construction: Software construction refers to the process of translating software requirements. It designs specifications into a working software product. It involves writing code, integrating components, and building the software system. Software construction encompasses activities like coding, debugging, unit testing, and integration testing[17].
In short, software features and requirements are closely linked in the software development process. Features describe the desired functionality, while requirements specify the overall expectations. Construction involves implementing the required features to meet the defined requirements and build the software system. Let's consider a real-world example of a login feature in a Java application using Hibernate for database interaction. Here's
an example implementation using Hibernate; [18]
Package: ***com.example.webapp***
Module: ***User.java***
Class: ***User***
Method: ***login(username, password)***

2.2 Code component

A code component refers to a self-contained and reusable unit of code that performs a specific function or represents a specific part of a software system. Software codebase is constructed with various abstraction levels of components such as modules, packages, classes, and methods these are hierarchically contained in a class, classes are contained within packages and packages are in the module. Requirements are implemented with them.
Code components are designed to be modular and independent, allowing developers to create software systems by combining and reusing these components. Each code component typically focuses on a specific task or feature, making it easier to understand, maintain, and modify the software. When linking software features to code components using automated techniques, the goal is to establish a connection between the desired functionality described by the software features and the specific code components responsible for implementing that functionality[19]. This process involves analyzing the requirements, understanding the code base, and identifying the code components that correspond to the features.

Module: In Java, modules are a way to organize and encapsulate code. Using Hibernate, we need to import the necessary modules into our project[20]. Here's an example of how we might import the Hibernate modules in a Java application:
import org.hibernate.Session;
import org.hibernate.SessionFactory;

Packages and Classes: Packages are used to organize related classes, and classes are the building blocks of our code[20]. In Hibernate, we typically create a package for our domain entities and another package for data access. Now create a simple example with a User entity and a User_A class:
package com.example.domain;.
public class User {

}
package com.example.dao;.
public class User A {

}

Methods: Methods contain the logic and operations that are performed on objects[21]. In our User_A class, we can define methods to interact with the User entity, such as saving a new user or retrieving existing users. Here's an example:

public void save(User user) {

}

public User getById(Long id) {

}

Overall, this example demonstrates the hierarchical containment of components in a software codebase. Modules contain packages, packages contain classes, and classes contain methods.

2.3 Linking steps

Here discussed the Linking process in our proposed method. The process is shown in Fig. 2.1.

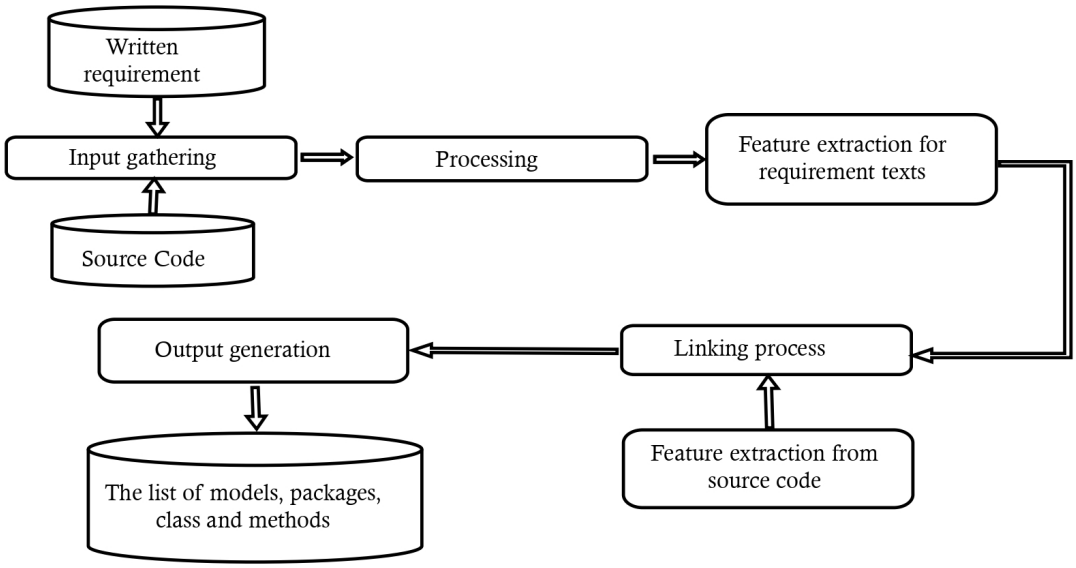


Figure 2.1: Linking Process figure.

Input Gathering: Collect the written requirements, including their descriptions and any associated documentation or metadata. Also, gather the source code of the software system.

Preprocessing: Clean and preprocess the feature descriptions and codebase to remove irrelevant information, standardize text format, and handle any noise or inconsistencies[22].

Feature Extraction for Requirement Texts: Analyze the feature descriptions using natural language processing (NLP) techniques. Extract key terms, identify relevant keywords, and build feature descriptors[23].

Feature extraction from source code: Analyze the codebase using static code analysis, parsing, or other techniques. Extract code elements such as modules, packages, classes, and methods. Identify their properties, relationships, and dependencies[24].

Linking Process: Apply the automated technique to link features to code components. This can involve matching feature descriptors with code element names and analyzing code annotations or comments[25].

Output Generations: Present the results of the automated linking process in a suitable format. This can include generating reports, visualizations, or structured data that represent the relationships between features and code components. Then finally we can get the list of associated the relevant requirement classes, modules, packages, and methods.

2.4 Applications

An automated technique for linking software features to code components is a process that analyzes feature descriptions and the codebase to establish connections. It is manipulating methods such as static code analysis, natural language processing, and machine learning. The technique maps features to relevant modules, classes, methods, functions, and packages within the code. It aims to provide better traceability and understanding of feature implementation in the software system. By automating this process, developers can quickly search the relevant code components for a given requirement and modify the codebase. The technique enhances code organization and enables effective communication between requirements and code components. It improves software documentation and facilitates collaboration among development teams. It helps in identifying the impact of feature changes on code components and ensures consistency between features and their corresponding code. The technique streamlines the software development lifecycle and enables faster feature implementation. It promotes code reusability and enhances system maintainability. It assists in identifying dependencies and potential code conflicts related to specific features. The technique reduces manual effort in tracing and managing feature-code associations. It enhances the overall software quality and reduces the risk of introducing bugs during feature implementation. It supports effective project management by providing insights into feature implementation progress. The technique contributes to better software documentation, making it easier to onboard new team members. It improves the efficiency of code maintenance, bug fixing, and system updates. Overall, the automated technique for linking software features to code components optimizes the software development process, enhancing productivity. Good requirements and API documentation are important for test case design refactoring, and third-party code inspection[26] [27].

2.5 Linking Process:

In this paper, we present a technique that utilizes theme analysis, a robust Natural Language Processing (NLP) method, to derive the feature-component map. Within this approach, theme analysis strategically intersects components across various scenarios, extracting specific components for individual features and identifying jointly and distinctly required components for a set of features. It is essential to note the close relationship between theme analysis, semantic meaning [21], and formal concept analysis[17], commonly employed in existing studies. The ensuing discussions offer distinctive insights into their interconnections.

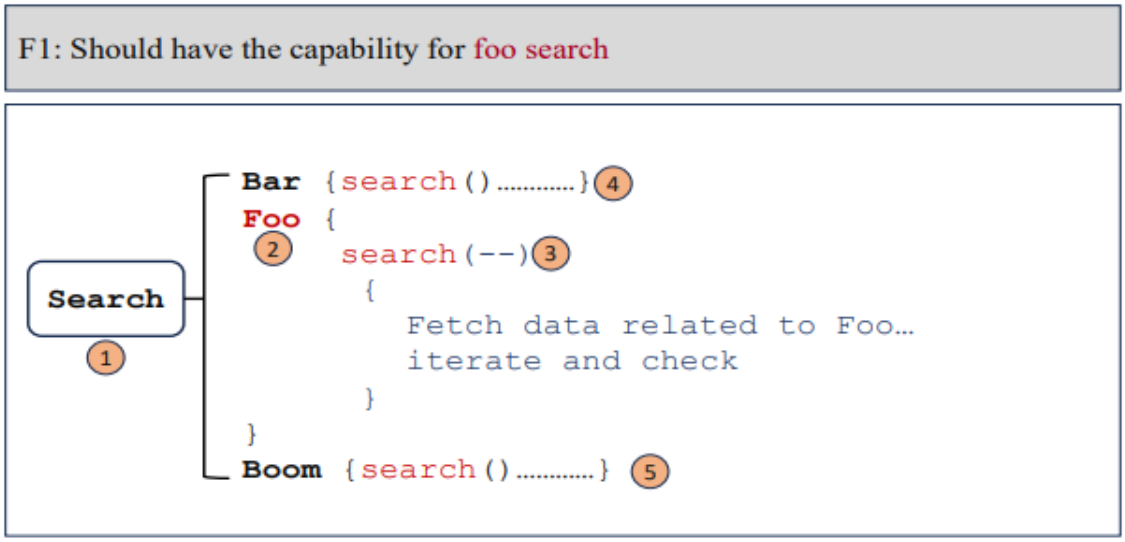


Figure 2.2: Linking Software Feature to Code Components

Semantic meaning: The human mental model of relevance pertains to the consideration of natural language meaning within software artifacts [8]. In a broader context, it elucidates the structuring of semantic meaning in a feature/requirement description, detailing the journey from conceptualization to implementation and expression in the commit message. This signifies the typical interpretation of a feature within a domain context.

