Cairo University
Faculty of Engineering
Department of Computer Engineering

Intellitest

A Graduation Project Report Submitted

to

Faculty of Engineering, Cairo University

in Partial Fulfillment of the requirements of the degree

of

Bachelor of Science in Computer Engineering.

Presented by

Yasmin Hashem Niazy
Bassant Hisham Mohamed
Sohad Hossam Edlin
Yasmeen Zaki Nossair

Supervised by

Dr.Nevin Darwish

December 28th, 2023

All rights reserved. This report may not be reproduced in whole or in part, by photocopying or other means, without the permission of the authors/department.

Abstract

The increasing complexity of software systems demands effective methods for understanding and maintaining source codes. In this context, traceability links between software artifacts, such as requirements, code, and documentation, play a crucial role in facilitating software comprehension, maintenance, and evolution. This project focuses on developing a comprehensive framework for automated trace link generation between use case documents and Java code files, leveraging machine learning techniques and natural language processing (NLP) tools.

The project comprises several modules, each serving a specific purpose in the traceability link generation process. The **Data Collection Module** retrieves and preprocesses relevant data, including use case documents, Java code files, and associated metadata. The **Feature Extraction Module** extracts features from both textual and code-based sources, capturing essential information for link prediction. Subsequently, the **Model Prediction Module** applies machine learning algorithms, such as Random Forest, to predict trace links based on the extracted features, as well as the **Deep Learning Prediction Module** for bench marking the machine learning outputs. Additionally, the **Bug Localization Module** aims to locate buggy versions of code files by analyzing bug reports and version control data.

Throughout the development process, various software tools and technologies are employed, including Visual Studio Code, Python, Jupyter Notebook, TensorFlow, and TreeSitting. These tools facilitate tasks such as code editing, data analysis, machine learning model training, and parsing source code. Furthermore, libraries such as NumPy, Pandas, and Scikit-learn provide essential functionalities for data manipulation, numerical computing, and machine learning tasks.

The proposed framework contributes to the automation of traceability link generation, reducing manual effort, and improving software maintenance efficiency. By establishing trace links between use case documents and code files, developers can better understand system requirements, track changes, and ensure consistency between software artifacts. Overall, this project aims to enhance software comprehension and maintenance processes, ultimately leading to the development of more reliable and maintainable software systems.

الملخص

يتطلب التعقيد المتزايد لأنظمة البرمجيات أساليب فعالة لفهم رموز المصدر والحفاظ عليها. في هذا السياق، تلعب روابط التتبع بين عناصر البرامج، مثل المتطلبات والتعليمات البرمجية والوثائق، دورًا حاسمًا في تسهيل فهم البرامج وصيانتها وتطويرها. يركز هذا المشروع على تطوير إطار عمل شامل لإنشاء رابط التتبع الآلي بين مستندات حالة الاستخدام وملفات كود Java، والاستفادة من تقنيات التعلم الآلي وأدوات معالجة اللغة الطبيعية (NLP). يتألف المشروع من عدة وحدات، تخدم كل منها غرضًا محددًا في عملية إنشاء رابط التتبع. تقوم وحدة جمع البيانات باسترداد البيانات ذات الصلة ومعالجتها مسبقًا، بما في ذلك مستندات حالة الاستخدام وملفات كود Java والبيانات التعريفية المرتبطة بها. تستخرج وحدة استخراج الميزات الميزات من المصادر النصية والمصادر المستندة إلى التعليمات البرمجية، وتلتقط المعلومات الأساسية للتنبؤ بالارتباط. بعد ذلك، تطبق وحدة التنبؤ النموذجية خوارزميات التعلم الآلي، مثل Random Forest، للتنبؤ بروابط التتبع بناءً على الميزات المستخرجة، بالإضافة إلى وحدة التنبؤ بالتعلم العميق لوضع علامات على مخرجات التعلم الآلي. بالإضافة إلى ذلك، تهدف وحدة توطين الأخطاء إلى تحديد موقع الإصدارات التي بها أخطاء من ملفات التعليمات البرمجية من خلال تحليل تقارير الأخطاء وبيانات التحكم في الإصدار. طوال عملية التطوير، يتم استخدام أدوات وتقنيات برمجية مختلفة، بما في ذلك Visual Studio Code وPython وJupyter Notebook وTensorFlow وTreeSitting. تسهل هذه الأدوات مهام مثل تحرير التعليمات البرمجية، وتحليل البيانات، والتدريب على نموذج التعلم الآلي، وتحليل التعليمات البرمجية المصدر. علاوة على ذلك، توفر المكتبات مثل NumPy وPandas وScikit-learn وظائف أساسية لمعالجة البيانات والحوسبة الرقمية ومهام التعلم الآلي. يساهم الإطار المقترح في أتمتة إنشاء روابط التتبع، وتقليل الجهد اليدوي، وتحسين كفاءة صيانة البرامج. من خلال إنشاء روابط تتبع بين مستندات حالة الاستخدام وملفات التعليمات البرمجية، يمكن للمطورين فهم متطلبات النظام بشكل أفضل وتتبع التغييرات وضمان الاتساق بين عناصر البرنامج. بشكل عام، يهدف هذا المشروع إلى تعزيز فهم البرامج وعمليات الصيانة، مما يؤدي في النهاية إلى تطوير أنظمة برمجية أكثر موثوقية وقابلة للصيانة.

ACKNOWLEDGMENT

We would like to sincerely thank our wonderful supervisor, Dr. Nevin Darwish, for all of her hard work this year in making sure we have the best product possible and that we stick to all deadlines in order to have a product that can benefit both the software industry and our future.

We extend our gratitude to Eng. Omar Samir for his unwavering assistance, vast knowledge, and generous sharing of it with us during the project.
Finally, but just as importantly, we want to express our gratitude to our parents for putting up with us and our workload all year long.

