ADDIS ABABA UNIVERSITY
COLLEGE OF TECHNOLOGY AND BUILT ENVIRONMENT
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

CO-CHANGE RECOMMENDATION USING DEPENDENCY GRAPH AND CONCEPT LATTICE

BY
TEWODROS MEHERETU

ADVISOR
Dr. SURAFEL LEMMA

A thesis submitted to the School of Electrical and Computer Engineering in partial fulfillment of the requirements for the Degree of Master of Science in Telecommunication Engineering (TIS track)

JUNE, 2025
ADDIS ABABA, ETHIOPIA

ADDIS ABABA UNIVERSITY
COLLEGE OF TECHNOLOGY AND BUILT ENVIRONMENT
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

The undersigned have examined the thesis titled:

CO-CHANGE RECOMMENDATION USING DEPENDENCY GRAPH AND CONCEPT LATTICE

BY

TEWODROS MEHERETU

Approval by Boards of Examiners

Dr. Sosina Mengestu	_____	_____
Head, SECE, CTBE	Date	Signature
Dr. Surafel Lemma	_____	_____
Advisor	Date	Signature
Dr. Fitsum Assamnew	_____	_____
Internal Examiner	Date	Signature
Dr. Bisrat Derebssa	_____	_____
External Examiner	Date	Signature

Declaration

I, Tewodros Meheretu Hunegnaw, hereby declare that this thesis is entirely my original work. All sources of information used in this study have been appropriately acknowledged and referenced. I confirm that this thesis has not been submitted, either in part or in full, for any other academic requirements or to any other learning institution. This work is solely my own contribution to the field of study.

Student Name: Tewodros Meheretu Hunegnaw

Signature: _____

Date: _____

JUNE, 2025

Acknowledgments

First and foremost, I express my deepest gratitude to the Almighty God for His constant guidance and protection; to Dr. Surafel Lemma for his dedicated mentorship that has significantly shaped my academic journey; and to my family for their unwavering love, support, and encouragement, which have been the foundation of my personal and academic achievements.

Abstract

Software change requests are immediate drivers of software evolution. Prompted by the need for new functionality, fixing bugs, or enhanced performance. Resolving such emergent issues can end up being extremely difficult to resolve since software source code entities are highly interconnected. Entities that must change together to assist in resolving change requests are called co-changes. Identification and utilization of bugs typically involve tedious, manual processes, to overcome this issue, bug localization techniques have emerged to help developers. Bug localization is the activity that seeks to find source code entities that are relevant to a given bug report. However, this method does not have any property to extract co-changing entities from the ranked list while applying the bug fix.

This research attempts to identify co-changing source code entities during bug fixing using a source-code-only approach. It is based on a method call dependency graph and a concept lattice of conceptual relationships. The lattice supports a concept lattice-based ranking to recommend the top ten co-change candidates. The study evaluates if the fusion of the two architectures improves performance and compares the results to that of a machine learning approach using a contrastive loss-trained Siamese network. The Siamese model is evaluated with both its native similarity-based and Approximate Nearest Neighbor (ANN) ranking strategies.

As such, the comparative study reflected considerable difference in the success level of the proposed approaches. The combined approach, which utilizes the concept lattice and dependency graph-based dependencies, achieved the rate of 90.33% success, followed by the concept lattice-based method at 84.05%. The dependency graph-based approach in isolation lagged behind with a success rate of 39.6%. On the other hand, Siamese network models varied in terms of context inclusion. The Siamese network achieved a low 24.63% success rate without context. The performance increased with contextual embeddings from GraphCodeBERT to 65.21%. Nevertheless, embedding ANN search achieved success rates of 71.01%. Lastly, the combined and concept lattice-based approaches showed the highest accuracy, with the Siamese network performing competitively only when enriched with contextual information.

Keywords: Co-change Recommendation, Bug Localization, Dependency Graph, Formal Concept Analysis (FCA), Concept Lattice, Siamese Network,ANN, FAISS.

Table of Contents

List of Figures

List of Acronyms

ANN Approximate Nearest Neighbor

FCA Formal Concept Analysis

CChpr Co-change Prospect

LiCh Likelihood of Co-change

FCP2Vec File-level Change Propagation to Vector

TARMAQ Targeted Association Rule Mining for All Queries

FAISS Facebook AI Similarity Search

IVF Inverted File Index

PQ Product Quantization

AI Artificial Intelligence

CPU Central Processing Unit

GPU Graphics Processing Unit

AST Abstract Syntax Tree

ECF Eclipse Communication Framework

NLP Natural Language Processing

JSON JavaScript Object Notation

OSR Overall Success Rate

Chapter 1. Introduction

Software programs evolve constantly to meet changing business needs and new technological trends. Updates have to be constantly implemented to fix defects, improve them, and introduce new features. However, handling changes in complex software programs is very difficult [1].

Modifications to part of a codebase will also have corresponding changes elsewhere within its component pieces to maintain cohesiveness and functionality. Such software co-change produces a range of complications that developers must deal with by navigating through large codebases, considering dependencies, and keeping module cohesiveness intact. Hand-coded co-changes have a higher chance of introducing defects, leading to bugs, regressions, and system crashes. Partial or faulty changes can have cascaded impacts within the system. The ripple effect is used to quantify the likelihood that changing one module will have unintended consequences on the overall program [2, 3, 4].

The absence of automated change management also makes it worse by adding to codebase complexity and making it increasingly problematic to maintain in the long term. With increasing software size, scalability is also of concern, but manual change management hinders the effective management of large and complex codebases. Contemporary software development is dynamic, where there are frequent updates and enhancements of software. But this makes software complex, bringing along with it bugs that mostly result from complex relationships among pieces of software. Mostly through manual processes, one can identify and fix these bugs.

Studies have shown that more than 70% of bug fixes involve changes to multiple files, and over 60% of defects are linked to change-prone or interdependent modules [5, 6, 7]. These figures underscore the scale and urgency of the co-change problem, reinforcing the need for automated support.

To address this challenge, bug localization techniques have been developed that give developers a ranked set of software entities that are related to a given bug report.

1.1 Bug Localization Technique

Bug localization is an essential part of software engineering whose goal is to determine the root causes of software faults [8]. This process employs many techniques and tools specifically designed to identify the exact lines of code or program elements that are causing a bug. Two of the most popular methods researched in existing bug localization literature are spectrum-based methods [9] and information retrieval-based techniques [10, 11, 12]. Derivation of static features from source code or execution data using the spectrum-based approach is time-consuming. The IR-based approach views source files as documents and bug reports as queries. Then it ranks possible files based on the similarity between the contents in the source files and the bug reports [13].

Since bug localization is an integral part of software evolution, many conclusions can be drawn from the associated research. For instance, Tantithamthavorn et al.[14] assess whether or not constructs impact bug localization and determine that methods and classes are constructs that significantly impact bug localization efficiency. Note that this means information retrieval is more successful over generalized programming languages such as C# and Java[15]. In addition, Saha et al.[16] who attempt to increase bug localization accuracy establish a more structured information retrieval. Their efforts come from an average structure and semantics of source code and linked bug reports, indicating their findings exceed average bug localization techniques[16].

A more broad overview of methods comes from Shree and Pushpalatha et al.[17] who created a meta-analysis that tests bug localization efficacy across datasets and frameworks which allows for comparison and understanding of how methods evolved from one to another. Mere validation that bug localization required IR and program spectra was increasingly advanced by Le et al.[18] who formed a multimodal approach to localization akin to a translator between natural language and coding language[18]. The most recent contribution was a deep learning based approach for SBugLocater which suggested a deep matched model consisting of three layers of semantics, relevancy and collaborative filtering. The standard deviation between the mean of other models in accurately finding relevant buggy files was large[19].

1.2 Problem Statement

The development of a modern software program is a dynamic process characterized by frequent updates and improvements to meet the evolving desires of people and technological advancements. Bugs appear in this development due to the complexity of software program systems, involve difficult dependencies between code components. Identifying bugs in software program improvement is usually a manual and time-consuming task. To solve this problem, several bug localization techniques have been introduced to help identify source of bugs [14, 16, 17, 18].

Bug localization techniques give a ranked list of source code entities that are relevant to a given bug report [14, 16]. When the technique is good, developers usually need to look only the first couple of source code entities in the ranked list to identify one of the entities that needs to be changed to fix the bug. For most bug reports, however, there is more than one entity that has to be co-changed to fully fix the bug. While some earlier works have attempted to identify co-changes through source code analysis [20, 21, 22, 23, 5, 24] and change history mining [22, 23, 5], and conceptual coupling techniques—e.g., Relational Topic Models and feature-based estimations—were powerful in detecting semantic relations [25, 26, 27], but are constrained by issues like high computational cost, validation gaps, and susceptibility to subjectivity or overfitting. In this paper, we attempt to bridge these limitations by proposing a hybrid method that combines dependency graph analysis and concept lattice construction to identify co-changing entities needed to fully rectify bugs initially detected by bug localization techniques.

To investigate and address the limitations outlined above, this study is guided by the following research questions.

RQ1 *Can Dependency graph and Concept lattice be used to identify co-changing entities from the source code?*

RQ2 *Does combining concept lattice and dependency graph based outputs improve the recommendation system?*

RQ3 *Do the concept lattice-based, dependency graph-based, and combined approaches outperform the Siamese network in co-change recommendation tasks?*

1.3 Objectives

1.3.1 General Objective

The objective of this research is to develop an approach that identifies source code entities that need to co-change to fully fix a bug using only the source code.

1.3.2 Specific Objectives

- Develop dependency and concept lattice using the source code which is containing classes, methods and attributes.
- To identify which entities need to be co-changed when fixing the bug.
- To develop a method that generates the top 10 co-change file recommendations for a given initial changed file.
- To compare the effectiveness of the primary co-change recommendation approaches with various configurations of the Siamese Network model.

1.4 Contribution

This study offers three key contributions to the domain of software maintenance and evolution:

- **Advancement in Software Maintenance Practices:** The proposed co-change recommendation approaches assist developers in identifying source code components that are frequently modified together, thereby facilitating more informed and accurate maintenance decisions.
- **Improved Developer Efficiency:** By offering guidance on potential co-changing entities, the system supports developers in efficiently navigating and managing complex and interdependent codebases.
- **Mitigation of Software Defects:** Through the identification of implicit and structural dependencies among source code entities, the approach reduces the likelihood of introducing errors due to incomplete or inconsistent changes, contributing to more reliable software systems.

1.5 Scope and Limitation

The Scope of this study lies in the conceptualization and development of a co-change recommendation system. The study is aimed at structured open-source software systems that provide available source code and bug report data for analysis. Consideration is omitted for proprietary software systems since the artifacts such as source code and historical change records required for analysis are not present.

1.6 Organization of the Study

The remaining part of this thesis is structured as follows: Chapter Two outlines the underlying theory of the methods applied, including principal concepts and background technologies. Chapter Three presents literature review with the three co-change recommendation strategies of main concern. Chapter Four outlines the methodology employed in this research, including data preprocessing, model formulation, and training procedures. Chapter Five discusses and presents experimental setup, results and discussion, and offers a glimpse into the performance of each approach. Chapter Six finally summarizes the main findings and suggests potential directions for future research.

Chapter 2. Background

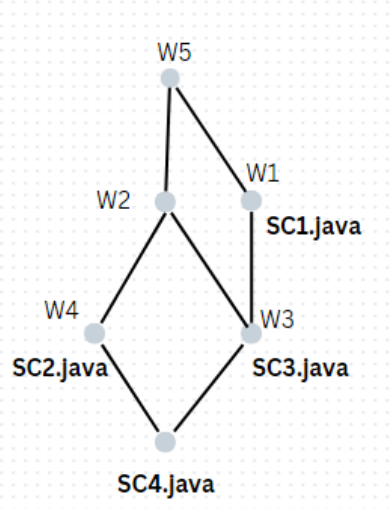
The theoretical underpinnings of Formal Concept Analysis (FCA) are presented in this chapter, along with a thorough rundown of FCA algorithms relevant to this study. The idea of dependency graphs and how they are used in co-change suggestion are covered in detail in the following sections. The chapter then goes on to examine the structure and function of Siamese networks within the framework of this research. The theoretical underpinnings of Approximate Nearest Neighbor (Approximate Nearest Neighbor (ANN)) search techniques are examined in the last part, emphasizing their importance for effective suggestion generation.

2.1 Concept lattice

WILLE et al.[28] created formal concept analysis, which has since been used in a wide range of fields, including computer sciences, mathematics, industrial engineering, biology, medicine, psychology, sociology, anthropology, and linguistics.

	W1	W2	W3	W4	W5
SC1.java	x		x		x
SC2.java		x		x	x
SC3.java	x	x	x		x
SC4.java	x	x	x	x	x

(a) Formal context



(b) Concept lattice

Figure 2.1: Formal concept representations

The study of hierarchically classifying objects according to their attributes is known as formal concept analysis, or FCA. A concept lattice is the collection of all hierarchical relationships in FCA [29]. The hierarchical relationship between qualities and objects is depicted by a concept lattice[29]. A formal concept representation of four objects (files) and five characteristics (words within the files) is shown in Figure 2.1. The formal context is shown in Figure 2.1a, which shows the words' presence in every file. The presence of a specific characteristic in a given file is indicated by a 'X' mark. The attribute 'W2', for example, is present in SC2.java, SC3.java, and SC4.java but not in SC1.java.

The concept lattice of the formal concepts shown as a table in Figure 2.1a is shown in Figure 2.1b. Different information can be inferred more easily with the use of a concept lattice representation. Since SC3.java is a descendent of SC1.java, for instance, all of the properties found in SC1.java are likewise present in SC3.java. The concept lattice also shows that 'W5' is the root node in every file, indicating that this property is present in every file.

Formally speaking, a concept lattice is a triplet (A,B,G), where A stands for a collection of objects (in this case, files), B for a collection of attributes (words), and G for a binary relation between A and B. The following subsection outlines key properties of Formal Concept Analysis (FCA) that are relevant to this study.