Concept: In existing studies, the *concept* is typically viewed as the customary representation of functionality or concern in natural language [28]. Examining ”F1” in Figure 5.3 illustrates this concept, revealing two components: one pertains to possessing a particular capability that means - *Should have the capability*, while the other involves a search functionality related to *-foo search*.

Theme: According to our proposal, we hypothesize that focusing on the main theme of *-foo search* within F1 can significantly improve the identification process. This approach aims to eliminate many noisy and irrelevant components from the process, as opposed to including all tokens, which may introduce irrelevant elements such as the *-capability*

word alongside *-foo search*. Extracting such a theme, however, poses a challenge.

Chapter III

Related Work

3.1 Introduction

In recent years, there were published different works based on Automated Softwaring Engineering. Some of them are briefly discussed in this chapter.

3.2 FSECAM: A Contextual Thematic Approach for Linking Feature to Muilti-level Software Architectural Components

The FSECAM (A Contextual Thematic Approach for Linking Feature to Muilti-level Software Architectural Components) approach adopts a contextual thematic strategy, extracting pertinent theme properties from feature/requirement documents to tackle challenges in linking features to code components. Through rigorous experimentation on two proprietary projects, FSECAM demonstrates notable enhancements in connecting features to architectural abstractions (modules, classes, methods) and commits to issues. The method significantly improves precision rates and F1 scores by over 50% for proprietary projects compared to baseline techniques. Notably, FSECAM distinguishes itself by offering better explainability than existing Deep Learning approaches. Its static data processing eliminates the need for extensive sample labeling, streamlining model customization. The culmination of these efforts results in the development of the FSECAM tool, which seamlessly integrates into development processes, providing immediate and cost-effective feature-to-code linking capabilities.

3.3 Traceability Transformed: Generating More Accurate Links with Pre-Trained

There are a lot of tools to create links software features to code components such as Vector Space Model (VSM), Latent Dirichlet Allocation (LDA), AI swarm techniques so on. However this effectiveness has been restricted by the availability of labeled data and efficiency at runtime. To avoid this problem propose a novel framework called Trace BERT (T-BERT) to generate trace links between source code and natural language artifacts. Here the different architectures are TWIN, SIAMESE, and SINGLE. Here Masked Language Modeling (MLM), Pooling technique, Online Negative Sampling (ONS), and Replaced Token Detection techniques are used. Experiment results showed that SIAMESE architecture achieved the best accuracy overall. However, this approach has some limitations such as applying only to Python implementation, Only OSS Project, which may not be

enough to generate a generalized conclusion, and sometimes creating links between code and documentation and then consuming more time[28].

3.4 An Information Retrieval Approach to Concept Location in Source Code

It is very difficult for a programmer to identify the location where it should be changed. When the system is very large and complex then it is almost impossible to find out the concept location. This paper works on concept location using an advanced information retrieval method, Latent Semantic Indexing (LSI) method, and Single Value Decomposition (SVD). These methods create traceability links between external documentation and the source code and provide all that is needed for concept location. The proposed method works on the basis of two types of queries one is an automatically generated query and another is user user-based query. Finally, the system returns the document from the software according to the queries. These generated documents help to find out the concept location. However, this approach some lakes such as In some complex cases fails to create a link between external documents and source code, It fails to define several query templates based on the type of shared concept concerning user queries[29].

3.5 An Approach for Mapping Features to Code Based on Static and Dynamic Analysis.

The paper's motivation is to improve the process of mapping features to code in software development. This is an important task because it allows developers to understand which parts of the code are responsible for specific features, making it easier to maintain and evolve the software. There are many technologies used here like CheckCode, BIT framework, and Structural Analysis for Java (SA4J). For the result, here used two different kinds of matrix which are Two Way Impact (TWI) and Weighted Two Way Impact (WTWI). Weighted Two Way Impact (WTWI) improves the results of TWI by considering information from the system architecture. In the future, to apply thresholds to determine automatically then consider classes to be feature relevant. Without further analysis and comparison to other feature location techniques, it is difficult to fully assess the effectiveness of the proposed metrics[30].

3.6 Locating Features in Source Code

The paper aims to provide a process and techniques that automate the identification of code components responsible for implementing a specific set of features. By combining static and dynamic analyses and utilizing concept analysis. It is to improve program understanding. However, the paper focuses on the use of formal concept analysis (FCA) and concept lattices as the underlying theoretical framework for feature location in source code. FCA is a mathematical theory that provides a way to analyze and understand complex data structures, such as software systems. The technique involves creating scenarios, extracting the static dependency graph, conducting dynamic analysis, interpreting the concept lattice, and performing static dependency analysis. Future work may involve conducting additional experiments or case studies to further validate the effectiveness and

applicability of the proposed method. Doing this work there are some difficulties like Data availability, Complexity of the problem, Reproducibility, Evaluation metrics, Generalization, and applicability[31].

3.7 Relink: Recovering Links between Bugs and Changes

In this paper, we can learn about the discussion between bugs and committed changes which plays an important role in Software maintenance to measure the quality and predict defects. Bug tracking is the process of logging and monitoring bugs or errors during software testing. It is also referred to as defect tracking or issue tracking. Mainly A software Bug occurs when an application or program doesn't work the way it is designed to function. There are many technologies used here like BugZilla, LINKSTER, and ReLink. The Bug tracking tools in the future will offer more customizable features and powerful integration with other 14 applications, where required. Doing this work there are some difficulties like improper bug logging process, use of different bug tracking templates, and no control over the test environment[32].

3.8 Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code

In this paper, learn about the problem of concept location in source code by presenting an approach that combines Formal Concept Analysis (FCA) and Latent Semantic Indexing (LSI). Latent semantic indexing (LSI) is an indexing and retrieval method that uses a mathematical technique called singular value decomposition (SVD) to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text. This approach produces a ranked list of search results. Given the ranked list of source code elements, the proposed approach selects the most relevant attributes from these documents. Reducing the programmer's efforts. The Formal Concept Analysis (FCA) method is a mathematical technique used for analyzing data and establishing relationships based on formal concepts. This approach also has some limitations such as FCA being well-suited for categorical data where attributes have discrete values, when it comes to certain features or components that may have continuous or numeric attributes then FCA's applicability becomes limited when dealing with such data types[33].

Chapter IV

Dataset

4.1 Introduction

Our dataset for performance analysis and evaluation comprises two closed-source projects, each belonging to different domains. The first project deals with a data repository, while the second project is a framework extensively utilized by scientific communities. Both projects are cloud-based systems, actively maintained, and undergoing continuous development. Table 4.1 provides a summary of the statistics for these projects, including a total of 340 classes, 122 modules, and 3126 methods.

4.2 Deccription

For our experimental dataset, we randomly selected 10 features from each project to form the test set. The development team associated components with these features, resulting in 37 classes, 75 modules, and 172 methods, as detailed in Table 4.2. However, it's worth noting that developers were unable to identify associated classes for eight features from Project 2 (P2), denoted by "NA" in various performance tables later in our analysis. The preparation of the test set involved approximately 80 hours per person from the development team, highlighting the significant manual effort required to locate the components of the selected features. In the subsequent sections, we delve into the findings of our analyses and assess the performance of our proposed techniques using this dataset.

Table 4.1: Experimental Dataset.

Project	Domain	Class	Module	Method
Project 1 (P1)	Data Repository	120	40	1000
Project 2 (P2)	Scientific workflow system	222	82	2126

Table 4.2: Test Dataset.

Project	Class	Module	Method
P1	26	29	34
P2	11	46	138

Chapter V

Methodology

5.1 Introduction

Some processes following our proposed method witches discussed below. The process is shown in Figure 5.2 and 5.1

5.2 Bootstrapping and Aggregating

Discussing an essential aspect of our proposed method, we focus on Bootstrapping and Aggregating. Numerous models are presented, each yielding distinct outputs. In this context, Bootstrapping functions as a selection mechanism from the output data, while Aggregating involves the averaging of the selected outputs.

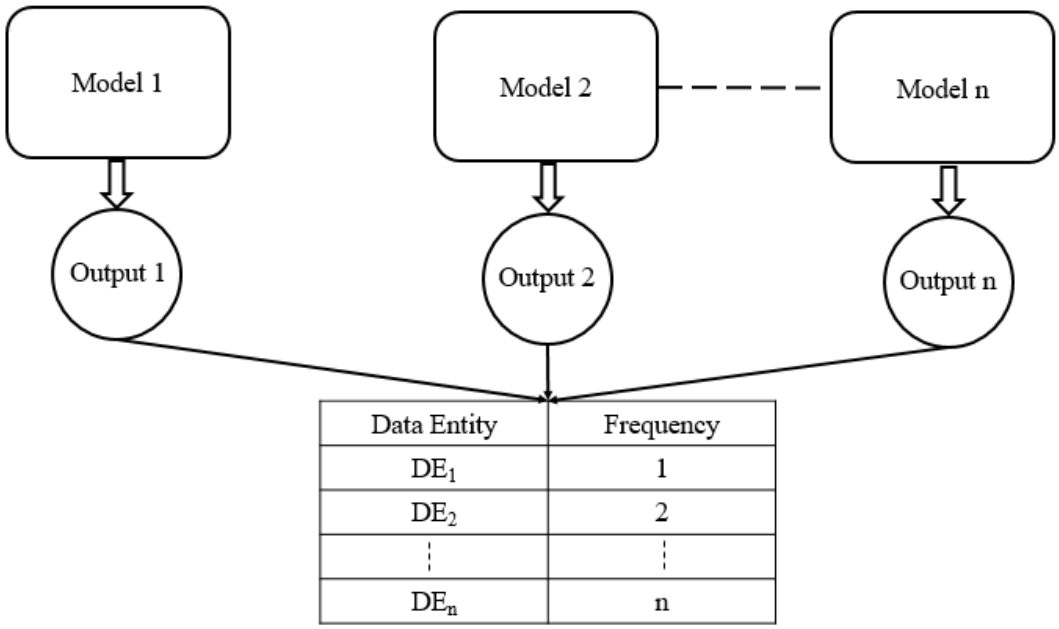


Figure 5.1: Aggregating

5.3 Score Calculation

In this section of our work, we discuss the mechanism of the score calculation for the ENSEMBLE Approach.

$$ScorePercentage = \frac{Frequency\_of\_the\_output\_entities}{Total\_number\_of\_methods} \tag{5.1}$$

In our proposed Method we use three approaches these are FSECAM, VSM, and LSI. When we calculate the frequency of the output entity as 1,2 and 3 then our calculation is like this...

$$ScorePercentagefor1 = \frac{1}{3} \times 100 = 33\% \tag{5.2}$$

$$ScorePercentagefor2 = \frac{2}{3} \times 100 = 67\% \tag{5.3}$$

$$ScorePercentagefor3 = \frac{3}{3} \times 100 = 100\% \tag{5.4}$$

5.4 Working Process

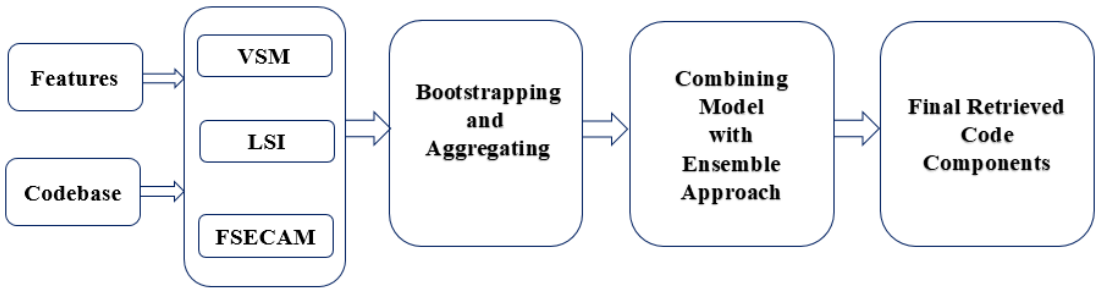


Figure 5.2: Proposed Method

VSM Approach: The Vector Space Model (VSM) is widely used in automated software tasks like code similarity analysis, search, and recommendation. It operates on collections of text documents, such as code snippets or source code files, representing each document as a numerical vector in a high-dimensional space. The model involves steps like Text Preprocessing, Term Frequency (TF) Calculation, Inverse Document Frequency (IDF) Calculation, TF-IDF Calculation, and Vector Representation. While VSM offers flexibility, it has limitations, including challenges with vocabulary size, sparse vectors, and the disregard for word order in its analysis.

LSI Approach: The Latent Semantic Indexing (LSI) model, utilized in information retrieval and natural language processing, can establish connections between Software Features and Code Components. It processes document collections by first structuring

them to represent feature-code relationships. Tokenization and conversion into a term-document matrix follow, with subsequent Singular Value Decomposition (SVD) to obtain matrices U, σ, and V. Dimension reduction mitigates noise, representing documents in a numerical space. LSI's advantages include noise and dimension reduction. However, limitations include contextual understanding, vocabulary constraints, sensitivity to preprocessing, and the need for sufficient training data.[34]

FSECAM Approach: A Contextual Thematic Approach for Linking Feature to Muiltilevel Architectural Components(FSECAM) approach employs a contextual thematic strategy, extracting relevant theme properties from feature/requirement documents to address challenges in linking features to code components. Through extensive experimentation on two proprietary projects, FSECAM demonstrates substantial improvements in connecting features to architectural abstractions (modules, classes, methods) and commits to issues. It notably enhances precision rates and F1 scores by over 50% for proprietary projects compared to baseline techniques. FSECAM stands out by offering superior explainability compared to existing Deep Learning approaches. Its static data processing eliminates the need for extensive sample labeling, streamlining model customization. The culmination of these efforts results in the development of the FSECAM tool, seamlessly integrating into development processes and providing immediate and cost-effective feature-to-code linking capabilities.

Step 1: The input of our proposed approach are different kinds of software features and texts. Besides, we also need to input source code components and their relationships.

Step 2: In this step, there are three types of approaches LSI, VSM and FSECAM works on the input data, and after processing the data these approaches produce output. The output of these three methods is different kinds of code components such as classes, methods, and modules.

Step 3: This model is trained on this type of data. The model creates a decision tree according to which code component will be relevant or not. When it is relevant then it's binary representation 1 whereas if it is not, then it's a binary representation 0. Finally, all the binary output representations will be countable. That's the overall process that will be done by Bootstrapping and aggregating. Texts Feature LSI, VSM, and FSECAM Voting, Ranking from Bootstrapping and Aggregating combine Model with ENSEMBLE Method. Final Model

Step 4: In this step which model produces 0 then these models will be rejected. And which models produce 1 then we will receive these methods. Finally, we combine our received methods. This process is done by the Ensemble method. The ENSEMBLE technique will work in our approach discussed below. Here the outcome of every approach is code components such as classes, methods, modules, and classes. Then all the outcomes from all the approaches are aggregated for our approach. Then a voting mechanism is applied. For example, every method is involved in the voting mechanism and their voting is measured when 33%,67%, or 100% we will measure this method and combine them.

That's called the ENSEMBLE approach. The produced outcome of every approach class, method, and module will be voted on. This voting mechanism will be held according to which classes will be repeated how many times in every approach. In the same way methods and modules will be voted. And will select these classes, methods, and modules which will be voted to count at the top position.

Step:5 Finally, all the selected code components will finally output as our proposed final model.

ENSEMBLE Approach: The ENSEMBLE approach is a powerful technique in machine learning that combines multiple base models to achieve improved accuracy, robustness, and reduced overfitting. It has proven to be successful in various real-world applications and is widely used in both traditional machine learning and deep learning. Ensemble methods are widely used in various machine-learning tasks, including classification, regression, and clustering, and they have demonstrated their effectiveness in numerous real-world applications.

Chapter VI

Performance Measurement

6.1 Introduction

The proposed method will be evaluated using the common metrics which are, accuracy, precision, recall, and F1 score. To evaluate the performance, we have used Precision, Recall, and F1-score, these are formulated as follows:

Precision: Precision refers to the proportion of correctly predicted positive observation to the total predicted positive observation. When the precision is high, the number of false positive predictions is low.

$$Precision = \frac{TP}{TP + FP} \tag{6.1}$$

Recall: he recall is known as sensitivity or true positive rate. Recall is the ratio of correctly predicted positive observations to all observations in actual class. When the recall is high, the number of false negative predictions is low.

$$Recall = \frac{TP}{TP + FN} \tag{6.2}$$

F1 Score: F1 score is a metric that takes into account both precision and recall. It is the weighted average of precision and recall. F1 score provides a combined effect of precision and recall. It is more useful than accuracy especially if the dataset is imbalanced. The target of this paper is to achieve a better F1 score.

$$F1Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6.3}$$

Where,

- True Positive (TP): True Positive is the number of positive cases that are labeled correctly.
- True Negative (TN): True Negative is the number of negative cases that are labeled correctly.
- False Positive (FP): False Positive is the number of positive cases that are labeled falsely.
- False Negative (FN): False Negative is the number of negative cases that are labeled falsely.

6.2 Performance Measurement of comparison on table :

In this section of our study, we present individual measurements for Project 1 and Project 2, displaying precision, recall, and F1 scores for each feature. Tables 6.2 and 6.4 showcase the average values for all features (Modules, Classes, and Methods) concerning precision, recall, and F1 score for voting scores 33%, 67%, and 100% in Project 1 and 2. In most cases, a voting score of 100% proves to be more accurate, except for Recall of Methods in Voting Score 1 in Project 1 and Recall of Classes in Voting Score 67% in Project 2, which outperform others.
Despite the varied performance, it is evident that overall results across all approaches are not promising. The potential reasons include deficiencies in project design, code quality, and message writing quality, falling short of established standards. The dataset itself may contain errors introduced by developers during creation. Generally, precision is inadequate, indicating a high association with irrelevant components, while recall remains relatively good. Notably, the FSECAM approach demonstrates the best performance across all levels Modules, Classes, and Methods compared to LSI and VSM approaches.