2.1.1 Important properties

This subsection discusses key FCA operations to provide a deeper understanding of the methods used in this study. Assume a formal context for each operation (A, B, G), $X \subseteq A$ and $Y \subseteq B$ [30]

2.1.1.1 Derivation (prime) operator

X'={b∈B for each a∈X:(a,b)∈G} and B'={a∈A for each b∈Y:(a,b)∈G}

The set of all context-related qualities that all of X's objects share is X', assuming that X is a collection of objects.

2.1.1.2 Infimum

$$(X_1, Y_1) \wedge (X_2, Y_2) = (X_1 \cap X_2, (Y_1 \cup Y_2)'')$$

The most precise generalization that encompasses both ideas is the infimum of two. The closest common node where the two concepts overlap is found by moving upward through a concept lattice.

2.1.1.3 Supremum

$(X_1, Y_1) \vee (X_2, Y_2) = ((X_1 \cup X_2)'', Y_1 \cap Y_2)$

The most generic notion that includes both as subconcepts is the supremum of two conceptions. It is found by moving from the two provided nodes downward in a concept lattice and locating the first (nearest) node where their descendent routes merge.

2.1.1.4 Implication

$X \Rightarrow Y$

A logical condition of the form if X, then Y is called an implication. In mathematical terms, an implication X⇒Y means that Y must be true if X is satisfied. The implication must meet the requirement $Y \in X''$ in order to be considered legitimate. Stated otherwise, the set Y must be a part of the set of objects associated with X'' in order for X to imply Y.

2.1.2 Computational Complexity

In traditional FCA, computation of a complete concept lattice implies listing all formal concepts of a given formal context. This is FCA problem with worst-case time complexity of:

$$\mathcal{O}\left(2^{\min(|A|,|B|)}\right)$$

where $|A|$ is the number of objects and $|B|$ is the number of attributes. In-Close and Next Closure are complete and non-redundant concept set generating algorithms but have rapidly deteriorating performance for dense or large contexts because of exponential increase in the number of possible concept combinations.

Conversely, our implementation takes a tailored and more scalable direction for applications to real-world software analysis. Instead of calculating the complete lattice, we construct an **attribute-object inverted mapping** using dictionary-based representations. This lightweight approach facilitates mining of concept-like relations.

- **Derivation operations** (e.g., computing X' or Y') are implemented using efficient dictionary intersection, yielding a time complexity of $\mathcal{O}(n)$ per lookup, where n is the size of the intersected set.
- **Concept structure generation** avoids exhaustive closure traversal and instead relies on direct mappings, filtering, and selective inclusion based on attribute frequency. This provides a compressed view of the data useful for co-change mining.
- **Memory usage** remains low, as the system only retains partial mappings (object-to-attribute and attribute-to-object), unlike traditional FCA, which requires storing all intents, extents, and lattice links.

Although this method doesn't necessary construct the entire concept lattice, it is considerably faster, more memory-friendly, and scalable to large systems. Moreover, it still retains the ability to mine interesting co-change concepts from the data structure, which allows one to utilize it for downstream recommendation and analysis tasks. This trade-off between completeness and computational complexity makes it highly valuable for software engineering applications, where scalability is usually a significant concern.

2.2 Dependency Graph

A dependency graph is a directed graph $G = (V, E)$, where each node $v \in V$ is a software artifact (e.g., a file, class, or method), and each edge $(u, v) \in E$ is a dependency from node u to node v. Here, nodes are used to denote Java files, and edges represent method-level dependencies between them in the form of function calls. This is derived from a graph theory formalization and allows structural software system analysis [31].

From a graph-theoretic perspective:

- V: set of Java files involved in the project.
- $E \subseteq V \times V$: set of directed edges, where (v_i, v_j) indicates that file v_i calls at least one method in file v_j.

The graph is *directed* and may contain *cycles*, indicating mutual or recursive dependencies.

The graph may also be *disconnected*, especially in large systems with loosely coupled modules.

This representation supports several analytical goals:

- **In-degree** $\deg^-(v)$: number of files that depend on file v.
- **Out-degree** $\deg^+(v)$: number of files that file v depends on.
- **Fan-in/Fan-out analysis**: used to assess the impact scope of a file and detect architectural bottlenecks or hotspots.
- **Reachability**: path-based queries can determine whether changes in one file may propagate to others.
- **Cycle detection**: essential for identifying tight coupling or design flaws.

The visual layout helps in understanding the system's coupling. For example, six sample Java files are shown in Figure 2.2, demonstrating how different method calls are distributed across the system.

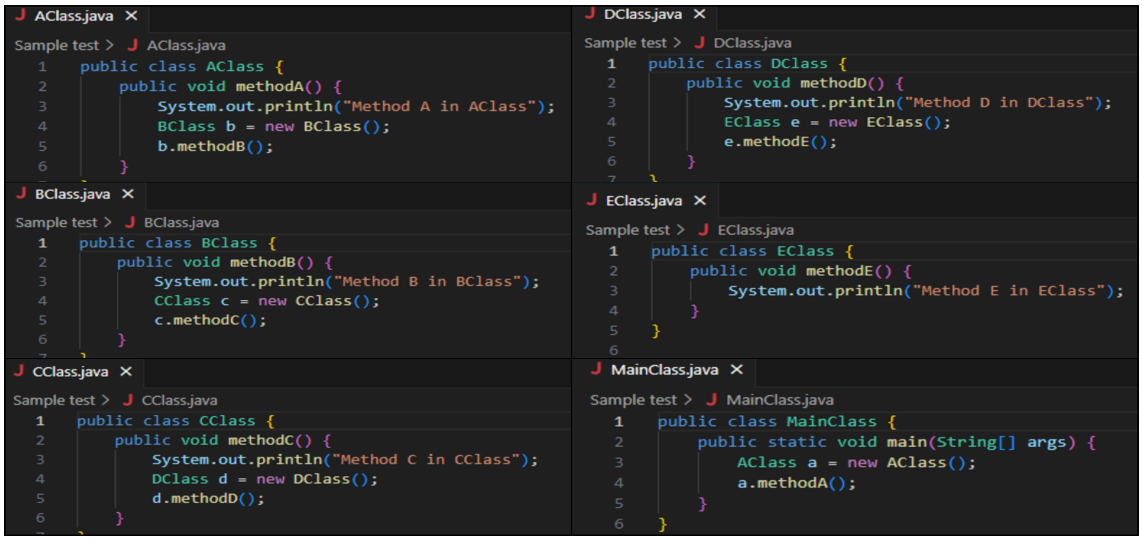


Figure 2.2: Sample Java Files

Figure 2.3 presents the resulting Java method call dependency graph built from these samples. A directed edge in this graph from node A to node B indicates that a method declared in file A calls a method in file B. For example, there is a chain of dependencies from `MainClass` to `AClass` to `BClass` with a path `MainClass` $\rightarrow$ `AClass` $\rightarrow$ `BClass`. Such a chain reveals transitive dependency relationships, which are important for change impact analysis.

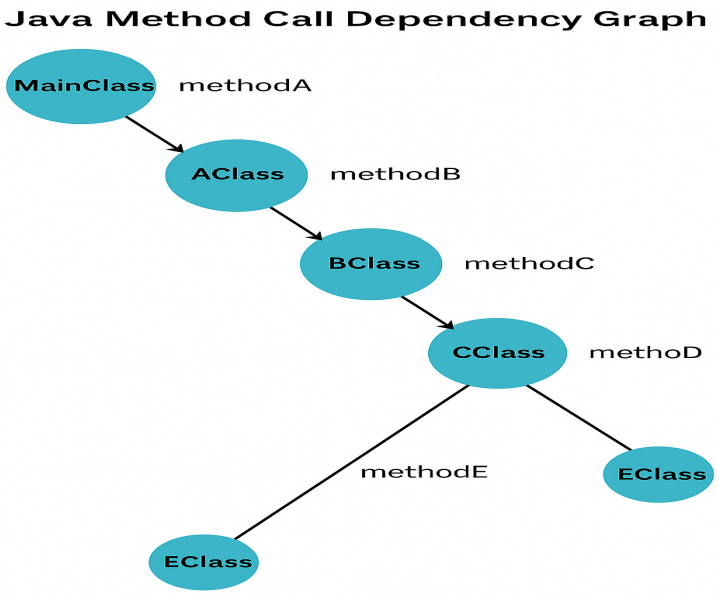


Figure 2.3: Java Method Call Dependency Graph

By showing these relationships explicitly and graphically, the graph aids vulnerable code identification, test case selection, and automatic co-change prediction methods. Its encoding as an adjacency matrix or adjacency list can be programmatically utilized for integration with semantics-based recommendation systems as well.

2.3 Siamese Network

Siamese network is a type of neural architecture that is meant to learn similarity functions using the comparison of two input samples. Originally developed by Bromley et al.[32] for signature verification, a Siamese network comprises two or more identical subnetworks sharing the same weights and structure. Each subnetwork takes one of the inputs and outputs a feature embedding. Its aim is to pull the alike items together in the embedding space and move the unalike items apart. Due to its capacity for modeling pair-wise relations, the Siamese network has been used extensively in applications like face recognition, one-shot learning, and code clone detection[33]. Its shared-weight architecture guarantees that the embeddings are learned in a symmetric and consistent manner, as required by similarity-based recommendation systems such as co-change recommendation.

2.3.1 Contrastive Loss

Contrastive loss is a metric learning task that Siamese networks use to train the model to discriminate between similar and dissimilar pairs of inputs. For a pair of samples, the network calculates their respective embeddings and then calculates their Euclidean distance. The contrastive loss function punishes the model when similar pairs are embedded distant from one another, or when dissimilar pairs are embedded close to one another. Formally, it is defined as:

$$\mathcal{L} = y \cdot D^2 + (1 - y) \cdot \max(0, m - D)^2$$

where D is the pair's Euclidean distance, $y \in \{0, 1\}$ denotes whether the pair is similar (1) or dissimilar (0), and m is a margin parameter[33]. Contrastive loss is effective for binary classification tasks, particularly for learning discriminative embeddings from small data.

2.3.2 Triplet Loss

Triplet loss extends the concepts of contrastive learning but uses the training routine on three samples: an anchor, a positive (similar), and a negative (dissimilar). The goal is to minimize the distance from the anchor to the positive by some margin over the distance from the anchor to the negative. This assists the model in learning finer relative distances among samples. Triplet loss function is defined mathematically as:

$$\mathcal{L} = \max(0, D(a, p) - D(a, n) + m)$$

where $D(a, p)$ denotes the distance between anchor and positive, and $D(a, n)$ denotes the distance between anchor and negative, and m is the margin[34]. Triplet loss has gained a lot of popularity in face verification and ranking-based recommender systems because of its strong comparative training procedure.

2.4 Approximate Nearest Neighbor (ANN)

Approximate Nearest Neighbor (ANN) search is a computational technique that attempts to efficiently fetch data points in high-dimensional space that are ”close enough” to a query point, without always necessarily returning the absolute closest match. In most machine learning and information retrieval systems—e.g., recommendation systems, image search, and natural language processing—datasets usually comprise millions of high-dimensional vectors. Nearest neighbor search in these spaces can be computationally expensive due to the ”curse of dimensionality,” under which traditional algorithms like k-d trees degrade severely in performance with high dimensionality[35]. ANN methods offer a trade-off between accuracy and speed by allowing small mistakes from the true nearest neighbors, leading to drastic gains in scalability and query time.

2.4.1 FAISS (Facebook AI Similarity Search)

Facebook AI Similarity Search (FAISS) is among the popular open-source Facebook Artificial Intelligence (AI) Research libraries providing state-of-the-art ANN methods optimized for dense vector similarity search and clustering at scale[36]. FAISS provides both Central Processing Unit (CPU) and Graphics Processing Unit (GPU) computing support, enabling one to search billions of vectors with low latency. It provides a number of indexing methods like Flat (brute-force), Inverted File Index (IVF), and Product Quantization (PQ) so that users can trade off between search accuracy, indexing time, and memory consumption. Its modularity and its highly optimized C++ backend with Python bindings make it especially well-suited for integration into deep learning pipelines, e.g., embedding-based recommender systems or co-change prediction in source code analysis.

In summary, for comparative purposes, the Siamese Network with a Contrastive Loss function was employed, as the dataset used in this research is not well suited for Triplet Loss. This is primarily because constructing effective triplets—consisting of anchor, positive, and hard negative examples—requires careful sampling strategies and large, well-balanced datasets. Given the limited co-change instances per bug ID’s and the sparse nature of the file relations, reliably identifying meaningful hard negatives would be challenging and could compromise the stability and convergence of triplet-based training.

Chapter 3. Literature Review

Over the last few years, the recognition that understanding the dependencies between different parts of a codebase is very important has become widespread in the software development world. A number of studies have tried to identify which files tend to get changed together—a phenomenon known as co-change. Knowing this is central to ensuring that the software remains high-quality and that bugs can be fixed efficiently.

In addition, Co-change plays a vital role not only in resolving software defects but also in enhancing overall software maintainability and developer productivity. By identifying co-change patterns, development teams can proactively target areas of the codebase prone to defects, thereby improving the speed and accuracy of bug fixing. Moreover, automated co-change tracking reduces the risk of introducing new defects during maintenance and contributes to better-informed design and implementation decisions.

Co-change detection is done by three main methods. The first is evolutionary (or historical) analysis; this examines version control data over time to detect patterns of co-change. The second is a conceptual method, which works by determining the relationships—semantic or otherwise—between entities in the code. The third method is a hybrid approach that uses both pattern detection in historical data and analysis of the semantic relationships between code entities to determine what has changed in what else and how reliably we can know that.

Every one of these groups has the distinct advantages and challenges of their own. If we add up the pluses and minuses, we get the advantages and challenges of modern software.