Module Comparison: In Table 6.5, the comparison of modules for four approaches (ENSEMBLE, FSECAM, LSI, and VSM) across all 20 features, including Project 1 and Project 2, is presented.Average value of Precision scores for the four approaches are as follows: 37.44% (ENSEMBLE), 22.51% (FSECAM), 8.75% (LSI), and 11.31% (VSM). Notably, the proposed ENSEMBLE approach demonstrates superior precision, outperforming the other three approaches. The ENSEMBLE approach exhibits a precision value 14.63 higher than FSECAM.

Average value of Recall and F1 Score values for the four approaches are 79.88%, 65.59%, 63.27%, and 68.26%, and 42.91%, 25.56%, 14.55%, and 18.23%, respectively. In terms of recall and F1 score, the ENSEMBLE approach outshines the other three approaches. Specifically, ENSEMBLE surpasses LSI by 11.62 in precision and outperforms itself by 17.35 in F1 scores. Across all metrics, the ENSEMBLE approach consistently outperforms the other three approaches.

Class Comparison: Table 6.6 presents a comparison of classes for the same four approaches across all 20 features.Average Precision scores for ENSEMBLE, FSECAM, LSI, and VSM are 12.72%, 4.83%, 2.82%, and 3.61%, respectively. Once again, the proposed ENSEMBLE approach exhibits superior precision, with a value 7.83 higher than FSECAM.

Average Recall and F1 Score values for the four approaches are 70.15%, 59.0%, 54.26%, 51.33%, and 17.99, 8.03, 5.33, and 6.59, respectively. In terms of recall and F1 score, the ENSEMBLE approach consistently outperforms the other three approaches. ENSEMBLE outperforms FSECAM by 11.15 in precision and 9.96 in F1 scores. Across all metrics, the ENSEMBLE approach demonstrates superior performance in class evaluation.

Method Comparison: Table 6.7 provides a comparison of methods for the same four

approaches across all 20 features. Average Precision scores for ENSEMBLE, FSECAM, LSI, and VSM are 11.50%, 10.80%, 0.85%, and 1.17%, respectively. Once again, the proposed ENSEMBLE approach showcases better precision than the other three approaches, with a value 0.70 higher than FSECAM.

Average Recall and F1 Score values for the four approaches are 22.38%, 33.68%, 35.08%, 34.22%, and 9.47%, 8.71%, 1.56%, and 2.18%, respectively. However, in the case of recall and F1 scores, the ENSEMBLE approach shows worse performance than the other three approaches. LSI outperforms ENSEMBLE by 11.84 in recall, while ENSEMBLE surpasses itself by 0.76 in F1 scores. Nevertheless, in two out of three cases, the proposed ENSEMBLE approach outperforms the other three approaches.

In summary, the ENSEMBLE approach consistently demonstrates superior performance across precision, recall, and F1 scores, outshining the other three approaches in most cases.

6.3 Performance Measurement of comparison on Bar Plot :

In this discussion, we assess the performance of four techniques: FSECAM, VSM, LSI, and our proposed ENSEMBLE approach, considering them as superior techniques for all data samples. Figure 6.1 illustrates a significant performance enhancement when associating software features with three fundamental architectural abstractions of code components: modules, classes, and methods, as indicated by the precision, recall, and F1 score for each feature in Project 1 and Project 2. Finally, we compare output values using a 6-bar plot for both projects.

Figure 6.1(a) represents the precision of Modules, Classes, and Methods for Project 1. In all cases, our proposed ENSEMBLE approach outperforms the other three approaches. For Modules, the ENSEMBLE approach achieves an average value is 43.5%, while the closest value from FSECAM is 20.40%. In the case of Classes and Methods, the values are 15.10%, 5.70%, and 15.80%, 15.30% respectively.

Figure 6.1(b) demonstrates that in 2 out of 3 cases for Modules, Classes, and Methods, our ENSEMBLE approach performs better than the other three approaches. The average recall of the ENSEMBLE approach is 88.80%, outperforming VSM with a value of 75.2%. However, in the case of Methods, ENSEMBLE fails to produce a value higher than the other three approaches.

Similarly, Figure 6.1(c) shows 2 out of 3 cases where our proposed ENSEMBLE approach performs better than the other three approaches in terms of F1 score. For Modules, the ENSEMBLE approach achieves average value is 47.60%, while the nearest value from VSM is 26.10%. However, in the case of Methods, ENSEMBLE fails to produce a value higher than the other three approaches.

In Figure 6.1(d), precision for Modules, Classes, and Methods in Project 1 is depicted,

while Figures 6.1(e) and 6.1(f) showcase recall and F1 scores for the same project. In 6.1(d), our proposed ENSEMBLE approach outperforms the other three approaches in 2 out of 3 cases. For Modules, ENSEMBLE achieves average precision value is 37.70%, surpassing the nearest value from FSECAM at 25.10%. However, in the case of Classes, the ENSEMBLE value is 3.80, and the closest value from FSECAM is 1.80. Similarly, for Methods, ENSEMBLE is at 5.43%, whereas FSECAM is at 7.60%, albeit failing to surpass the other three approaches.

Moving to 6.1(e), it again demonstrates 2 out of 3 cases where our ENSEMBLE approach excels. For Modules, ENSEMBLE achieves average value recall is 70.50%, surpassing the nearest value from FSECAM at 53.30%. In the case of Classes, ENSEMBLE is at 29.20 while the closest value from FSECAM is 20.80%. However, in the case of Methods, ENSEMBLE falls short at 17.40 compared to FSECAM's 29.10.

In 6.1(f), all cases for Modules, Classes, and Methods show our proposed ENSEMBLE approach performing better than the other three approaches. For Modules, ENSEMBLE attains an average precision of 37.70%, outperforming FSECAM's nearest value of 25.10%. Regarding Classes and Methods, the values are 3.40%, 1.80%, and 6.20%, 2.90%, respectively.

In summary, for all 14 cases of 18 cases, our proposed ENSEMBLE approach produce better than the other three approaches. However, across these figures, our ENSEMBLE approach consistently outperforms the other three techniques in a majority of cases. The precision, recall, and F1 score for various features exhibit superior performance, although some areas, particularly in Methods, indicate room for improvement.

6.4 Performance Measurement of comparison on Box Plot:

In the context of performance evaluation, box plots emerge as powerful tools for visualizing the similarities and differences in distributions across various performance metrics. Figures 6.2 to 6.4 depict box plots, providing a comprehensive overview of the performance of different approaches within modules, classes, and methods. Across all cases, there are overlaps in precision among the four methods. However, ENSEMBLE stands out, as evidenced by significantly different precision plots in Figures 6.2(a), 6.3(a), and 6.4(a), particularly noticeable in the presence of outliers. ENSEMBLE's precision plots reveal outliers with high values, significantly exceeding other approaches. Furthermore, ENSEMBLE exhibits the highest median precision, implying a potentially higher central tendency. Notably, the ENSEMBLE also showcases the highest range, indicating variability in performance. In contrast, among the remaining methods, FSECAM consistently outperforms others in precision, while Ensemble surpasses FSECAM.

Moving on to recall, ENSEMBLE again demonstrates distinct plots in Figures 6.2(b), 6.3(b), and 6.4(b), characterized by outliers of high values. The median recall for ENSEMBLE is consistently the highest, suggesting a potentially higher central tendency. However, a nuanced observation in Figure 6.17 reveals that FSECAM outperforms EN-

SEMBLE in certain instances. Similar to precision, the Ensemble exhibits the highest range, indicating variability in performance. Despite FSECAM performing better than other methods in the recall, ENSEMBLE consistently outshines FSECAM.

Finally, for the F1 Score in Figures 6.2(c), 6.3(c), and 6.4(c), ENSEMBLE once again displays significantly different plots with outliers of high values. The median F1 Score for ENSEMBLE is consistently the highest, pointing to a potentially higher central tendency. Figure 6.21 introduces an interesting twist where FSECAM surpasses ENSEMBLE in terms of range, although Ensemble remains superior in central tendency. As before, the ENSEMBLE exhibits the highest range, suggesting variability in performance. In summary, while FSECAM performs better than other approaches in certain aspects, ENSEMBLE emerges as the overall superior method across various metrics and categories, outclassing FSECAM, LSI, and VSM.