Take evolutionary methods, for example. They tend to work pretty well when we're trying to capture changes over time. However, they have a somewhat easier time with shallow relationships than with deep, sometimes hidden ones. Conversely, conceptual approaches can provide insights based on the structure and functionality of the code but may lack the empirical grounding provided by historical data.

As we delve into the literature surrounding these co-change recommendation techniques, it becomes evident that while significant progress has been made in identifying files that tend to change together, there remains a gap in the ability to translate this information into actionable insights for bug localization. The idea of this research is that bug localization can identify buggy files but doesn't suggest co-changed entities to fully fix the bug, so this study use co-change recommendation to support the bug localization.

The following sections present a review of the literature related to the three primary approaches for identifying co-changing files.

3.1 Evolutionary Co-change Approach

Evolutionary coupling, also known as change coupling, has been considerably delved in software engineering exploration and practice. If a group of program realities changed together(i.e.,co-changed) constantly during the elaboration of a software system, it's anticipated that the realities in the group are coupled. This type of coupling which we can realize from the evolutionary history of a software system is called evolutionary coupling. If a group of program realities displayed evolutionary coupling in history, a change in one reality in the future is probably to bear matching changes to the other realities in the group [37].

Islam et al. [37] introduce a novel concept called transitive association rules, which help to identify evolutionary coupling among program entities that have not co-changed in the past. The authors refer to this coupling as transitive evolutionary coupling. Regular association rules cannot predict coupling between program entities that have not previously changed together. A regular association is a formal expression of the form $X \rightarrow Y$, where X is the antecedent and Y is the consequent. It represents a relationship between sets of program entities. Association rules are used to identify frequent item sets in large databases and can be applied to software systems to detect relationships between program entities. An association rule has two measures: support and confidence. Support is the number of commits an entity or a group of entities changed. Confidence of an association rule $X \rightarrow Y$ is the conditional probability that Y will change in a commit operation given that X has changed in that commit operation. The authors apply transitivity to regular association rules to create transitive association rules. A co-change prediction mechanism that employs both regular and transitive association rules outperforms a state-of-the-art technique called Targeted Association Rule Mining for All Queries (TARMAQ). The use of transitive association rules in conjunction with regular association rules can enhance the prediction of co-change among program entities. However, Islam et al.'s technique is very time-consuming, making it unsuitable for making co-change suggestions in real-time coding environments.

Agrawal et al. [38] propose an approach that combines software metrics with revision history to establish relationships among co-changed classes and predict future changeability patterns. This approach utilizes regression analysis to build a prediction model for change proneness. The authors validate their method using eight open-source software applications, demonstrating its potential applicability in diverse software development contexts. The practical implications of their work include the identification of co-changed classes, prediction of changeability patterns, and evaluation of change impact. The results indicate promise for significantly benefiting software application maintenance. However, two limitations worth considering are the assumption of static relationships and Missing co-change identification, Assumption of static relationships: The approach assumes a consistent relationship between software metrics and change proneness over time, which may not hold true in dynamic software development environments. Missing co-change identification: Unlike other approaches that consider structural information, this method does not address the specific identification of co-changed classes.

Software change impact analysis aims to identify classes that are likely to be affected by a change. Revision history provides valuable information for identifying evolutionary coupling. Agrawal et al. [1] did an empirical study on the age of change history influences co-change prediction results, with older commits having less relevance than newer ones. Recent studies suggest that emphasizing recent changes can improve co-change prediction accuracy due to their higher likelihood of indicating instability. Their study empirically evaluates the impact of commit age on co-change probability prediction. The analysis utilizes 13 open-source software systems and two coupling metrics: Likelihood of Co-change (LiCh) and Co-change Prospect (CChpr). LiCh helps calculate the co-change probability for a pair of classes, while CChpr estimates class change coupling. To enhance changeability prediction accuracy, the proposed framework filters change commits before calculating coupling metrics. Singular or lengthy change commits are disregarded as they provide limited insights into co-changed classes. Revision history is then divided into two sets: the reference set (R) and the prediction set (P). The reference set comprises the entire revision history excluding the latest revision and is used to generate co-changeability patterns. The prediction set consists solely of the latest revision history and is used to calculate the age of co-changed classes. Once the age of co-changed classes is determined, a weight factor is calculated, assigning a higher weight to more recent co-changes.

Finally, LiCh and CChpr are calculated using the weight factor. To identify classes affected by a change, the change influence set is employed, utilizing a cut-off point. LiCh values above the cut-off point are considered indicative of co-changed classes. The results demonstrate that assigning higher weight to newer revisions and lower weight to older revisions improves prediction accuracy. This research provides new insights into the role of revision age in making accurate predictions. However, the study does not provide a comparison with other existing approaches for co-change prediction, limiting the ability to assess the effectiveness of the proposed approach. Also, the potential scalability issues of the proposed framework when applied to larger software applications with a higher number of classes and commits.

Mondal et al. [39] explore five different ranking mechanisms for prioritizing co-change candidates identified through evolutionary coupling. These mechanisms are evaluated on thousands of revisions across ten software systems. Their findings reveal that the optimal ranking mechanism can vary depending on the software system's evolutionary stage. To address this challenge, they propose HistoRank, a history-based ranking approach that analyzes past co-change occurrences to prioritize candidates. HistoRank demonstrates superior performance across all subject systems. Further investigation of two HistoRank variants confirms that the original version, which prioritizes the most recent co-changes, achieves the best results. However, the revision history lengths of the investigated subject systems might not be sufficient to generalize the findings. Also, the author does not explore the scalability of the proposed HistoRank approach on larger software systems.

Daihong Zhou et al. [40] did an empirical study that delves into the evolutionary coupling and categorizes frequent co-changes between files. It achieves this through an analysis of 27,087 co-change commits across 6 open-source systems. The study meticulously extracts fine-grained change information from version control systems to investigate co-change relationships between files based on 5 distinct types of program entities. It identifies 6 types of dominating co-change relationships, which can be attributed to structural coupling, semantic coupling, or implicit dependencies. Interestingly, the study reveals that files may exhibit varying co-change relationships at different phases in their evolutionary history. It further demonstrates how the rich information embedded within these fine-grained co-change relationships can empower developers to confidently modify code at multiple locations and facilitate comprehensive change impact analysis. The co-change relationships identified in this study hold significant potential to support software maintenance tasks by providing valuable insights into potential design problems and guiding effective maintenance actions. However, the author does not discuss the potential impact of different programming languages or development practices on the observed co-change relationships.

3.2 Conceptual Co-change Approach

In software co-change, conceptual coupling refers to the implicit or semantic relationship between software artifacts, showing the degree of interdependence based on conceptual similarity. It's connected to the concept of change coupling, which describes how software artifacts evolve in tandem. This idea is critical for understanding the relationships and inter dependencies between various aspects of a software system, and it can be used to supplement existing measures, particularly in activities such as effect analysis and change propagation. Developers can acquire insights into the historical relationships and hidden dependencies inside the software system by analyzing conceptual coupling and co-change relationships, which is useful for change effect analysis and maintaining a well-structured and resilient system [41].

Conceptual analysis is an approach that extracts the conceptual dependence based on non-source code information, together with prominent identifiers in the source code. Ajienka et al. [42] seek to close gaps in the research on measuring the semantic coupling of software classes and the interaction between semantic and change coupling. The study examines the efficiency of evaluating semantic coupling in a software system using identifier-based methodologies versus word corpora of entire classes. The actual results reveal that identifier-based methods are more computationally efficient than corpora-based methods for computing semantic coupling, but they cannot always be utilized interchangeably. There is no association between semantic and change coupling, and there is a directional relationship between the two, with over 70 % of semantic dependencies being linked by change coupling but not vice versa. The report presents the findings of an empirical study of 79 Java-based open-source projects to determine the degree of semantic connection between the pairings of classes and predict the possibility of future co-change. However, the examination of the correlation between identifier and corpora-based methods of computing semantic coupling is time-consuming, resulting in the investigation of only a subset of projects. This may restrict the finding's generalizability.

Poshyvanyk et al. [43] propose a novel set of operational measures for the conceptual coupling of classes in object-oriented software systems, using information retrieval techniques. The authors compare these measures with existing structural coupling metrics to assess their effectiveness in impact analysis. The results demonstrate that the proposed measures capture different aspects of coupling between classes and serve as superior indicators of change ripple effects. This research lays the groundwork for further research in several areas: combining conceptual and structural coupling measures for more comprehensive impact analysis and hidden dependency detection, extending prior work on software clustering, concept location, and high-level concept clone detection. However, the research acknowledges several limitations to consider. First, the conceptual coupling measures rely on the assumption of rational naming conventions for identifiers and comments in the source code. Second, they currently do not account for polymorphism and inheritance. Additionally, the evaluation is based on a single large software system, potentially limiting the generalizability of the results. Finally, the case study employed only structural metrics derived from static source code information, while the results might differ if dynamic coupling measures were used.

Mathur et al. [27] underscore the pivotal role of pre-processing and grouping semantically related change sets before pinpointing logical coupling, particularly within scenarios involving multiple code contributors working on the same change. The researchers employ Support and Confidence as association rule-based metrics to forecast the extent of logical coupling among software artifacts. To validate their proposed method, they present a preliminary evaluation utilizing real-world Git repositories and meticulously document their key observations. By harnessing techniques such as cosine similarity and work item hashtag matching, the author group change sets effectively. Cosine similarity serves to calculate the degree of similarity between revision comments found within software change sets. This calculation hinges upon the term frequency (tf) weight of each term within the respective revision comment. To compute the cosine similarity, the input query and documents undergo a conversion into their corresponding unit vectors of words. Hashtag matching emerges as a technique adept at grouping semantically related change sets within atomic-commit featured Version Control Systems. It operates in concert with cosine similarity to augment the identification of logical dependencies within software artifacts. The authors' findings unveil that hashtag grouping achieved a high degree of accuracy, while cosine similarity yielded minimal false positives. However, they advocate for further research and evaluation to rigorously assess the generalizability and robustness of the proposed method across a diverse array of software systems and development environments.

By utilizing Formal Concept Analysis (FCA), Meleake et al.[44] presented a conceptual-based method for detecting co-changes that is independent of substantial historical data. By building concept lattices from source code properties, their approach proposes conceptually linked co-changes that capture the structural and semantic linkages seen in the existing codebase. In order to improve the matching process with up-to-date contextual information and increase the relevance of the proposed co-changes, the method includes commit messages and commit content as extra characteristics during the recommendation phase. The researchers cloned 36 Java-based Apache projects from GitHub and extracted 106,877 contributions with two to eight modified files in order to assess the methodology. This range was chosen to exclude huge refactoring commits that might add noise by touching numerous unrelated files, while also representing plausible co-changes. For every project version (tag), concept lattices were created, and the proposed files were ranked according to attribute similarity in order to evaluate the recommendations' correctness. 52.82% of the top 10 recommendations, according to the study, coincided with real co-changes, indicating both the approach's promise and areas for improvement.

3.3 Hybrid Co-change Approach

Either historical co-change data or conceptual commonalities between software artifacts have been the main focus of popular techniques. Recent studies, however, highlight the effectiveness of hybrid strategies that combine conceptual and historical data to improve the accuracy of co-change predictions.

Based on historical change log data, HA Ahmed et al.[45] present a File-level Change Propagation to Vector (FCP2Vec) approach to anticipate change propagation in software systems. Based on the file being worked on by developers, FCP2Vec is created as a recommendation system to identify files that may change propagation in the future. To understand the historical development sequence of transactions software change log data, the solution employs a skip-gram algorithm with negative sampling and unsupervised nearest neighbors. FCP2Vec's usefulness is demonstrated in a case study utilizing three publicly available data sets: Vuze, Spring Framework, and Elasticsearch. When compared to the Dependency network approach, the results indicate greater accuracy in predicting change propagation. However, the author does not address how the accuracy of change propagation estimates may be impacted by noise or outliers in the change log data.

Wiese et al. [46] presents an empirical study exploring the effectiveness of incorporating contextual information from issues, developer communication, and commit metadata into software artifact co-change prediction. The study finds that customized prediction models utilizing this contextual information significantly outperform the widely used baseline of association rules in terms of F-measure. It also identifies the most influential types of contextual information, encompassing both social (e.g., communication network closeness) and technical (e.g., code churn) aspects. However, the study's external validity may be limited due to its focus on a specific data-set, and potential over-fitting in the prediction models remains a concern. However, the findings suggest that incorporating contextual information can lead to more accurate co-change recommendations, thereby improving the effectiveness of co-change prediction in software development.Additionally, Jia et al. [47] presented a learning-to-rank method that predicts and ranks co-changed methods at the pull-request level by combining source code features and change history. The Random Forest model outperforms other models in accuracy, highlighting the value of combining several data sources for co-change prediction, according to their studies conducted across 150 open-source Java projects.

This paper builds on and extends the work of Meleake et al. [44], who employed FCA as a technique to identify co-changing source code entities. While their study focused on getting conceptual relationships from FCA, this paper advances that method by combining it with dependency graph analysis and applying it in a broader recommendation scheme. This is done so that detection of co-change in software maintenance processes is more accurate and useful.

Chapter 4. Methodology

There are four sections in this chapter. The dataset and the data collection procedure are presented in the first part. The main methodology, comprising concept lattice building, dependency graph development, and their integration for recommendations generation, is described in the second section. The third section introduces the Siamese network-based approach, explaining its training and recommendation phases, which are later used for comparison. The evaluation method used to gauge the efficacy of all methods are covered in the final section.

4.1 Dataset

For the research conducted in this study, the Eclipse Communication Framework (ECF) project—an open-source Java-based software system—was selected as the target system. The source code consists of 2,802 files, providing a sufficiently large and complex code-base to evaluate the scalability and robustness of the proposed co-change recommendation methods. The ECF subject system was cloned from GitHub due to its publicly available source code and contextual information.

From this subject system, 207 Gold set cases were selected after a series of preprocessing steps. These steps are detailed in Subsection 4.2.2. The selected Gold sets were grouped by the number of actual co-changed files (i.e., ground truth files excluding the first changed file). The distribution is as follows, as presented in Table 4.1:

Table 4.1: Distribution of Gold Sets by Number of Actual Co-changed Files

Number of Co-changed Files	Number of Gold Set Cases
1	59
2	39
3	25
4	14
5	8
6	8
7	12
8	6
9	3
10	6
11	1
12	1
13	1
14	2
16	1
17	2
18	1
19	1
20	17

4.1.1 Data collection method

For data collection, the ECF subject system was obtained from the BLIZZARD project [48] on GitHub, a bug localization tool that hosts six open-source systems, including the one selected for this experiment. The source code files in BLIZZARD were renamed using numerical identifiers for security and other purposes. However, to ensure compatibility with the recommender system, these files were reverted to their fully qualified package names. Retaining the numerical name format would have made it impractical for the recommender system to locate the initially changed file within the generated dataset, as the Gold set retains the original source code names.

In addition, the renaming of the files to their corresponding class names caused many duplicates, which introduced ambiguity during the recommendation process. This was corrected by extracting the entire package structure and reorganizing it.

For instance, if a Gold set entry had the path ”framework/bundles/org.example.module.ui /src/org/example/internal/module/ui/Component.java”, the system renamed it to ”org_example_internal_module_ui_Component.java” to make it unique. The recommender system then used this renamed format to query the dataset, search for matches, and generate a list of files most likely to co-change with the initially changed file.

4.2 Concept lattice and Dependency graph

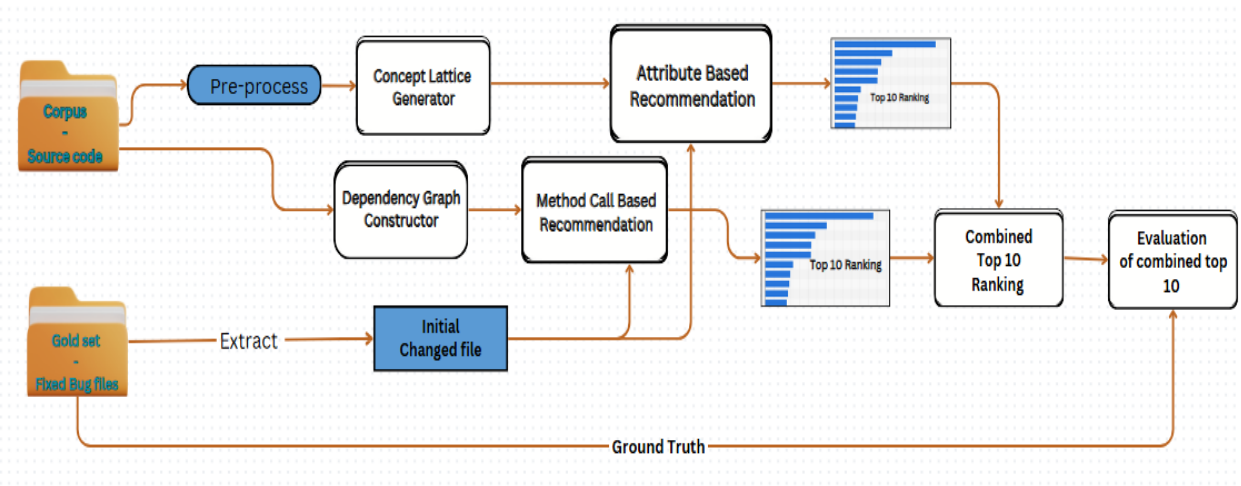


Figure 4.1: High level overview of the Model

This section discusses the proposed methodology used for this research. The source code and the Gold set data provide the input for our study, as illustrated in Figure 4.1. The Dependency Graph Generator and Recommendation System, context lattice, and preprocessing are the three primary parts of our method that work together to achieve the objective. Each component is explained in detail in the subsection that follows.

4.2.1 Concept lattice Pre-processing

To improve the quality of the input, the preprocessing stage is tasked with the removal of irrelevant information. This stage, which works on source code files, includes three basic processes: tokenization, removal of stopwords, and stemming. Together, these processes guarantee that the preprocessed data is well structured and ready for further analysis.

Tokenization is a critical preprocessing task that transforms raw Java source code into structured lexical units to enable subsequent analysis such as concept lattice construction and similarity-based recommendation. Tokenization in this work takes a composite rule-based method involving camel case splitting, whitespace segmentation, and vocabulary filtering. Camel case identifier names such as parseUserInputData are separated into semantically meaningful tokens such as ”parse”, ”user”, ”input”, and ”data” based on recognition of case changes from lower to upper case letters. Whitespaces are used for token delimiting fetched from code constructs such as variable declarations, method calls, and inline comments (such as String userInput = scanner.next(); is tokenized into ”string”, ”user”, ”input”, ”scanner”, ”next”). For semantic significance, the token stream is processed using the nltk.corpus.words module, which provides an English word list to maintain valid linguistic terms. However, to avoid deleting meaningful domain-specific identifiers that exist outside common dictionaries, a manually compiled programming lexicon is kept side-by-side. Such programming lexicon rules maintain software engineering words such as ”dto”, ”repo”, ”configurator”, and ”auth”, which are removed by default English filters but possess significant contextual significance within code. Such rules in aggregate deliver a normalized token set with semantic coverage for overall language semantics as well as domain-specific usage, preserving a high-quality representation of source code for downstream analysis.

Stopword removal is an Natural Language Processing (NLP) preprocessing method that removes frequent words—e.g., ”the,” ”is,” or ”and”—from text data to allow more weight for important words. It improves co-change pattern mining by assigning more importance to important keywords with a higher chance to represent relations among various entities.

The final preprocessing task is stemming, a text normalization technique that reduces words to their root word by removing prefixes and suffixes. The enhancement is required so that various forms of the same word will be processed in a way that consistency will be achieved in text analysis. For instance, the terms ”analyze,” ”analyzing,” and ”analyzed” all trace back to the original term ”analyze.” Stemming clusters related word forms together and thereby enhances the efficiency and effectiveness of co-change pattern mining.

4.2.2 Gold set pre-processing

A Gold set is a collection that consists of a set of modified files according to bug reports, and a Gold set is saved by its corresponding bug ID. The Gold set is used as the ground truth or gold standard to assess software engineering practices.

Our preprocessing of the Gold set for co-change recommendation involved three significant steps. First, file path entries that were single were eliminated since they cannot help in discovering any co-change relation. Second, duplicate file paths in each Gold set were eliminated so that each file was represented only once against each bug ID to avoid over representation. Third, Gold sets that had over 21 file paths were eliminated to avoid noise and for better computational efficiency. These pre-processing steps make the dataset more reliable, leading to a more efficient and accurate co-change recommendation system.

4.2.3 Concept lattice generation

To generate a concept lattice, stemmed words act as attributes, and files serve as objects. JavaScript Object Notation (JSON) is utilized to implement this structure, as it provides a flexible, language-independent format that simplifies object and attribute retrieval. The JSON-based tabular representation offers a structured method for organizing concept lattice, as demonstrated in Figure 2.1, ensuring efficient data accessibility and analysis.

The JSON object representing the formal context shown in Figure 2.1 is structured as follows.

```
object={"Nodes":{
  "SC1.java":{"W1":true,"W3":true, "W5":true},
  "SC2.java":{"W2":true,"W4":true, "W5":true},
  "SC3.java":{"W1":true,"W2":true, "W3":true, "W5":true},
  "SC4.java":{"W1":true,"W2":true, "W3":true, "W4":true, "W5":true} },
 "Edge":{
  "W1":{"SC1.java":true,"SC3.java":true,"SC3.java":true},
  "W2":{"SC2.java":true,"SC3.java":true,"SC4.java":true},
  "W3":{"SC1.java":true,"SC3.java":true,"SC4.java":true},
  "W4":{"SC2.java":true,"SC4.java":true },
  "W5":{"SC1.java":true,"SC2.java":true,"SC3.java":true,"SC4.java":true} }
}
```

Figure 4.2: JSON-based Concept Lattice Representation with Objects and Attributes

Figure 4.2 presents a JSON representation of a concept lattice derived from source code. The object.Nodes section maps to the objects (or classes) in the lattice — each key (e.g., SC1.java, SC2.java) represents a source code file, and the associated values (e.g., W1, W3) represent the attributes (e.g., words) that file contains. Conversely, the object.Edge section lists each attribute (e.g., W1, W2) and maps it back to the set of source code files (objects) that exhibit it.

The JSON object effectively encapsulates all essential information from the tabular representation in a clear and structured manner. It includes both files and stemmed words as attributes, facilitating efficient retrieval during recommendations. While maintaining both attributes may lead to increased memory consumption, the trade-off is justified by the computational advantages it offers, making the memory overhead a manageable consideration.

Concept lattice generation is a process that involves a series of successive stages, each stage playing a differently significant role in total computational expense. At pre-processing based on parsing and structuring attribute data in source files, time expense was incurred around 2.00 seconds and maximum memory usage of 22,910.06 KB. Next, the object structure creation—when the file nodes were given their attributes—was achieved in 0.17 seconds with approximately 8,123.52 KB of memory consumption. Finally, the file structure creation (or node-to-attribute mapping) took a practically negligible 0.04 seconds and 7,880.01 KB of memory, showing that this action carries minimal or no computational burden. The Python's internal data structure was converted into JSON form in 0.60 seconds, and then disk output took another 0.26 seconds.

In total, the overall runtime of the entire pipeline of concept lattice building was below 3.1 seconds, and the memory usage of preprocessing took the most. The computation had a very low overhead, which confirms that the construction of the concept lattice is efficient and scalable enough to be used in practical uses with huge codebases.

4.2.4 Dependency graph generation

The visual representation of the dependency graph is not utilized in this study due to two primary limitations. First, it is not a feasible input for the recommendation system, and second, as the source code size increases, visualizing the graph becomes impractical. Instead, JSON is adopted for a more structured and scalable representation. In this format, each entry under "Nodes" represents a Java class and includes: "methods", listing the methods within the class; "fan_in", identifying incoming dependencies from other classes and methods; and "fan_out", specifying outgoing dependencies to invoked methods. Additionally, the "Edges" array explicitly defines method calls between different classes, detailing both the source class and method initiating the call and the target class and method being invoked. This structured representation facilitates efficient data retrieval and analysis.

The JSON representing the dependency graph shown in Figure 2.3 is structured as follows.

```
{"Nodes": { "AClass.java": { "methods": [ "methodA"  ],
            "fan_in": [ {  "source_file": "MainClass.java",
                    "calling_method": "main"  }  ],
            "fan_out": [ { "target_file": "BClass.java",
                    "called_method": "methodB"  }  ]  },
        "BClass.java": { "methods": ["methodB" ],
```

```
            "fan_in": [{ "source_file": "AClass.java",
                    "calling_method": "methodA" } ],
            "fan_out": [ { "target_file": "CClass.java",
                    "called_method": "methodC"  } ] },
        "CClass.java": {"methods": ["methodC"],
            "fan_in": [{"source_file": "BClass.java",
                    "calling_method": "methodB" }],
            "fan_out": [{"target_file": "DClass.java",
                    "called_method": "methodD" } ] },
        "DClass.java": { "methods": [ "methodD" ],
            "fan_in": [ { "source_file": "CClass.java",
                    "calling_method": "methodC" } ],
            "fan_out": [ { "target_file": "EClass.java",
                    "called_method": "methodE" } ] },
        "EClass.java": { "methods": [ "methodE" ],
            "fan_in": [ { "source_file": "DClass.java",
                    "calling_method": "methodD" } ],
            "fan_out": [ ] },
        "MainClass.java": { "methods": [ "main" ],
            "fan_in": [ ],
            "fan_out": [ { "target_file": "AClass.java",
                    "called_method": "methodA" } ] } },
    "Edges": [ { "source": "AClass.java",
            "target": "BClass.java",
            "method": "methodA",
            "called_method": "methodB"  },
        {   "source": "BClass.java",
            "target": "CClass.java",
            "method": "methodB",
            "called_method": "methodC"  },
        {   "source": "CClass.java",
            "target": "DClass.java",
            "method": "methodC",
            "called_method": "methodD"  },
        {   "source": "DClass.java",
            "target": "EClass.java",
```

```
    "method": "methodD",
    "called_method": "methodE"  },
{   "source": "MainClass.java",
    "target": "AClass.java",
    "method": "main",
    "called_method": "methodA" }]}
```


Figure 4.3: JSON-based Dependency Graph Representation

Figure 4.3 shows the JSON representation of a method dependency graph, illustrating the relationships between Java classes and their method interactions. Each class is represented as a **Node**, containing its defined methods, incoming dependencies (fan-in), and outgoing dependencies (fan-out), while the **Edges** specify method calls between classes. Although the provided sample exhibits a chained dependency structure, in a larger software system, dependencies are typically more complex and interconnected rather than strictly hierarchical. The JSON format ensures a structured yet flexible representation, capable of capturing diverse dependency patterns, including multiple incoming and outgoing connections. Additionally, MainClass.java has an empty fan-in because no other class calls its main method, indicating its role as the program's entry point. Similarly, EClass.java has an empty fan-out, meaning its methodE does not invoke any further methods, making it an endpoint in this specific example. This structured representation aids in co-change recommendation, and providing a clear overview of how methods interact across different classes.

To better understand the practical resource requirements of the system, we also measured the computation time and memory usage at the major phases of the dependency graph construction process. The node construction phase, structuring and extracting file-level method calls, took 33.57 seconds to finish and used about 36,168 KB of memory. The second phase of edge construction, which consists of connecting method call relationships between files (Caller and Called methods), lasted 27.77 seconds and consumed 16,300 KB of memory.

The last step of graph serialization, in which the graph structure was serialized into a JSON representation for further downstream analysis, was comparatively light, finishing in 1.51 seconds and using approximately 52,492 KB of memory, most probably because of temporary in-memory object formatting. In total, the whole process of graph construction was finished in 62.91 seconds, using a peak of 25,736.48 KB of memory.