Table 6.1: Individual Measurement For voting score 33%,67% and 100% on Project 1

Feature		Modules			Classes			Methods		
Name	Voting score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
23	33%	4.55	12.5	6.67	3.57	33.33	6.45	0.22	4.17	0.41
	67%	12.5	25.0	16.67	6.67	33.33	11.11	0.44	4.17	0.79
	100%	53.85	87.50	66.67	5.17	100.0	9.84	1.82	8.33	2.99
25	33%	5.0	20.0	8.0	6.25	33.33	10.53	NA	NA	NA
	67%	7.14	20.0	10.53	7.69	33.33	12.50	0.31	3.23	0.56
	100%	14.71	100.0	25.64	2.2	6.67	4.62	1.20	19.35	2.25
83	33%	1.89	25.0	3.51	2.04	50.0	3.92	NA	NA	NA
	67%	8.33	25.0	12.50	7.14	50.0	12.5	NA	NA	NA
	100%	21.05	100.0	34.78	3.51	100.0	6.78	4.46	63.64	8.33
159	33%	7.69	8.33	8.0	14.29	25.0	18.18	NA	NA	NA
	67%	14.29	16.67	15.38	4.0	25.0	6.90	0.47	13.33	0.91
	100%	35.71	83.33	50.0	5.26	100.0	10.0	0.51	13.33	0.97
401	33%	1.59	25.0	2.99	1.82	8.33	2.99	NA	NA	NA
	67%	8.33	25.0	12.50	4.55	8.33	5.88	NA	NA	NA
	100%	21.05	100.0	34.78	1.69	8.33	2.82	NA	NA	NA
F159	33%	4.55	50.0	8.33	3.03	33.33	5.56	0.27	50.0	0.53
	67%	12.50	50.0	20.0	9.09	33.33	14.29	1.25	50.0	2.44
	100%	11.76	100.0	21.05	5.56	100.0	10.53	NA	NA	NA
F212	33%	1.45	14.29	2.63	1.64	14.29	2.94	NA	NA	NA
	67%	7.14	14.29	9.52	3.33	14.29	5.41	1.0	15.38	1.88
	100%	50.0	100.0	66.67	16.67	100.0	28.57	1.12	15.38	2.09
F275	33%	3.23	50.0	6.06	1.20	50.0	2.35	NA	NA	NA
	67%	14.29	50.0	22.2	10.0	50.0	16.67	14.29	50.0	22.22
	100%	50.0	100.0	66.67	50.0	100.0	66.67	100.0	50.0	66.67
F291	33%	5.0	10.0	6.67	4.17	16.67	6.67	NA	NA	NA
	67%	16.67	30.0	21.43	3.30	16.67	5.13	0.28	16.67	0.54
	100%	58.33	70.0	63.64	7.84	66.67	14.04	1.73	16.67	2.53
F310	33%	NA	NA	NA	8.33	80.0	15.09	3.12	33.33	5.71
	67%	20.0	42.86	27.27	8.33	80.0	15.09	3.12	33.33	5.71
	100%	50.0	42.86	46.15	40.0	40.0	40.0	NA	NA	NA

Table 6.2: Averaging Measurement For voting score 33%,67% and 100% on Project 1

	Modules			Classes			Methods		
Voting score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
33%	3.88	23.90	5.87	4.63	34.43	7.47	1.20	29.17	0.53
67%	12.10	29.90	16.30	6.00	34.40	11.10	2.60	23.30	4.70
100%	43.477	88.37	47.61	15.13	79.26	21.23	15.83	26.67	12.26

Table 6.3: Individual Measurement For voting score 33%,67% and 100% on Project 2

Feature		Modules			Classes			Methods		
Name	Voting score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
19	33%	0.34	14.29	0.66	NA	NA	NA	0.6	15.62	1.15
	67%	3.45	14.29	5.56	NA	NA	NA	4.21	40.62	7.62
	100%	11.43	57.14	19.05	NA	NA	NA	2.72	15.62	4.63
21	33%	12.5	50.0	20.0	NA	NA	NA	NA	NA	NA
	67%	2.08	25.0	3.85	NA	NA	NA	NA	NA	NA
	100%	3.64	50.0	6.78	NA	NA	NA	1.05	33.33	2.04
27	33%	0.43	12.5	0.84	NA	NA	NA	NA	NA	NA
	67%	2.86	25.0	5.13	NA	NA	NA	0.81	6.67	1.45
	100%	21.21	87.5	34.15	NA	NA	NA	NA	NA	NA
28	33%	1.89	16.67	3.39	NA	NA	NA	0.19	7.69	0.36
	67%	50.0	66.67	57.14	NA	NA	NA	11.11	7.69	9.09
	100%	21.43	50.0	30.0	NA	NA	NA	NA	NA	NA
30	33%	1.92	25.0	3.57	NA	NA	NA	0.20	3.57	0.39
	67%	0.41	25.0	0.80	NA	NA	NA	7.56	46.43	13.0
	100%	8.7	50.0	14.81	NA	NA	NA	0.44	3.57	0.79
37	33%	2.38	25.0	4.35	NA	NA	NA	2.23	76.47	4.33
	67%	2.53	50.0	4.82	NA	NA	NA	1.79	11.76	3.10
	100%	100.0	75.0	85.71	NA	NA	NA	NA	NA	NA
F20	33%	0.41	8.33	0.79	7.14	25.0	11.11	NA	NA	NA
	67%	5.26	16.67	8.0	4.44	50.0	8.16	0.22	10.0	0.42
	100%	22.0	91.67	35.48	0.85	25.0	1.65	0.41	10.0	0.78
F29	33%	2.17	14.29	3.77	NA	NA	NA	NA	NA	NA
	67%	0.42	14.29	0.82	NA	NA	NA	NA	NA	NA
	100%	3.12	14.29	5.13	NA	NA	NA	6.25	16.67	9.09
F32	33%	2.33	9.09	3.70	2.04	16.67	3.64	NA	NA	NA
	67%	11.76	18.18	14.29	3.70	16.67	6.06	0.95	10.0	1.74
	100%	42.11	72.73	53.33	2.82	33.33	5.19	NA	NA	NA
F36	33%	0.39	33.33	0.77	NA	NA	NA	NA	NA	NA
	67%	3.45	33.33	6.25	NA	NA	NA	NA	NA	NA
	100%	42.86	100.0	60.0	NA	NA	NA	16.67	25.0	20.0

Table 6.4: Averaging Measurement For voting score 33%,67% and 100% on Project 2

	Modules			Classes			Methods		
Voting score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
33%	2.48	20.85	4.18	1.2	24.17	1.62	0.81	25.84	1.56
67%	7.83	28.84	5.50	1.4	26.67	2.78	3.81	19.02	5.55
100%	31.41	70.44	37.70	1.8	29.2	3.42	5.43	17.37	6.22

Table 6.5: Performance of Modules association with our proposed approach for Project 1 and 2

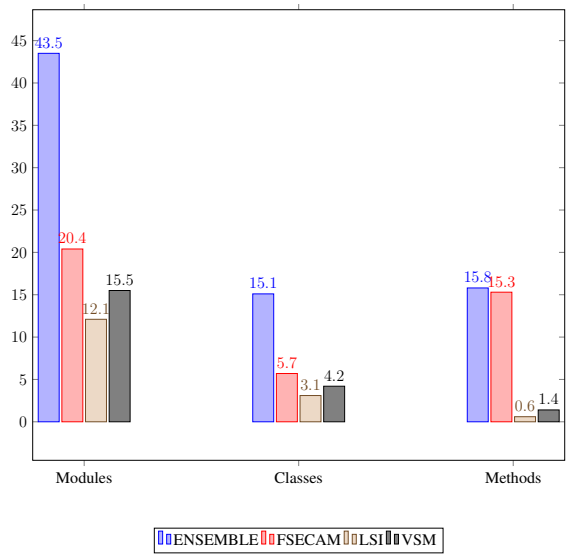
Feature	ENSEMBLE			FSECAM			LSI			VSM		
#FID	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
23	53.85	87.50	66.67	18.75	75.0	30.0	23.33	87.50	36.84	25.0	87.5	38.89
25	14.71	100.0	25.64	8.70	88.0	15.69	8.0	80.0	14.55	7.84	80.0	14.29
83	21.05	100.0	34.78	12.0	75.0	20.69	4.0	75.0	7.59	9.68	75.0	17.14
159	35.71	83.33	50.0	20.45	75.0	32.14	30.30	83.33	44.44	22.22	83.33	35.09
401	21.05	100.0	34.78	14.29	75.0	24.0	3.33	75.0	6.38	10.0	75.0	15.65
F159	11.76	100.0	21.05	5.0	50.0	9.09	2.44	50.0	4.65	4.0	50.0	7.41
F212	50.0	100.0	66.67	42.86	85.71	57.14	7.14	85.71	13.19	15.79	85.71	26.67
F275	50.0	100.0	66.67	9.09	50.0	15.38	2.86	50.0	5.41	12.50	50.0	20.0
F291	58.33	70.0	63.64	50.0	80.0	61.51	17.65	60.0	27.27	20.51	80.0	32.65
F310	50.0	42.86	46.15	23.08	42.86	30.0	21.43	85.71	34.29	27.27	85.71	41.38
19	11.43	57.14	19.05	0.85	42.86	1.66	5.56	42.86	9.84	6.38	42.86	11.11
21	3.64	50.0	6.78	0.99	25.0	1.90	1.52	25.0	2.86	1.87	50.0	3.60
27	21.21	87.5	34.15	7.07	87.5	13.08	2.30	75.0	4.46	6.60	87.50	12.28
28	21.43	50.0	30.0	87.78	83.33	41.67	6.94	83.33	12.82	11.11	33.33	16.67
30	8.7	50.0	14.81	0.39	25.0	0.77	1.51	25.0	2.20	0.37	25.0	0.74
37	100.0	75.0	85.71	60.0	75.0	66.67	2.22	50.0	4.26	2.70	75.0	5.22
F20	22.0	91.67	35.48	14.29	91.67	24.72	11.36	83.33	20.0	3.67	91.67	7.05
F29	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
F32	42.11	72.73	53.33	23.53	72.73	35.56	10.45	63.64	17.95	26.67	72.73	39.02
F36	42.86	100.0	60.0	28.57	66.67	40.0	3.92	66.67	7.41	0.73	66.67	1.44
Average	37.44	79.88	42.91	22.51	65.59	25.56	8.75	63.27	14.55	11.31	68.26	18.23