These numbers show that the static analysis module is computationally lightweight and scalable and can analyze large-sized Java code bases in the order of a minute with moderate memory and is manageable to be applied in large-scale software maintenance workflows.

4.2.5 Recommendation Systems

4.2.5.1 Concept lattice based Recommendation

This phase takes as input the initially changed file along with the concept lattice derived from the source code. The primary objective of the recommendation system is to generate a list of files that are likely to co-change with the initially changed file. Specifically, it identifies files that share the most similar attributes with the initial changed file, thereby facilitating more accurate co-change predictions.

While Formal Concept Analysis (FCA) traditionally relies on implications, as discussed in Section 2.1, this approach requires provide a supporting reference, which may not always be feasible. To address this limitation, an alternative method involves identifying the supremum or infimum of the proposed change, selecting a concept that serves as either an ancestor or a descendant. However, this method also assumes the existence of a single node with attributes precisely matching those of the initial changed file, which is often not guaranteed in practice.

To address the limitations of implications, supremum, and infimum while preserving their conceptual foundation, we propose a recommendation system. This recommender system calculates the amount of attribute similarity between the file initially changed and other files using the concept lattice from its source code. The amount of similarity is calculated by counting shared attributes that capture semantics as represented in the lattice. Then, the files will be ordered by greatest to least similarity to the initially changed file. By running the recommender this way, we ensure that the recommendations stay relevant to the attribute profile with similar attributes as the initially changed file.

Algorithm 4.1: Concept lattice based Recommendation Process

Input: bug_ids : list of bug report identifiers

concept_lattice : conceptual relationship data for all files

Output: top_files : ranked list of co-change candidates for each bug ID

```
foreach bug_id in bug_ids do
    initial_file ← get the initially changed file for bug_id
    ground_truth ← get actual co-changed files for bug_id
    normalize(initial_file & ground_truth)
    foreach file in project_files do
        shared_count[file] ← number of attributes shared with initial_file
    rank ← sort shared_count in descending order
    top_files[bug_id] ← top 10 files from recommendation
return top_files
```

As illustrated in algorithm 4.1, the concept lattice based recommendation procedure begins by loading the concept lattice, which is the attribute representation of the source code. For each Bug ID in the Gold set, the algorithm will find the initially changed file and its co-changed files in the Gold set, which is the ground truth. The filenames are then normalized to be consistent with the file names used in the concept lattice.

After the inputs are normalized, the algorithm loops through all the files in the project and computes a shared attribute count to measure the number of attributes that are shared between the files and initial changed file. The algorithm will rank the files in descending order by their shared attribute count score. It will take the top ten files with the highest shared attribute counts to use as the recommendations of co-change files for the given Bug ID. Finally, results are recorded per bug ID, and a complete summary of top-ranked recommendations is compiled.

4.2.5.2 Dependency Graph based Recommendation

This phase utilizes the initially changed file and the dependency graph extracted from the source code as inputs. The primary goal of the recommendation system is to identify and generate a list of files that are likely to co-change with the changed file. Specifically, it detects files that share a dependency or relationship with the initial change file, facilitating more accurate co-change predictions.

This Dependency graph based recommendation approach assesses the extent of method interactions between the initially changed file and the dependency graph extracted from the source code by calculating dependency scores based on the frequency and depth of method calls. By ranking files based on the frequency of method calls, it generates a prioritized list, emphasizing files with the strongest dependencies on the initially changed file. This method ensures that the recommended files maintain the closest functional relationship to the changed file.

Algorithm 4.2: Dependency Graph Based Recommendation Process

Input: bug_ids : list of bug report identifiers
dep_graph : method call relationships between files

Output: top_files : ranked list of co-change candidates for each bug ID

```
foreach bug_id in bug_ids do
    initial_file ← get initially changed file for bug_id
    ground_truth ← get actual co-changed files for bug_id
    normalize(initial_file & ground_truth)
    foreach file in project_files do
        dep_score[file] ← number of direct & indirect method calls with
          initial_file
    rank ← sort dep_score in descending order
    top_files[bug_id] ← top 10 files from rank
return top_files
```

The high-level workflow for Dependency graph based recommendation is summarized in algorithm 4.2. This process begins by loading the dependency graph that encodes caller-callee relationships among source files. For each bug instance, the initially changed file is compared against all other files in the project to calculate dependency scores based on the frequency and depth of method call interactions. Both direct and indirect method calls are considered. The files are ranked accordingly, and the top 10 candidates are selected as co-change recommendations. Finally, results are recorded for each bug ID, and a complete summary is saved.

4.2.5.3 Combined top ten ranking

This ranking technique combines the top 10 results from both concept lattice based ranking and dependency graph based ranking to generate a unified list. The merging process is score-driven, ensuring that files ranked highly in one method but lower in the other are appropriately repositioned. For instance, if a file is ranked first in the concept lattice based method but seventh in the dependency graph based method, its final rank is adjusted based on a combined score. Files appearing in only one of the lists are evaluated based on their respective scores, with the highest-ranking ones included in the final top 10.

Algorithm 4.3: Combined Top-10 Based Recommendation Process

Input: concept_results, dependency_results : per-bug id recommendations
bug_ids : list of bug report identifiers
w1, w2 : weights for concept and dependency scores, where $w1 + w2 = 1$

Output: combined_recommendations : top 10 co-change candidates per bug ID

```
Initialize Co-Change Recommender system
Load concept lattice and dependency graph results from JSON files
foreach bug_id in bug_ids do
    concept_recs ← get concept lattice-based recommendations for bug_id
    dependency_recs ← get dependency graph-based recommendations for
      bug_id
    initial_file ← get initially changed file for bug_id
    foreach file in union(concept_recs, dependency_recs) do
        norm_concept_score ← (concept_score / max_concept_score) × 100%
        norm_dep_score ← (dependency_score / max_dep_score) × 100%
        combined_score[file] ← (w1 × norm_concept_score) + (w2 ×
          norm_dep_score)
        mark_source[file] ← concept, dependency, or both
    ranked ← sort files by combined_score descending
    top_10 ← select first 10 files from ranked
    save combined recommendations and metadata for bug_id
    print results with ground truth match indicators
return combined_recommendations
```

Algorithm 4.3 illustrates the high-level process of the combined recommendation method. This method uses conceptual similarity based on shared attributes as well as procedure-based relationships based on method call dependencies for a hybrid recommendation.
The process traverses each bug ID after system initialization and loading the output of both concepts lattice and dependency graph. For each recommended file, it computes two normalized scores:

$$\text{Normalized Concept Score} = \left(\frac{\text{Concept Score}}{\text{Max Concept Score}}\right) \times 100\%$$

$$\text{Normalized Dependency Score} = \left(\frac{\text{Dependency Score}}{\text{Max Dependency Score}}\right) \times 100\%$$

To generate a combined ranking, the technique calculates an aggregate score from a weighted sum of the two normalized values:

$$\text{Combined Score} = (w_1 \times \text{Normalized Concept Score}) + (w_2 \times \text{Normalized Dependency Score})$$

Where:

$$w_1 = 0.7 \quad \text{is the weight for concept score,}$$

$$w_2 = 0.3 \quad \text{is the weight for dependency score,}$$

$$w_1 + w_2 = 1$$

The value weights used for the hybrid score were empirically obtained. Various combinations were attempted to verify their effect on overall rate of success, and the combination $w1$ =0.7 for concept score and $w2$ =0.3 for dependency score produced optimal performance over the entire dataset. This setting was always producing maximum overall rate of success in co-changing file detection to justify its adoption.
A ranking is then carried out of all candidate files based on the sum of their scores. The top 10 are selected as final recommendations. Results are then saved and cross-referenced with the ground truth files to validate correctness. This hybrid method ensures semantic cohesion combined with structural relevance in the output of the recommendation.

4.3 Siamese Network

Due to the increasing use of machine learning in contemporary software engineering tools, comparing the proposed co-change recommendation approach with a machine learning-based method helps to reveal its relative efficiency and applicability. To this end, a Siamese network was selected as the model machine learning method. It was selected for three essential reasons:

1. Pairwise similarity learning fit quite well with the task of co-change identification.
2. it enables supervised learning from unlabeled data without requiring large sets of labeled data, instead uses synthetically generated positive and negative pairs;
3. and it can work without contextual embeddings, making possible comparable comparison with plain code-based structural representations.

This comparison not only exhibits the efficacy of the proposed method but also situates its performance in the context of machine learning-based solutions.

4.3.1 Siamese Network without Contextual information

4.3.1.1 preprocessing

The initial phase involved preprocessing raw Java source code files into syntactic embeddings. The single source files were analyzed into an Abstract Syntax Tree (AST), a tree structure for source code grammatical structure. Path-contexts—triplets consisting of a start token, syntactic path, and end token—were derived based on grammar paths from ASTs. This approach is based on the Code2Vec representation, which encodes source code semantics through AST traversal [49].

For representing path-context elements in a fixed-size vector, a Word2Vec model was learned from the training data[50]. Each file was then represented by computing the vector representation of its path-contexts and aggregating them, with a single dense file per file. These embeddings were stored in a normalized file name to vector dictionary. Importantly, no bug-report or co-change information was used for this step; embeddings were derived purely from the syntactic structure of each file.

4.3.1.2 Model training

In the second phase, a Siamese neural network was trained on the syntactic embeddings[32]. The network architecture consisted of two identical parameter-sharing subnetworks, each with two fully connected layers. The two sub-networks took a pair of file embeddings as input and projected them onto a lower-dimensional feature space.

The network was trained on a set of pairs of files. They were labeled as similar (positive) or dissimilar (negative) irrespective of accessing historical co-change information or gold sets. Positive pairs were sampled by selecting two distinct files from the dataset randomly, whereas negative pairs were formed by pairing a file with another randomly selected file that was not assumed to be similar. Although the labels were synthetic, this method enabled the network to generalize an embedding space that captures structural similarity between files.

The Siamese model was optimized using contrastive loss function reducing distance between embeddings of similar file pairs and penalizing dissimilar file pairs more than a given margin for similar embeddings [33]. Ten epochs were trained using the Adam optimizer [51]. The trained Siamese model was then stored for use in future recommendation tasks.

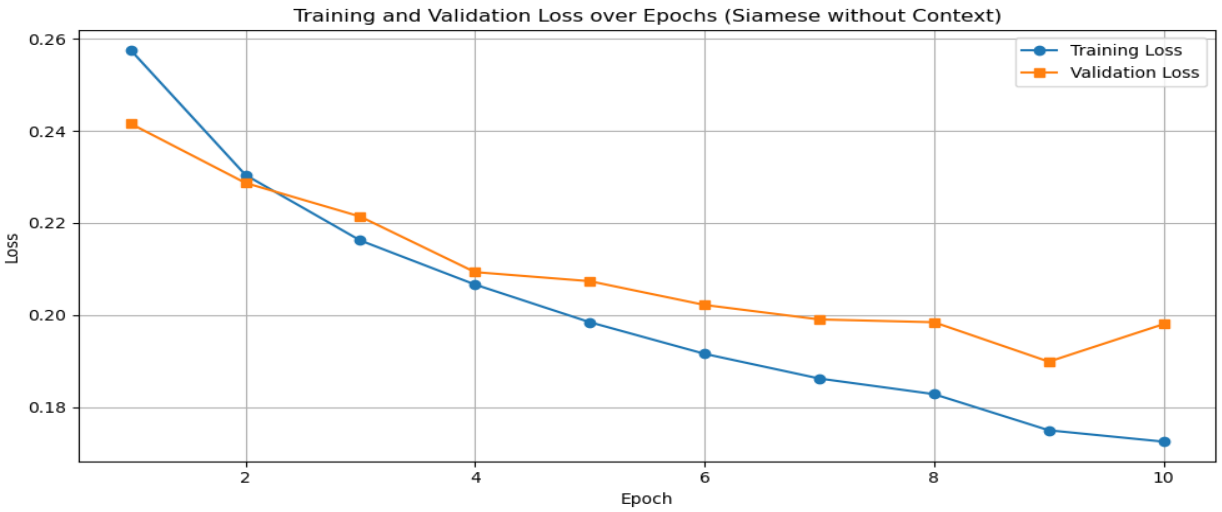


Figure 4.4: Training Loss over Epochs (Siamese without Contextual Info)

The graph shown in Figure 4.4 represents the learning pattern of the Siamese network when it is trained without contextual information. The x-axis presents the number of training epochs (from epoch 1 to epoch 10), and the y-axis displays corresponding values of contrastive loss. The two separate lines represent training loss and validation loss across epochs, offering a clear view of the learning pattern of the model.

As can be seen, training loss reduces steadily over time, indicating effective learning as well as efficient optimization of file similarity representations. However, while validation loss follows a related declining pattern in the beginning, from around the fifth epoch onwards it gradually deviates from training loss. This slow growth of validation loss towards the later epochs suggests initial overfitting—where the model continues to learn how to do better than the training data but begins generalizing badly on unseen validation pairs.

Unlike the model that learned from contextual embeddings, this one has a slightly less stable validation curve, presumably because of the semantic sparsity of the input features. Nevertheless, both losses are still very close, and the overall profile shows that the training process is converging heavily.
In short, although the model performs quite nicely, the observed validation trend confirms that contextual clues need to be added to boost generalization and avoid potential overfitting when learning over an extended time span.

4.3.2 Siamese Network with Contextual information

4.3.2.1 preprocessing

At this preprocess, the source code was transformed into numerical representations with contextualized information using GraphCodeBERT, a pre-trained transformer model developed with special focus on programming languages. GraphCodeBERT captures both syntactic form and data flow dependencies of code to further enhance its semantic representation abilities across programs [52].
All Java files in the dataset were processed through GraphCodeBERT to generate high-dimensional pooled embeddings containing the overall semantics of the source code. Each pooled embeddings was saved independently for each file. For contextual co-change relations, the gold set, divided by bug IDs, with each set containing files co-changed together, was utilized. The first file in each bug ID was designated as the ”anchor,” and the remaining ones were designated as ”positive” co-changed examples.
For Siamese network training, positive and negative file pairs were generated. Positive pairs were generated by pairing each anchor file with its respective co-changed files in the gold set. Negative pairs were generated from random sampling of files that are not related and are not in the same co-change group. These positive (+1) and negative (+0) labeled pairs were then employed as training data for the Siamese network, which was contrastive-loss-trained to drive positive pairs away from and negative pairs towards each other [33].

4.3.2.2 Model training

This constructed dataset was then used to train a Siamese network model to acquire file similarity in the context of co-changes. The network employed two equal parameter-sharing subnetworks each comprising two fully connected layers. The network consumes a pair of file embeddings and projects them to a lower space where semantically similar files mapped closer and different files mapped farther.

Contrastive loss was used to define the training goal, which was expressed as follows:

$$L(x_1, x_2, y) = y \cdot D^2 + (1 - y) \cdot \max(0, m - D)^2$$

Where D is the Euclidean distance between the two projected embeddings, $y \in \{0, 1\}$ is the similarity label, and m is a selected margin. This loss will punish the model for putting similar files too far away from each other and dissimilar files too close to one another. Training was conducted for 10 epochs with the Adam optimizer [51], where learning rate was 0.001. In each epoch, the model learned to tune its parameters in a way to minimize contrastive loss for all input pairs. The trained model was stored and then used in the recommendation step, where it needed to select the top-10 most similar files based on an initial changed file.

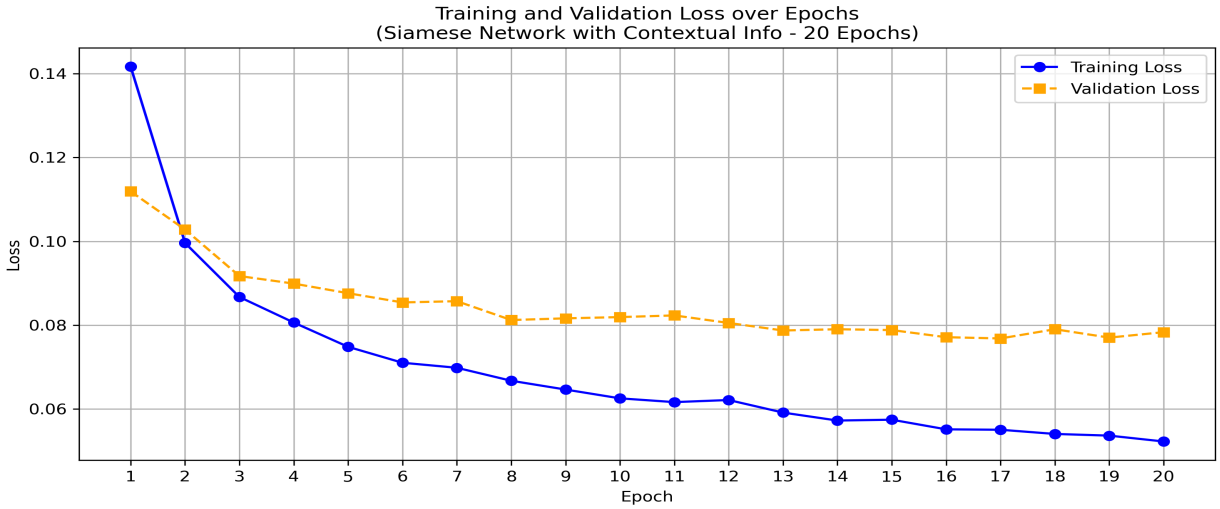


Figure 4.5: Training Loss over Epochs (Siamese with Contextual Info)

Figure 4.5 illustrates the training and validation loss curves of the Siamese network trained with contextual information and where semantic code embeddings were fetched through GraphCodeBERT. The x-axis is for training epochs and the y-axis for contrastive loss. Training loss drops sharply in the initial epochs—particularly between epoch 1 and epoch 2—and then goes on to maintain a steady decline throughout all 20 epochs, a strong sign of successful learning and convergence. Validation loss, however, fluctuates and levels off, which indicates that model generalization is not always improved by increased training. Small rises in validation loss (e.g., between epochs 6 and 7, and between epochs 17–18) indicate minor overfitting. This is also evident by the increasing gap between training and validation loss for later epochs.

Overall, the trend indicates that the application of contextual embeddings assists the model in obtaining faster and more stable training convergence, though generalization remains somewhat variable.

In order to understand better the computational resources used by the Siamese network's training pipeline, we timed the computer time and memory used during its most resource-intensive phases. The loading of the embeddings phase, which included parsing and importing precomputed file-level representations into memory, lasted 69.62 seconds and used around 187,788 KB of memory. Then the dataset construction step that produced the positive and negative training pairs from the golden set took 0.22 seconds with a slight boost in the memory usage to 188,188 KB, giving a sense of the lightness of this step. Model training, in which the Siamese network was contrastive loss trained for several epochs and validated on a held-out validation set for each epoch, was the second most memory-intensive task and took 30.06 seconds with peak memory of 257,684 KB because of batch handling, gradient calculation, and midpoint checks.

Lastly, the model serialization process, which is tasked with writing the trained model to disk, was only 0.11 seconds with the utilization of 257,780 KB, representing the memory retention achieved in training.

The entire pipeline for training took 100.02 seconds with the highest peak memory utilization of 257,780 KB. Although the maximum time was taken at the embedding loading step (69.62 seconds), model serialization step invoked highest memory usage because of in-memory representation of learnt weights and file I/O buffering. The results affirm computational viability of incorporation of the solution, thereby ensuring its utility for deployment in large-scale software analytics and recommendation systems.

4.3.3 Recommendation phase

4.3.3.1 Siamese Network-Based Recommendation

Once the Siamese network is trained to recognize similar and dissimilar pairs of files, it is used to recommend co-changed files for a given input. Its recommendation process begins by embedding all source files by using the learned model to obtain consistent, fixed-length vector representations. For each gold set bug report, first file from the gold set is taken as the initial changed file, and the objective is to recommend the top 10 likely co-changed files with it. The steps involved in this procedure are highlighted in algorithm 4.4.

The essence of this procedure is to compute pairwise distances (Euclidean distance) from the first file's embedding to other embeddings. They measure how semantically close other files are to the first file in the learned embedding space. The nearest ones are the recommendations.

Each prediction is evaluated against suggested files and ground truth from the gold set.

Algorithm 4.4: Siamese Network-Based Recommendation Process

Input: siamese_model : trained Siamese network
file_embeddings : embeddings of all files
gold_set : bug-wise grouped co-changed files

Output: top_10_candidates : top 10 co-change candidates per bug ID

```
Load and normalize all file embeddings
foreach bug_id in gold_set do
    anchor ← first file of bug_id
    ground_truth ← remaining co-changed files
    anchor_embedding ← generate embedding using siamese_model
    foreach file in file_embeddings do
        distance[file] ← compute distance between anchor_embedding and
          file_embedding
    ranked ← sort files by ascending distance
    top_10_candidates[bug_id] ← top 10 files from ranked
    evaluate the output against ground_truth
    record and save results for each bug_id
return top_10_candidates
```

4.3.3.2 ANN Recommendation using FAISS

In the final stage of the pipeline model, a recommendation system was created on the basis of Approximate Nearest Neighbor (ANN) search using the FAISS library. It successfully recommends semantically similar source code files by comparing their vector representations achieved during the training process of the Siamese network.

Initially, all the .pt embedding files (each of which represents a Java source file) were passed through the trained Siamese network to generate a transformed embedding using the forward_once method. These resulting vectors were stored in an in-memory matrix and also mapped to their corresponding normalized file names for later use.

The resulting embedding matrix was indexed by IndexFlatL2, a brute-force L2 distance search engine of FAISS, to support efficient top-k similarity queries. Once the index was constructed, each bug's "initial" file (the first file in the gold set) was used as a query to retrieve the top 10 nearest neighbors (excluding itself). The neighbors were compared against the ground truth files in the gold set to evaluate the recommendation accuracy.The process is highlighted in algorithm 4.5.

This approach leverages FAISS to dramatically speed up similarity retrieval in high-dimensional vector spaces at minimal cost in terms of accuracy, which makes it extremely well-suited for code search and recommendation at scale.

Algorithm 4.5: Approximate Nearest Neighbors using FAISS-Based Recommendation Process

Input: siamese_model : trained Siamese network
file_embeddings : raw embeddings of all files
gold_set : bug-wise grouped co-changed files

Output: top_10_candidates : top 10 co-change candidates per bug ID

foreach *file in file_embeddings* **do**
    normalize filename
    transformed_embedding ← siamese_model.forward_once(embedding)
    store transformed_embedding in matrix and record mapping
Build FAISS index from all transformed_embeddings
foreach *bug_id in gold_set* **do**
    normalize file paths
    query_file ← first file of bug_id
    query_embedding ← transformed embedding of query_file
    top_neighbors ← use FAISS to retrieve top 10 nearest files to query_embedding
    compare top_neighbors with ground truth
    record and save results for each bug_id
return *top_10_candidates*

4.4 Evaluation Method

To determine how well such a method assists developers in fully debugging buggy code, its effectiveness must be evaluated using reliable ground truth data, specifically the Gold set. The first line of the Gold set is the initially changed file, and subsequent lines are the ground truth (expected co-changed) files. This setup makes it possible to compare the recommended files point-wise with the actual co-change instances. Comparing with this ground truth data set, the approach has an empirical assessment of its precision and effectiveness in finding co-change relationships.

Chapter 5. Experiment

This chapter is divided into two main sections: the first outlines the experimental setup and the evaluation metric; the second presents the results and discussion, analyzing the effectiveness of each approach in identifying actual co-changing files.

5.1 Experimentation Setup

In this experiment, the first line of the Gold set is the initially changed file and it's treated as the developer's fixed bug code, as identified by the bug localization process, the remaining line of the Gold set is used as ground truth and used to evaluation the outputs. The initial changed file is excluded from both the set of actual co-changed files and the top-10 recommended files.

5.1.1 Evaluation Metrics

The evaluation process computes the accuracy with which the co-change recommendation system can retrieve related files for a target bug ID instance from the output generated in the output directory. For each JSON file, the system reads the recommended data and retrieves the ground truth files for a specific bug ID. The overall goal is to verify whether the top-10 recommendations contain at least one actual co-changed file. As the system processes every recommendation outcome, it accumulates the count of ground truth files that belong to the Gold set group and the count of them that were correctly accessed. The metrics are then aggregated according to the number of ground truth files per Gold Set—i.e., instances with one, two, or three-plus actual co-changes—to enable the examination of performance at different levels of co-change complexity.

As detailed in algorithm 5.6, the system loops over all JSON outputs, carries out file-level comparisons, maintains accurate recommendation by co-change group size, and produces a nicely formatted report.

To compare the performance across various levels of co-change complexity, a distribution-based success rate per group of Gold sets ordered by the number of ground truth co-changed files (not including the originally changed file) is calculated. For each group with n ground truth files, the system determines how many such files were ranked within top-10 recommendations. This analysis computes how many times 0, 1, ., through n correct files were recalled.

This is measured using the following formula:

$$\text{Success Rate}_{k,n}(\%) = \frac{\text{No. of Gold sets with exactly } k \text{ correct recommendations out of } n}{\text{Total number of Gold sets with } n \text{ ground truth files}} \times 100\%$$

Here, n is the number of actual co-changed files (excluding the initial changed file), and k is the number of them that were correctly predicted in the top-10 recommendation list. This metric allows the system to assess how reliably it retrieves not just some, but how many relevant files for Gold sets of varying sizes.

Apart from this distribution-based breakdown, the analysis also calculates the Overall Success Rate (OSR), as the proportion of cases in the whole dataset where there was at least one actual co-changed file correctly retrieved in top 10 recommendations. This is calculated as:

$$\text{OSR (\%)} = \frac{\text{No. of files with atleast one correct recommendation in Top-10}}{\text{Total number of output files in the gold set}} \times 100\%$$

This formula embodies the proportion of test cases where at least one ground truth file was successfully recommended, serving as a general indicator of the method's success and applicability. After testing all of the predictions, the result—i.e., distribution-based success rate and overall success rate—is compiled and placed in a structured Word document for inspection and reporting.

Algorithm 5.6: Analysis Report Generator Process

Input: input_dir : directory containing JSON recommendation output files

Output: analysis_report.docx : Word document summarizing co-change recommendation accuracy

```
Define the input directory containing JSON output files
Initialize counters: gt_counts, correct_predictions
foreach file in input_dir do
    json_data ← load JSON file
    gt_files ← extract ground truth files from json_data
    rec_files ← extract recommended files from json_data
    count_match ← number of rec_files found in gt_files
    update gt_counts and correct_predictions using ground truth file count
Create new Word document
Add title and introductory paragraph
Create table with:
    - Headers for ground truth counts and totals
    - Cells with percentage of correct predictions per ground truth count
Add summary statistics:
    - Total files analyzed
    - Distribution by ground truth count
    - Success rates per ground truth count
    - Overall success rate
Save the Word document to input_dir
```

5.2 Result and Discussion

5.2.1 Result

This subsection presents the evaluation outcomes of the proposed co-change recommendation methods. Each approach is assessed based on its effectiveness in retrieving actual co-changed files from the gold set.