Table 6.6: Performance of Classes association with our proposed approach for Project 1 and 2

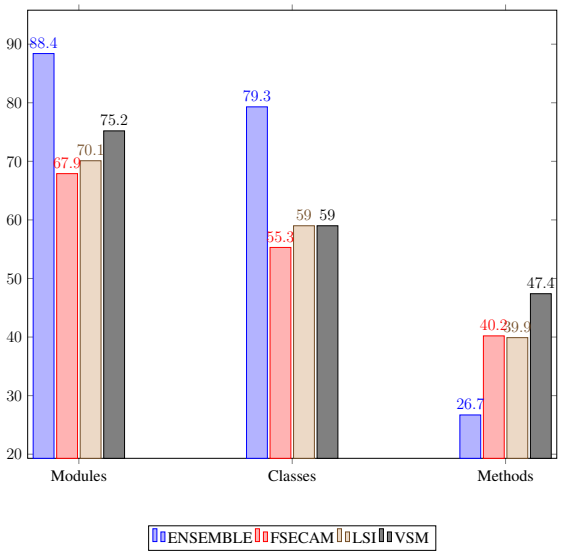
Feature	ENSEMBLE			FSECAM			LSI			VSM		
#FID	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
23	5.17	100.0	9.84	2.60	66.67	5.0	2.90	66.67	5.56	2.41	66.67	4.65
25	2.2	6.67	4.62	0.98	33.33	1.90	1.02	33.33	1.98	0.89	33.33	1.74
83	3.51	100.0	6.78	1.56	50.0	3.03	0.90	50.0	1.77	1.43	50.0	2.78
159	5.26	100.0	10.0	2.94	75.0	5.66	3.80	75.0	7.23	2.97	75.0	5.71
401	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
F159	5.56	100.0	10.53	3.64	66.67	6.90	2.11	66.67	4.08	3.12	66.67	5.97
F212	16.67	100.0	28.57	12.77	85.71	22.22	5.45	85.71	10.26	6.90	85.71	12.77
F275	50.0	100.0	66.67	9.09	50.0	15.38	1.10	50.0	2.15	10.0	50.0	16.67
F291	7.84	66.67	14.04	5.08	50.0	9.23	3.41	50.0	6.38	3.23	50.0	6.06
F310	40.0	40.0	40.0	12.50	20.0	15.38	6.78	80.0	12.50	7.02	80.0	12.9
19	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
21	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
27	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
28	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
30	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
37	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
F20	0.85	25.0	1.65	0.76	25.0	1.48	NA	NA	NA	0.62	25.0	1.21
F29	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
F32	2.82	33.33	5.19	1.16	16.67	2.17	0.74	16.67	1.42	1.09	16.67	2.04
F36	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Average:	12.72	70.15	17.99	4.83	49.0	8.03	2.82	54.26	5.33	3.61	51.33	6.59

Table 6.7: Performance of Methods association with our proposed approach for Project 1 and 2

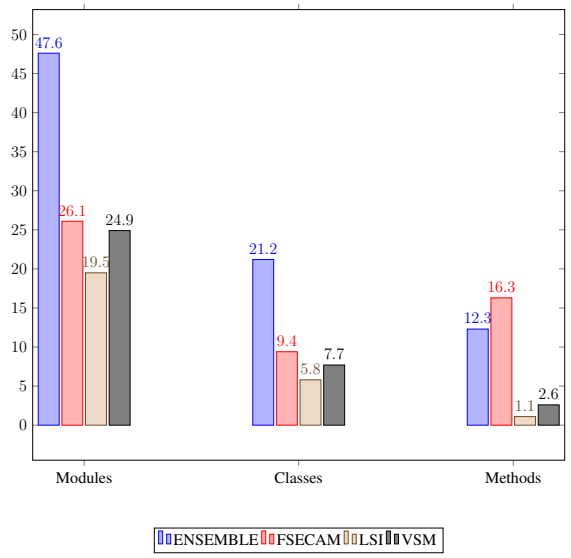
Feature	ENSEMBLE			FSECAM			LSI			VSM		
#FID	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
23	1.82	8.33	2.99	0.64	12.50	1.22	0.64	8.33	1.19	0.85	16.67	1.61
25	1.20	19.35	2.25	0.83	22.58	1.59	0.83	19.35	1.59	0.71	22.58	1.38
83	4.46	63.64	8.33	2.36	63.64	4.56	0.81	63.64	1.59	1.54	63.64	3.0
159	0.51	13.33	0.97	0.47	26.67	0.92	0.39	33.33	0.76	0.48	26.67	0.94
401	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
F159	NA	NA	NA	NA	NA	NA	0.20	50.0	0.39	0.81	100.0	1.61
F212	1.12	15.38	2.09	1.15	23.08	2.19	0.30	23.08	0.60	0.67	30.77	1.31
F275	100.0	50.0	66.67	100.0	100.0	100.0	0.34	100.0	0.67	5.13	100.0	9.76
F291	1.73	16.67	2.53	1.80	33.33	3.42	0.19	16.67	0.37	0.30	33.33	0.60
F310	NA	NA	NA	NA	NA	NA	1.25	44.44	2.43	1.69	33.33	3.21
19	2.72	15.62	4.63	1.75	71.88	3.43	5.06	56.25	9.28	1.45	15.62	2.66
21	1.05	33.33	2.04	0.53	33.33	1.05	0.26	33.33	0.52	0.43	33.33	0.85
27	NA	NA	NA	0.70	6.67	1.27	NA	NA	NA	0.60	6.67	1.1
28	NA	NA	NA	8.7	15.38	11.11	0.19	7.69	0.37	NA	NA	NA
30	0.44	3.57	0.79	0.35	3.57	0.65	1.85	53.57	3.58	3.35	50.0	6.28
37	NA	NA	NA	60.0	88.24	71.43	NA	NA	NA	0.97	11.76	1.79
F20	0.41	10.0	0.78	0.33	20.0	0.65	0.18	10.0	0.36	0.27	20.0	0.52
F29	6.25	16.67	9.09	0.36	16.67	0.70	0.24	16.67	0.47	0.22	16.67	0.43
F32	NA	NA	NA	0.78	10.0	1.45	NA	NA	NA	0.87	10.0	1.6
F36	16.67	25.0	20.0	2.86	25.0	5.13	0.41	25.0	0.82	0.26	25.0	0.51
Average	11.50	22.38	9.47	10.80	33.68	8.71	0.85	35.08	1.56	1.17	34.22	2.18