Concept lattice Analysis report- concept lattice-based Approach

No of co-changes identified in the top 10	No. of Actual co-changed files excluding initial changed files																			
	1 (59)	2 (39)	3 (25)	4 (14)	5 (8)	6 (8)	7 (12)	8 (6)	9 (3)	10 (6)	11 (1)	12 (1)	13 (1)	14 (2)	15 (0)	16 (1)	17 (2)	18 (1)	19 (1)	20 (17)
0	25.42%	25.64%	4.0%	7.14%	12.5%		16.67%			50.0%										
1	74.58%	38.46%	64.0%	71.43%	50.0%	50.0%	25.0%	33.33%	33.33%	16.67%		100.0%				100.0%				11.76%
2		35.9%	20.0%	14.29%	12.5%	12.5%	25.0%	16.67%		16.67%			100.0%				50.0%			23.53%
3			12.0%	7.14%	25.0%	25.0%	16.67%	33.33%	33.33%					50.0%						17.65%
4							16.67%		33.33%					50.0%						5.88%
5						12.5%		16.67%		16.67%	100.0%						50.0%		100.0%	
6																		100.0%		11.76%
7																				5.88%
8																				5.88%
9																				11.76%
10																				5.88%
At least one Actual Co-change in the top 10, 174 out of 207 = 84.05%																				

Figure 5.1: Concept lattice analysis report- concept lattice-based recommendation

Figure 5.1 presents the results of the concept lattice-based recommendation approach, where the columns represent the number of actual co-changed files (excluding the initially changed file) in each instance, ranging from 1 to 20. The rows indicate the number of these actual co-changes that were successfully identified within the top 10 recommendations. Each cell in the table shows the percentage of cases where a given number of actual co-changes were detected among the top 10 ranked recommendations, allowing for an evaluation of how effectively the approach captures co-change relationships. The total number of instances corresponding to each column is provided in parentheses for reference.

The experiments show that the concept lattice-based recommendation algorithm was able to recommend at least one real co-change within the top 10 recommendations in 174 of 207 instances, which yields an overall success ratio of 84.05%. This finding illustrates the applicability of the concept lattice for modeling semantically similar co-changes among software products. The number of observed co-changes demonstrates that larger numbers of true co-changes are correlated with an increasing chance of having at least one within the top 10. However, variability exists across different instances, which indicates that although the model is very good in the general case, sensitivity to particular structural or semantic contexts can cap performance on boundary cases. This is both a confirmation of the robustness of the methodology and an opportunity for hybrid or adaptive optimizations to enhance consistency.

Dependency Graph Analysis report- Dependency graph-based Approach

No of co-changes identified in the top 10	No. of Actual co-changed files excluding initial changed files																		
	1 (59)	2 (39)	3 (25)	4 (14)	5 (8)	6 (8)	7 (12)	8 (6)	9 (3)	10 (6)	11 (1)	12 (1)	13 (1)	14 (2)	15 (0)	16 (1)	17 (2)	18 (1)	19 (1)	20 (17)
0	71.18 %	56.43 %	56.0 %	72.0 %	50.0 %	25.0 %	75.0 %	50.0 %		100.0 %	100.0 %			50.0%			100.0 %	100.0 %		47.05 %
1	28.82 %	38.57 %	36.0 %	20.0 %	50.0 %	37.5 %	8.3%	33.33 %	33.33 %					50.0%						5.88%
2		5.0%	8.0%			12.5 %			33.33 %			100.0 %								
3				8.0%		12.5 %	16.7 %	16.67 %					100.0 %			100.0 %				17.64 %
4						12.5 %			33.33 %											11.76 %
5																			100.0 %	5.88%
6																				5.88%
7																				
8																				
9																				
10																				5.88%
At least one Actual Co-change in the top 10, 82 out of 207 = 39.6%																				

Figure 5.2: Dependency graph analysis report-dependency graph-based recommendation

Figure 5.2 shows the results of the dependency graph-based approach indicate a notably lower success rate in identifying co-change relationships compared to the concept lattice-based method. Specifically, the analysis reveals that at least one actual co-change was identified in the top 10 recommendations for 82 out of 207 cases, resulting in an overall success rate of 39.6%. This suggests that while the dependency graph-based approach captures some co-change relationships, its predictive effectiveness is significantly weaker.

Furthermore, the distribution of recovered co-changes shows that even when the number of actual co-changes increases, the likelihood of recovering at least one within the top 10 is not stable. In certain cases, even when there were a number of co-changes, the method failed to retrieve them effectively. The findings refer to potential limitations in the method's reliance on method call dependencies, which do not always encode co-change patterns accurately. Analytically, this underperformance is reflective of the structural rigidity of dependency graphs that may not capture semantically related but syntactically independent pieces of code. To this end, the results point to the importance of refinement—e.g., via hybrid methods—for improving predictive power and flexibility in the face of diverse software change scenarios.

Combined Analysis report- Combined-based Approach

No of co-changes identified in the top 10	No. of Actual co-changed files excluding initial changed files																		
	1 (59)	2 (39)	3 (25)	4 (14)	5 (8)	6 (8)	7 (12)	8 (6)	9 (3)	10 (6)	11 (1)	12 (1)	13 (1)	14 (2)	15 (0)	16 (1)	17 (2)	18 (1)	19 (1)	20 (17)
0	15.25 %	7.69%	8.0%	7.14%	12.5%		16.67 %			33.33 %										
1	84.74 %	48.71 %	64.0%	57.14 %	62.5%	50.0%	25.0%	16.67 %		33.33 %										11.76 %
2		43.58 %	16.0%	14.28 %			25.0%	33.33 %	33.33 %	16.66 %							50.0%			11.76 %
3			12.0%	21.43 %	12.5%	25.0%	16.67 %	33.33 %	33.33 %			100.0 %	100.0 %	50.0 %						29.41 %
4						12.5%	16.67 %		33.33 %							100. 0%				5.88%
5					12.5%	12.5%		16.67 %		16.66 %	100.0 %			50.0 %			50.0%			
6																		100.0 %		5.88%
7																				11.76 %
8																				5.88%
9																			100.0 %	5.88%
10																				11.76 %
At least one Actual Co-change in the top 10, 187 out of 207 = 90.33%																				

Figure 5.3: Analysis report of combined-based recommendation

Figure 5.3 illustrates that the combined approach demonstrates a notable improvement over each of the individual methods when applied separately. Specifically, at least one actual co-change was identified in the top 10 recommendations for 187 out of 207 cases, yielding an overall success rate of 90.33%. This increase indicates that integrating both methods effectively enhances the predictive capability, capturing a broader range of co-change patterns.

The findings suggest that the combined approach addresses the limitations of the individual approaches, blending the strengths of both. The higher rate of success implies that concept lattice-based and dependency graph-based dependencies complement each other to build a more complete model of co-change prediction. Analytically, such collaboration demonstrates that structural and semantic indicators—when integrated—can provide a richer background on which to model co-evolutionary behavior. Furthermore, greater consistency in co-change detection across varying amounts of real changes again solidifies the robustness of this hybrid method. The practical implications of these results are highly real in actual use in real-world scenarios since they show the ability of the method to increase developer support in software maintenance and bug fixing by targeting the best-fitting file recommendations with greater accuracy.

Siamese Analysis report- Siamese Network-based Approach (without context)

No of co-changes identified in the top 10	No. of Actual co-changed files excluding initial changed files																		
	1 (59)	2 (39)	3 (25)	4 (14)	5 (8)	6 (8)	7 (12)	8 (6)	9 (3)	10 (6)	11 (1)	12 (1)	13 (1)	14 (2)	15 (0)	16 (1)	17 (2)	18 (1)	19 (1)	20 (17)
0	89.83 %	84.61 %	80.00 %	85.71 %	87.50 %	50.00 %	75.00 %	100.00 %	33.33 %	50.00 %	100.0 %	100.0 %	100.0%			100. 0%	50.00 %		100.00 %	11.76 %
1	10.17 %	15.38 %	20.00 %	7.14%	12.50 %	37.50 %	8.33%		66.66 %	50.00 %				100.0 %						23.52 %
2				7.14%		12.50 %	16.66 %										50.00 %	100.00 %		29.41 %
3																				
4																				17.64 %
5																				5.88%
6																				
7																				
8																				
9																				5.88%
10																				5.88%
At least one Actual Co-change in the top 10, 51 out of 207 = 24.63%																				

Figure 5.4: Analysis report of Siamese network-based recommendation approach without context

Figure 5.4 presents the performance values of the Siamese network-based recommendation approach without context. Like in the previous evaluation tables, each cell represents the percentage of instances in which a given number of actual co-changed files were correctly identified within the top 10 recommendations. The numbers in the parentheses at the column headers represent the number of test cases with the corresponding number of co-changed files.

Results are a very low success ratio relative to the other approaches. Out of 207 test cases, the Siamese model with no contextual input succeeded to detect at least one correct co-change among the top 10 suggestions in 51 cases with an overall success rate of 24.63%. The majority of the predictions are classified in the zero-match class, reflecting the limitation of the approach when contextual file correlations are not available.

While these results confirm the model's ability to capture certain patterns of similarity, they also indicate an inherent weakness—i.e., absence of contextual grounding. Without including inter-file relationships, the model cannot meaningfully capture the fine-grained dependencies that govern real co-changes. This result highlights the importance of contextual information in learning meaningful co-change behavior and suggests that stand-alone embeddings may be insufficient for reliable recommendation. In practice, these limitations reduce the effectiveness of such a method for development maintenance tasks to obtain correct and actionable co-change suggestions.

Siamese Analysis report- Siamese Network-based Approach (with context)

No of co-changes identified in the top 10	No. of Actual co-changed files excluding initial changed files																		
	1 (59)	2 (39)	3 (25)	4 (14)	5 (8)	6 (8)	7 (12)	8 (6)	9 (3)	10 (6)	11 (1)	12 (1)	13 (1)	14 (2)	15 (0)	16 (1)	17 (2)	18 (1)	19 (1)	20 (17)
0	67.79 %	41.03 %	32.00 %	21.42 %	25.00 %		25.00 %													
1	32.20 %	51.28 %	40.00 %	50.00 %	25.00 %	50.00 %	33.33 %	16.66 %	33.33 %	66.66 %		100.%	100.0%							
2		7.69%	16.00 %	28.57 %	37.50 %	50.00 %	41.66 %	33.33 %	33.33 %	16.66 %	100.%						50.00 %	100.00 %		5.88%
3			12.00 %		12.50 %				33.33 %					100.0 0%			50.00 %		100.00 %	17.64 %
4								16.66 %												5.88%
5								33.33 %		16.66 %						100. 0%				17.64 %
6																				11.76 %
7																				5.88%
8																				5.88%
9																				11.76 %
10																				17.64 %
At least one Actual Co-change in the top 10, 135 out of 207 = 65.21%																				

Figure 5.5: Analysis report of Siamese network-based recommendation approach with context

Fig. 5.5 shows the results related to the effectiveness of the recommendation method adopted through a Siamese network after incorporating contextual information. This model variant leverages contextualized code representations from GraphCodeBERT, which capture both structural and semantic aspects of source code.

The arrival of contextual embeddings has witnessed significant improvement over the models that didn't benefit from contextual knowledge. Specifically, the contextual Siamese model effectively located at least one co-changed file from the top 10 recommendations in 135 out of 207 test cases, yielding a general success rate of 65.21%. This improved performance shows the ability of the model to recognize more sophisticated code semantics and interdependencies, particularly for several co-changed files.

Other than the descriptive accuracy rate, these results are significant of the necessity for the integration of context-aware representations in models for software maintenance. The performance gain is suggestive that semantic and structural cues picked up by GraphCodeBERT are dense cues for modeling co-change behavior that otherwise would be underpicked under context-agnostic settings. Furthermore, boosted generalizability across varying test cases is suggestive of a more scalable and robust method. In general, these findings emphasize the benefit of using transformer-based contextual embeddings in the task of recommending co-change and enhancing accuracy in detecting and ordering relevant files.

Siamese Analysis report- ANN-based Approach (20 epoch)

No of co-changes identified in the top 10	No. of Actual co-changed files excluding initial changed files																		
	1 (59)	2 (39)	3 (25)	4 (14)	5 (8)	6 (8)	7 (12)	8 (6)	9 (3)	10 (6)	11 (1)	12 (1)	13 (1)	14 (2)	15 (0)	16 (1)	17 (2)	18 (1)	19 (1)	20 (17)
0	57.62 %	35.90 %	24.00 %		37.50 %		16.66 %				100.00 %									
1	42.37 %	38.46 %	36.00 %	35.71 %	12.50 %	50.00 %	41.66 %			16.66 %			100.0%							
2		25.64 %	28.00 %	14.28 %	37.50 %	25.00 %	25.00 %	33.33 %	66.66 %	33.33 %		100.00 %		100.0 0%			100.00 %			
3			12.00 %	50.00 %	12.50 %	12.50 %	16.66 %	16.66 %	33.33 %	16.66 %						100. 0%		100.00 %	100.00 %	11.69 %
4						12.50 %		16.66 %		16.66 %										23.52 %
5								33.33 %												5.88%
6										16.66 %										17.64 %
7																				17.64 %
8																				5.88%
9																				17.64 %
10																				11.69 %
At least one Actual Co-change in the top 10, 147 out of 207 = 71.01%																				

Figure 5.6: Analysis report of ANN-based recommendation approach with context

Figure 5.6 depicts the Siamese network performance utilizing contextual embeddings and Approximate Nearest Neighbor (ANN) search. The model achieved a high success rate of 71.01%, indicating at least one co-changed file among the top 10 suggestions in 147 out of the 207 cases of bugs. This represents a significant boost over low-epoch and non-contextual variants, demonstrating once more the power of longer training and more semantically aware input. The enhancement shows that the model, trained further with more rounds, acquires more discriminative representations which, together with effective ANN-based ranking, result in better accuracy and consistency in co-change prediction.

While quantitatively strong, the broader significance of the results lies in demonstrating how deeper contextual knowledge combined with scalable search is beneficial for learning-based recommendation systems. Combining transformer-based embeddings with optimized retrieval presents a promising avenue for enhancing software evolution support tools. These findings indicate that greater model capacity, contextual understanding, and retrievable approaches can work together to provide higher-quality large software system suggestions—most particularly in complex, realistic bug-fixing scenarios where accuracy is critically important.