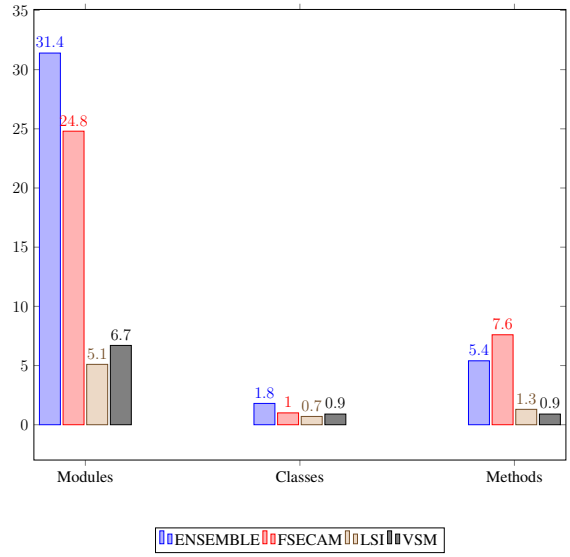
(a) Performance Precision on Project 1



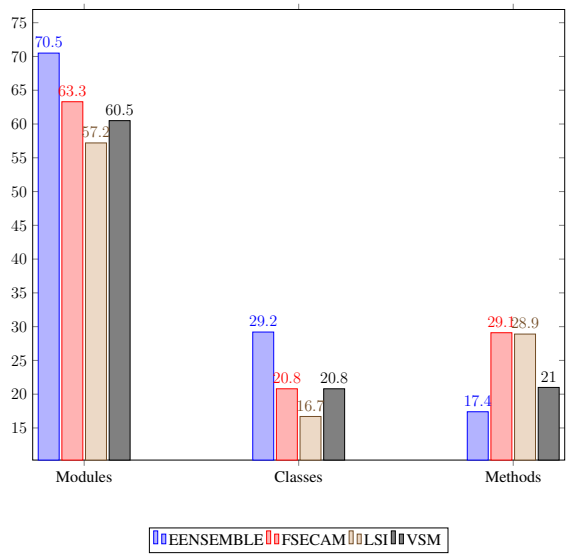
(b) Performance Recall on Project 1



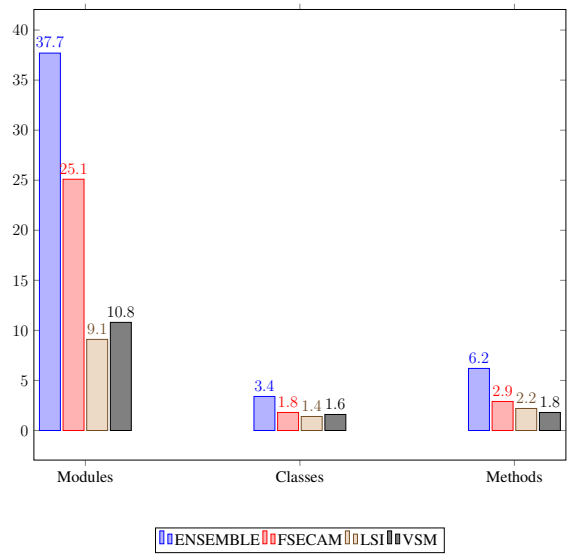
(c) Performance F1 Score on Project 1



(d) Performance Precision on Project 2

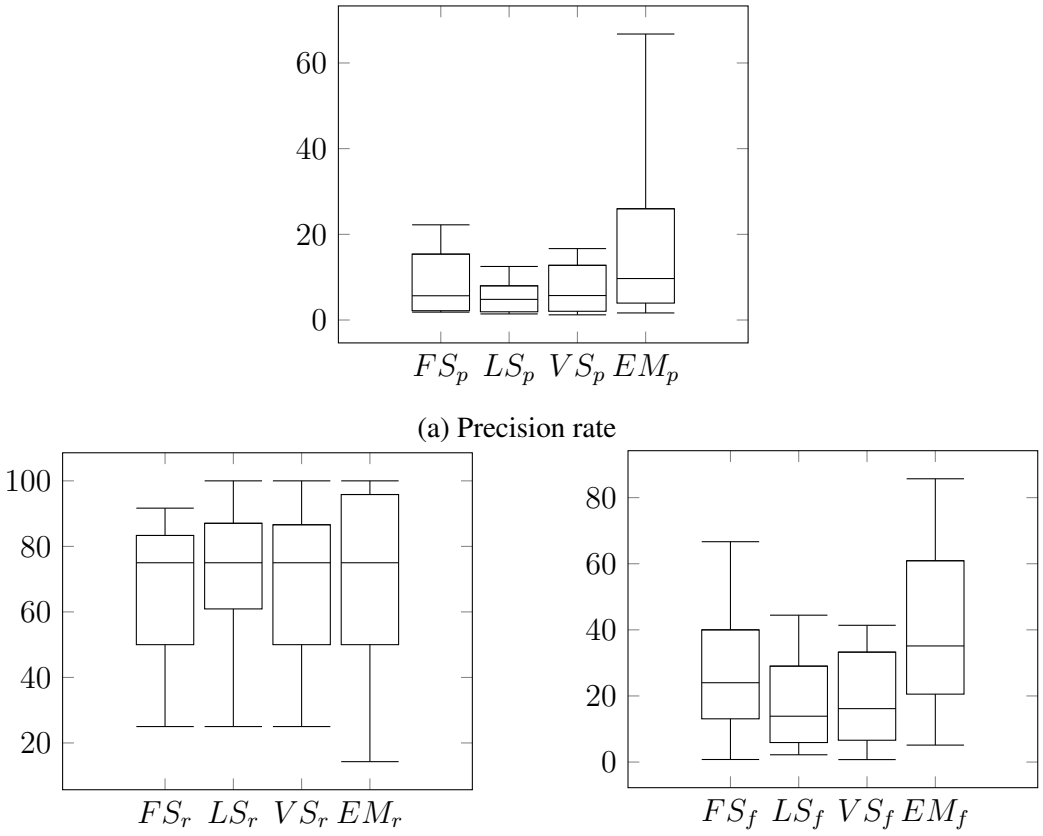


(e) Performance Recall on Project 2



(f) Performance F1 Score on Project 2

Figure 6.1: Performance Measurement on Projects 1 and 2

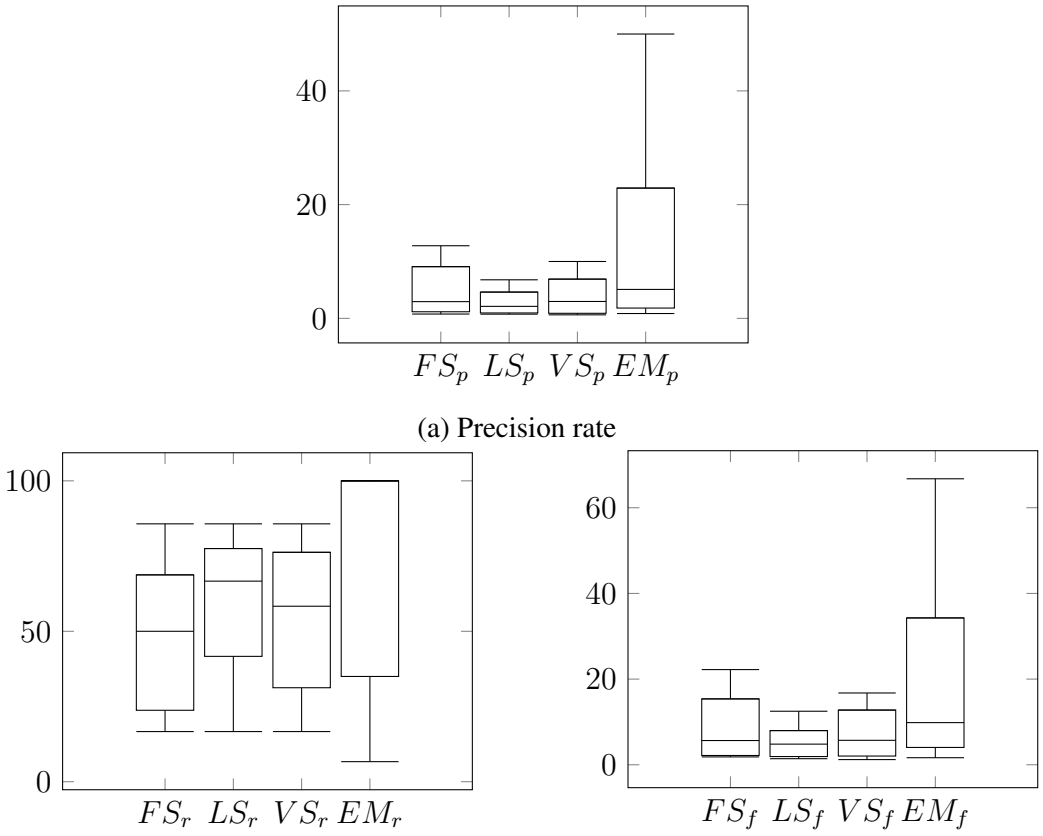


(a) Precision rate

(b) Recall rate

(c) F1 score rate

Figure 6.2: Range of Precision(p), Recall (r) and F1 score(f) of modules on Project 1 and 2.

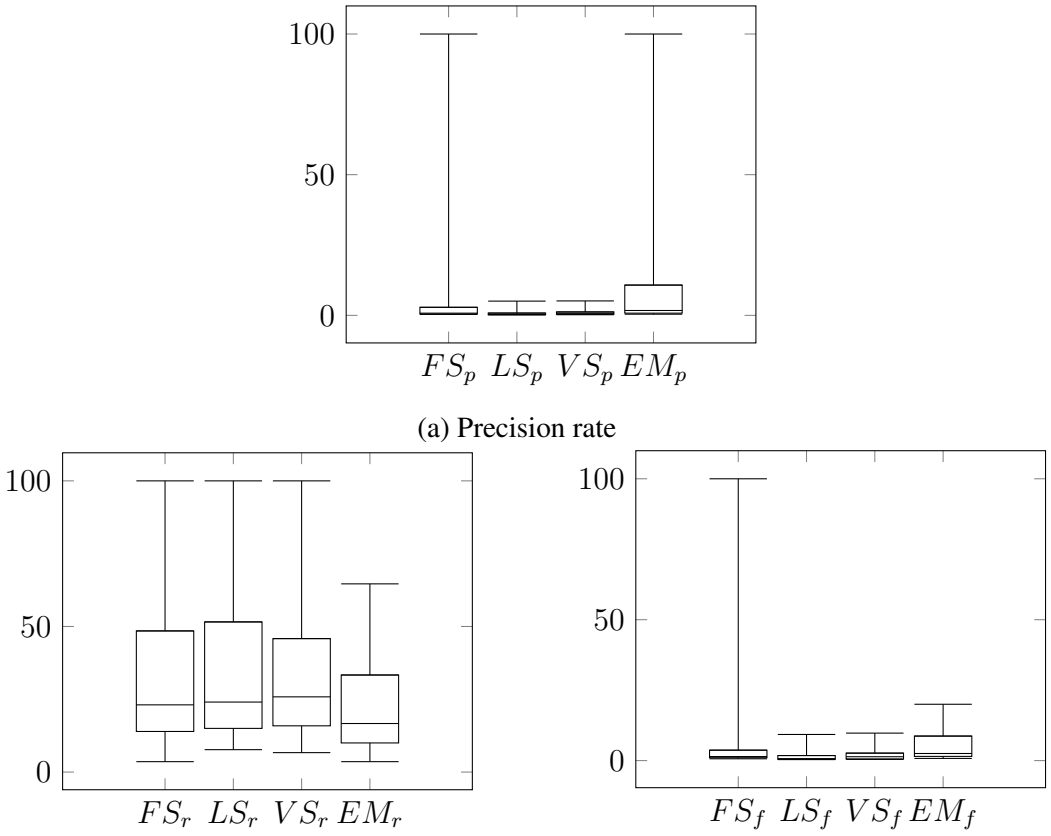


(a) Precision rate

(b) Recall rate

(c) F1 score rate

Figure 6.3: Range of Precision (p), Recall (r) and F1 score (f) of Classes on Project 1 and 2.



(a) Precision rate

(b) Recall rate

(c) F1 score rate

Figure 6.4: Range of Precision (p), Recall (r) and F1 score (f) of methods on Project 1 and 2 .

Chapter VII

Conclusion and Discussion

7.1 Conclusions

The automated technique for linking software features to code components has shown promising results in improving the software development process. Previous researchers have used specific methods like LSI, VSM, and FSECAM but all used methods combine and we create an ENSEMBLE approach from these instances. Our proposed approach will be more accurate than previous work. Our proposed technique offers several benefits, including improved software understanding, increased productivity, better collaboration, and enhanced documentation. Hopefully, it may give better classification accuracy. This approach will help the software developer to find the concept location and also help the new developer to understand the system.

7.2 Discussion

The comparison of outcomes among different approaches provides valuable insights into their permissiveness in ideal scenarios, characterized by standardized feature descriptions and code. Specifically, the evaluation considers modules, classes, and methods within the contexts of LSI, VSM, and FSECAM techniques.

In modules, LSI techniques exhibit no instances surpassing a certain range, whereas VSM techniques show superior performance in one class instance and one method instance compared to FSECAM techniques. Notably, the proposed ENSEMBLE approach outperforms LSI and VSM techniques in 14 out of 18 cases across Project 1 and 2 datasets. Analyzing precision, recall, and F1 scores for modules and classes, the ENSEMBLE approach consistently demonstrates superiority over other techniques in 12 cases. However, for methods, FSECAM techniques outshine the ENSEMBLE approach in precision and F1 score for Project 1, while recall and F1 score for Project 2 exhibit better performance in FSECAM techniques in 6 out of 18 cases.

Comparing the evaluated results numerically with other techniques, Project 1 demonstrates top-tier performance among previous methods. FSECAM exhibits precision scores of 20.4, surpassing ENSEMBLE at 43.5, and LSI, and VSM at 12.1, and 15.5 for Modules, Classes, and Methods, following a similar pattern. Recall values for VSM are 67.9, while Ensemble excels at 88.4, and LSI and VSM show 70.1, and 75.2 for Modules and Classes. In the Method category, VSM outperforms Ensemble with a score of 47.4 against

26.2. Regarding the F1 score, Ensemble consistently outshines VSM, FSECAM, and LSI for Modules and Classes. However, a discrepancy emerges in the Method category, where FSECAM (16.3) outperforms Ensemble (12.5).

Similarly, for Project 2, FSECAM leads with a precision of 24.8, outperforming ENSEMBLE at 31.4, and LSI, and VSM at 5.1, and 6.7 for Modules and Classes. In the Methods category, FSECAM (7.6) surpasses ENSEMBLE (5.4). Recall values for VSM are 63.3, while ENSEMBLE excels at 70.5, and LSI and VSM show 57.2, and 60.5 for Modules and Classes. In the Methods category, FSECAM excels over ENSEMBLE with a score of 29.1 against 17.4. For the F1 score, ENSEMBLE consistently outperforms VSM, FSECAM, and LSI for Modules, Classes, and Methods.

Overall, the observed trend suggests that in scenarios with well-written features and ideal conditions, the proposed ENSEMBLE approach consistently outperforms VSM, LSI, and FSECAM techniques.

7.3 Future Directions

- We evaluated all values by using only Python language and will try to apply another programming language.

- We worked on a closed-source project, but it will apply to an open-source project.

- We will work on more explainable properties of the linking approach.

Bibliography

[1] K. Kelly, *The inevitable: Understanding the 12 technological forces that will shape our future*. Penguin, 2016.

[2] N. L. S. Nuraini, P. S. Cholifah, A. P. Putra, E. Surahman, I. Gunawan, D. A. Dewantoro, and A. Prastiawan, "Social media in the classroom: A literature review," in *6th International Conference on Education and Technology (ICET 2020)*, pp. 264–269, Atlantis Press, 2020.

[3] R. Kitchin and M. Dodge, *Code/space: Software and everyday life*. Mit Press, 2014.

[4] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, "Staged deployment in mirage, an integrated software upgrade testing and distribution system," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 221–236, 2007.

[5] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, pp. 73–87, 2000.

[6] K. Rohit, *Software engineering: Principles and practices*. Vikas Publishing House, 2010.

[7] M. Feathers, *Working Effectively With Legacy Code: Work Effect Leg Code _p1*. Prentice Hall Professional, 2004.

[8] K. R. B. R. K. A. S. Amit Kumar Mondal, Mainul Hossain Chanchal, "Fsecam: A contextual thematic approach for linking feature to muilti-level architectural components," 2023.

[9] A. M. Vable, S. F. Diehl, and M. M. Glymour, "Code review as a simple trick to enhance reproducibility, accelerate learning, and improve the quality of your team's research," *American Journal of Epidemiology*, vol. 190, no. 10, pp. 2172–2177, 2021.

[10] V. Garousi, "Applying peer reviews in software engineering education: An experiment and lessons learned," *IEEE Transactions on Education*, vol. 53, no. 2, pp. 182–193, 2009.

[11] H. D. Mills, "The management of software engineering, part i: Principles of software engineering," *IBM Systems Journal*, vol. 19, no. 4, pp. 414–420, 1980.

[12] B. Potter and G. McGraw, "Software security testing," *IEEE Security & Privacy*, vol. 2, no. 5, pp. 81–85, 2004.

[13] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 375–384, 2010.

[14] Y. Zhao, A. Padmanabhan, and S. Wang, "A parallel computing approach to viewshed analysis of large terrain data using graphics processing units," *International Journal of Geographical Information Science*, vol. 27, no. 2, pp. 363–384, 2013.

[15] A. Olszak and B. N. Jørgensen, "A unified approach to feature-centric analysis of object-oriented software," in *IASTED Software Engineering and Applications (SEA 2010)*, ACTA Press, 2010.

[16] S. P. Roger and R. M. Bruce, *Software engineering: a practitioner's approach*. McGraw-Hill Education, 2015.

[17] K. Wiegers and J. Beatty, "Software requirements: 3-rd edition," *J. Beatty.-Washington: MS Press*, 2013.

[18] C. Bauer, *Hibernate in action*. 2005.

[19] N. D. Birrell and M. A. Ould, *A practical handbook for software development*. Cambridge University Press, 1988.

[20] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pp. 106–115, IEEE, 2010.

[21] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012.

[22] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012.

[23] P. H. Feiler and W. S. Humphrey, "Software process development and enactment: Concepts and definitions," in *[1993] Proceedings of the Second International Conference on the Software Process-Continuous Software Process Improvement*, pp. 28–40, IEEE, 1993.

[24] L. P. Saikia and S. Singh, "Feature extraction and performance measure of requirement engineering (re) document using text classification technique," in *2018 4th International Conference on Recent Advances in Information Technology (RAIT)*, pp. 1–6, IEEE, 2018.

[25] I. E. Araar and H. Seridi, "Software features extraction from object-oriented source code using an overlapping clustering approach," *Informatica*, vol. 40, no. 2, 2016.

[26] E. Bagheri and F. Ensan, "Semantic tagging and linking of software engineering social content," *Automated Software Engineering*, vol. 23, pp. 147–190, 2016.

[27] H.-J. Happel and S. Seedorf, "Applications of ontologies in software engineering," in *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pp. 5–9, Citeseer, 2006.

[28] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker b last: Applications to software engineering," *International Journal on Software Tools for Technology Transfer*, vol. 9, pp. 505–525, 2007.

[29] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability transformed: Generating more accurate links with pre-trained bert models," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 324–335, IEEE, 2021.

[30] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *11th working conference on reverse engineering*, pp. 214–223, IEEE, 2004.

[31] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling, "An approach for mapping features to code based on static and dynamic analysis," in *2008 16th IEEE International Conference on Program Comprehension*, pp. 236–241, IEEE, 2008.

[32] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on software engineering*, vol. 29, no. 3, pp. 210–224, 2003.

[33] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 15–25, 2011.

[34] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pp. 37–48, IEEE, 2007.