5.2.2 Discussion

The outcomes of the three main methods—concept lattice-based, dependency graph-based, and combined approach—report different levels of achievement in the detection of co-change relations. The concept lattice-based technique achieved a high success rate of 84.05%, which is a testament to its effectiveness at detecting implicit relationships among co-changing files. In contrast, the dependency graph-based model achieved a considerably lower success rate of 39.6%, which indicates that the use of method call links alone is insufficient for modeling co-change behavior with high accuracy. This weakness might, however, be overcome by incorporating a wider variety of structural dependencies like class hierarchies, import relations, or data flows. The hybrid method that combines both dependency graph and concept lattice data achieved the highest success rate of 90.33%, confirming again that multi-source static data provides greater predictive power.

In comparison to these approaches, the Siamese network-based models varied with the use of contextual information and without contextual information. The most straightforward Siamese model trained in the absence of contextual information score a success rate of only 24.63%, which was worse than all the static baselines by a large margin. This shows the limitation of using only the vectorized AST path contexts in co-change relationship detection.

But the inclusion of contextual information via GraphCodeBERT significantly boosted the model's performance. The Siamese model trained using contextual information achieved an improved success rate of 65.21%, which testifies to the effectiveness of semantic and structural representations of code in learning patterns of co-change.

Second, when Approximate Nearest Neighbor (ANN) search was applied to the Siamese model trained using contextual embeddings resulted in a better performance of 71.01%. The result is close to the concept lattice-based baseline, and much better than the dependency graph-based method.

Despite their valuable empirical findings, this co-change recommendation techniques like those proposed by Revelle et al. and Mathur et al. [26, 27] have been described as computationally costly and weakly validated, especially in dynamic or unforeseen settings. The static analysis techniques of this study (concept lattice and dependency graph) address the validation gap head-on by demonstrating generalization over a 207-case test. Moreover, computational profiling also testifies to their feasibility—running in less than one minute with minimal memory overhead—thereby overcoming cost concerns. On the learning side, the underperformance of the simple Siamese network (24.63%) points to a fundamental shortcoming: when vectorized AST path-contexts lack semantic richness and cannot capture cross-file dependencies. Such a limitation is all the more relevant because software evolution is often based on indirect or semantic code relationships. The counterintuitive outcome is addressed by the syntactic and shallow character of AST path-contexts, limiting the capacity of the model to recognize underlying patterns of change. The addition of contextual embeddings in GraphCodeBERT addresses this limitation through aligning learned representations with actual co-change behaviors.

However, one can note that while contextual Siamese models performed better than their non-contextual counterpart (71.01% vs 24.63%), they were still behind both the concept lattice-based (84.05%) and combined (90.33%) approaches. The performance gap is mainly because of limitations in the gold set used for training the Siamese model. In this setup, the model is being trained to map the first (anchor) file to its corresponding co-changed files, but the size and scale of this dataset were too low to expose the network to a wide enough range of co-change patterns. Since co-change behavior varies extremely across projects and involves subtle or domain-specific dependencies, a small training set restricts the model's ability to generalize to new, unseen examples. This limitation highlights the importance of expanding and diversifying the training data rather than relying on model depth or encoder complexity exclusively.

Briefly, while both the concept lattice-based (84.05%) and integrated solution (90.33%) achieved the highest static accuracy, the Siamese network with contextual embeddings saw a highly promising 71.01% rate of success. While slightly less, this learning-based solution provides a scalable, generalizable one that adapts to unseen data and gets better with repeated retraining. These findings suggest that hybrid solutions—combining static structure with learned semantics—are best geared to advance the state of co-change recommendation in modern software engineering.

The following responses are derived from the experimental results obtained across all evaluated methods. Each question is answered based on empirical evidence and comparative analysis of the evaluated co-change recommendation approaches.

RQ1 *Can Dependency graph and Concept lattice be used to identify co-changing entities from the source code?*

Yes,Concept lattice analysis assists developers in identifying additional co-changing entities that may need to be modified to fully resolve a bug initially located by bug localization techniques. With at least one real co-change found in the top 10 recommendations in 84.05% of situations, the concept lattice-based method shows a high success rate. This suggests that meaningful relationships between software entities based on shared attributes can be captured by concept lattice analysis. In a similar vein, the dependency graph-based method, which makes use of method calls, has a lower but still noteworthy success rate of 39.6%. This implies that although dependency graphs are capable of detecting co-change links, they are not as effective as concept lattice analysis, maybe as a result of implicit dependencies being overlooked.

RQ2 *Does combining concept lattice and dependency graph based outputs improve the recommendation system?*

Yes, the recommendation system is greatly enhanced by merging results based on dependency graph and concept lattice. The combined technique outperforms both independent methods, achieving an 90.33% success rate in detecting at least one true co-change in the top 10 recommendations. This illustrates how combining concept-based links with structural dependencies from method calls improves the precision and resilience of co-change predictions, using the advantages of both approaches to create a more potent recommendation system.

RQ3 *Do the concept lattice-based, dependency graph-based, and combined approaches outperform the Siamese network in co-change recommendation tasks?*

Yes, all three conceptual approaches—concept lattice-based, dependency graph-based, and combined—are better than the Siamese network trained not taking contextual information into consideration. Among these, the combined approach is best with the greatest percentage of success rate, outperforming both machine learning implementations. Moreover, even the concept lattice-based approach is better than the Siamese model trained on contextual embeddings, demonstrating the strength of structural and conceptual heuristics compared to learned representations in this experiment.

Chapter 6. Conclusion and Future Work

6.1 Conclusion

In this work, a co-change recommendation system was proposed through concept lattice construction and dependency graph analysis. The hybrid approach, by taking advantage of conceptual relationship and structural dependencies, achieved very good performance in co-change relationship detection. The combined method particularly achieved a notable success rate of 90.33%, outperforming the concept lattice-based (84.05%) and dependency graph-based (39.6%) approaches when used individually. The results confirm that the static analysis-based approaches—particularly the combined and concept lattice-based ones—provided more precise suggestions than the context-aware Siamese network. This confirms that, in this experimental scenario, conceptual approaches can be effective for co-change recommendations.

In comparison, the Siamese network machine learning method required augmented contextual representations from pretrained models like GraphCodeBERT to approach similar performance. While the context-aware Siamese network assisted by Approximate Nearest Neighbor (ANN) search achieved a competitive success rate of 71.01%, it was still behind the primary hybrid method. Moreover, the baseline Siamese network without contextual data performed significantly worse (24.63%), reflecting its dependence on outside semantic information. Overall, the results support that the primary proposed techniques—when thoughtfully combined—are more powerful and robust than the machine learning alternatives examined in this research when it comes to idetifying the co-change without rely on contextual information.

This work effectively meets its initial goals set out in 1.3. First, by building a concept lattice and dependency graph purely from source code elements like classes, methods, and attributes, the system showed that significant static structures can be produced without requiring historical or context information. Second, these representations were applied effectively to detect co-changing items in bug fix instances to verify the system's ability to suggest files that should change together. Third, the proposed methods universally produced top-10 co-change file suggestions based on the initially changed file, and their performance was measured quantitatively. Lastiy, comparison experiments indicated that the proposed static methods outperformed different configurations of the Siamese network model, particularly those without contextual embeddings.

For actual-world implementation, this system could be used in software development environments as a light-weight plugin that performs static analysis during code review or bug triaging iterations. With little computational cost and no need to access history of changes, the approach is highly suited for early-stage projects or poorly documented legacy codebases.

6.2 Future Work

Future work may explore enriching the concept lattice with static source code characteristics as well as history metadata of past co-change. Adding this second layer of data may further improve the conceptual relation granularity and raise the co-change prediction accuracy. Similarly, the graph of dependencies can be strengthened by extending its scope beyond method calls to also include other structural dependencies such as class hierarchy, and import dependencies, so that it represents a larger aspect of software architecture. And there is future work possible on scaling up the system to accommodate other programming languages and dynamically varying weights in the hybrid approach based on project-specific factors.

References

[1] Anushree Agrawal and Rakesh K Singh. Predicting co-change probability in software applications using historical metadata. *IET Software*, 14(7):739–747, 2020.

[2] Sue Black. Deriving an approximation algorithm for automatic computation of ripple effect measures. *Information and Software Technology*, 50(7-8):723–736, 2008.

[3] Sue Black. Computing ripple effect for software maintenance. *Journal of software maintenance and evolution: research and practice*, 13(4):263–279, 2001.

[4] Nashat Mansour and Hani Salem. Ripple effect in object oriented programs. *Journal of Computational Methods in Sciences and Engineering*, 6(s1):S23–S32, 2006.

[5] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on software engineering*, 31(6):429–445, 2005.

[6] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[7] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 4:1–4:11. ACM, 2011.

[8] Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[9] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[10] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 234–243, 2007.

[11] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *2009 IEEE international conference on software maintenance*, pages 351–360. IEEE, 2009.

[12] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 689–699, 2014.

[13] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[14] Chakkrit Tantithamthavorn, Surafel Lemma Abebe, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology*, 102:160–174, 2018.

[15] Florian Garnier, Robert A. Buchmann, and Martin Monperrus. How effective are information retrieval-based bug localization techniques on different programming languages? In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 134–144. IEEE, 2016.

[16] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.

[17] Kavya Shree NS and MN Pushpalatha. A survey of the bug localisation techniques. *International Journal of Software Engineering and Its Applications*, 12(2):17–30, 2018.

[18] Tien-Duy B Le, Richard J Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 579–590, 2015.

[19] Yuchen Huang et al. S-buglocator: A semantic-aware deep learning approach to bug localization. *Empirical Software Engineering*, 27(4):1–35, 2022.

[20] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings. 26th international conference on software engineering*, pages 491–500. IEEE, 2004.

[21] Lionel C Briand, Jurgen Wust, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 475–482. IEEE, 1999.

[22] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 190–198. IEEE, 1998.

[23] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*, 30(9):574–586, 2004.

[24] Igor Scaliante Wiese, Reginaldo Ré, Igor Steinmacher, Rodrigo Takashi Kuroda, Gustavo Ansaldi Oliva, Christoph Treude, and Marco Aurélio Gerosa. Using contextual information to predict co-changes. *Journal of Systems and Software*, 128:220–235, 2017.

[25] Malcom Gethers and Denys Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *2010 IEEE international conference on software maintenance*, pages 1–10. IEEE, 2010.

[26] Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical software engineering*, 16:773–811, 2011.

[27] Neeraj Mathur, Sai Anirudh Karre, and Y Raghu Reddy. Grouping semantically related change-sets to enhance identification of logical coupling. In *SEKE*, pages 618–777, 2019.

[28] WILLE Rudolf. Restructuring lattice theory: an approach based on hierarchies of concept. In *Symposium. on. Ordered Sets, 1982*, 1982.

[29] Franjo Škopljanac Mačina and Bojan Blaškovic. Formal concept analysis - overview and applications. *Procedia Engineering*, 69:1258–1267, 2014.

[30] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Berlin/Heidelberg, 1999.

[31] Søren Enevoldsen, Kim G. Larsen, Anders Mariegaard, and Jiří Srba. Dependency graphs with applications to verification. *International Journal on Software Tools for Technology Transfer*, 22:635–654, 2020.

[32] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a ”siamese” time delay neural network. In *Advances in Neural Information Processing Systems*, volume 6, 1994.

[33] Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005.

[34] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 815–823, 2015.

[35] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Communications of the ACM*, volume 51, pages 117–122. ACM, 2008.

[36] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[37] Anaytul Islam, Moksedul Islam, Manishankar Mondal, Banani Roy, Chanchal Roy, and Kevin Schneider. Detecting evolutionary coupling using transitive association rules. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 113–122. IEEE, 2018.

[38] Anushree Agrawal and Rakesh Kumar Singh. Identification of co-changed classes in software applications using software quality attributes. *Journal of Information Technology Research (JITR)*, 13(2):110–128, 2020.

[39] Manishankar Mondal, Banani Roy, Chanchal K Roy, and Kevin A Schneider. Historank: History-based ranking of co-change candidates. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 240–250. IEEE, 2020.

[40] Daihong Zhou, Yijian Wu, Lu Xiao, Yuanfang Cai, Xin Peng, Jinrong Fan, Lu Huang, and Heng Chen. Understanding evolutionary coupling by fine-grained co-change relationship analysis. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 271–282. IEEE, 2019.

[41] Nemitari Ajienka and Andrea Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. *Journal of Systems and Software*, 134:120–137, 2017.

[42] Nemitari Ajienka, Andrea Capiluppi, and Steve Counsell. An empirical study on the interplay between semantic coupling and co-change of software classes. *Empirical Software Engineering*, 23:1791–1825, 2018.

[43] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical software engineering*, 14:5–32, 2009.

[44] Melaeke Serawit and Surafel Lemma Abebe. Co-change recommendation using formal concept analysis. AAiT, 2022.

[45] Hamdi Abdurhman Ahmed and Jihwan Lee. Fcp2vec: Deep learning-based approach to software change prediction by learning co-changing patterns from changelogs. *Applied Sciences*, 13(11):6453, 2023.

[46] Igor Scaliante Wiese, Rodrigo Takashi Kuroda, Igor Steinmacher, Gustavo Ansaldi Oliva, Reginaldo Ré, Christoph Treude, and Marco Aurelio Gerosa. Pieces of contextual information suitable for predicting co-changes? an empirical study. *Software Quality Journal*, 27:1481–1503, 2019.

[47] Yiping Jia, Safwat Hassan, and Ying Zou. Enhancing software maintenance: A learning to rank approach for co-changed method identification. *arXiv preprint arXiv:2411.19099*, 2024.

[48] Rahman and Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proc. ESEC/FSE*, page 11, 2018.

[49] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40:1–40:29, 2019.

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

[51] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[52] Daya Guo, Shaoquan Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Nan Duan, Alexey Svyatkovskiy, Shuo Fu, Jiang Bian, Jian Yin, et al. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*, 2021.