It was a long journey with a beautiful ending, so we are deeply grateful to everyone who was in on it, including our friends and family, our doctors and TAs all along the 4 years, and everyone we met along the way, offering us help technically, emotionally, and with any sort of support.

Table of Contents

2.1. Target Audience 20
2.2.1. VisualStudio [28] 20
2.2.2. Requirements Traceability Links[25] 21
2.2.3. SymFlower[26] 21
2.3. Business Model 22
This section starts by identifying distinguished features that give Intellitest its competitive advantage and then proceeds to give the business case. 22
2.3.1 Competitive Advantage: 22
2.3.2 Business Case: 22
2.3.3 Subscriptions Planning: 23
1. Target Market Size: 23
2. Market Share and Growth: 23
3. User Base Growth: 24
4. Revenue Projection: 25
2.3.4 Financial Analysis: 25
Revenue Projection: 25

Cost Analysis: 25
Summary: 26
Chapter 3: Literature Survey **26**
3.1 Technical Background Knowledge 26
3.1.1 Natural Language Processing (NLP) [1] 27
3.1.2 Temporal Properties [1] 27
3.1.3 Dynamic selection [9] 27
3.1.4 Naïve Bayes [8] 27
3.1.5 Logistic Regression [9] 28
3.1.6 K-Nearest Neighbor [9] 28
3.1.7 Area under ROC(Receiver Operating Characteristic) curve [22] 28
3.1.8 Software Fault Prediction [9] 28
3.1.9 Source Code Modeling [4] 29
3.1.10 Support vector machine(SVM) [7] 31
3.1.11 Vectorization methods [17] 31
3.1.12 Multi-layer Perceptron [21] 32
3.1.13 Inverse Collection Term Frequency(ICTF) [24] 32
3.1.14 vector space model [23]: 32
A term-by-document matrix is used in the VSM to represent software artifacts and documents. The term frequency-inverse document frequency (TF-IDF) of each member of the matrix expresses the term's relevance within the document and corpus. Remember that every document is a vector, hence we can use cosine similarity to calculate how similar two documents are to one another as follows: 32
3.1.15 Latent Semantic Analysis(LSA)(referred to in code in appendix E.1) [23]: 32
3.1.16 Latent Dirichlet Allocation(LDA) (referred to in code in appendix E.1) [23]: 33
3.1.17 Jensen-Shannon [23]: 33
3.1.18 Entropy [24] 33
3.1.19 Okapi BM25 [23]: 33
3.1.20 Language Model with Dirichlet (LM-Dirichlet), and Jelinek-Mercer smoothing (LM-JM) [23]: 34
3.1.21 Correlation-based Feature Subset Selection (CFS-Subset)[23]: 34
3.1.22 Pearson’s correlation [23]: 34
3.1.23 Gain Ratio [23]: 34
3.1.24 information gain [23]: 35
3.1.25 Symmetrical Uncertainty [23]: 35
3.1.26 Synthetic Minority Oversampling Technique [23]: 35
3.1.27 Random Majority Undersampling [23]: 35
3.1.28 Pointwise mutual information (PMI) [24]: 36
3.1.28 Similarity Collection Query (SCQ) [24]: 37
3.2 Comparative Study of Previous Work 37
3.2.1 “Call Me Maybe: Using NLP to Automatically 37
Generate Unit Test Cases Respecting Temporal 37
Constraints” [1] 37
3.2.2 “An Overview of Quality Metrics Used in Estimating Software Faults” [6] 38

3.2.3 “Software fault prediction (SFP) based on the dynamic selection of learning technique: findings from the eclipse project study” [9] 40
3.2.4 “Constructing Test Cases Using Natural Language Processing” [10] 43
3.2.5 “Fast Static Analyses of Software Product Lines — An Example With More Than 42,000 Metrics” [11] 43
3.2.6 “Unified Abstract Syntax Tree Representation Learning for Cross-Language Program Classification” [12] 43
3.2.7 “Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction” [13] 44
3.2.8 “Predicting Defects for Eclipse” [14] 46
3.2.9 “Software Fault Prediction using Wrapper based Ant Colony Optimization Algorithm for Feature Selection” [15] 48
3.2.10 “Unit Test Case Generation with Transformers” [16] 50
3.2.11 “Software Fault Prediction Tool” [2] 51
3.2.12 “Transfer Learning Code Vectorizer based Machine Learning Models for Software Defect Prediction” [7] 51
3.2.13 “Vulnerable Code Detection Using Software Metrics and Machine Learning” [3] 52
3.2.14 “Software Defect Prediction Using Software Metrics with Naïve Bayes and Rule Mining Association Methods” [8] 56
3.2.15 “A Novel Neural Source Code Representation based on Abstract Syntax Tree” [5] 56
3.2.16 “Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges” [4] 57
Seq2seq Models: 58
Graph Neural Networks (GNNs): 58
Hybrid Approaches: 58
3.2.17 “Output Sampling for Output Diversity in Automatic Unit Test Generation” [18] 59
3.2.18 “On the “Naturalness” of Buggy Code” [19] 59
3.2.19 “Source Code Preprocessing Method Analysis in Unit Test Code Classification” [17] 60
3.2.20 “Software Defect Prediction via Convolutional Neural Network” [20] 61
3.3 Implemented approach 64
4.5.1. Functional Description 75
4.5.2. Modular Decomposition 76
4.5.3. Design Constraints 76
4.6.1. Functional Description 77
4.6.2. Modular Decomposition 78
4.6.3. Design Constraints 79
Module 1: Maintainability Score Module 82
Module 2: Preprocessor Module 82
Module 3: Bug Localization Module 83
A.1.8. Requests 91
A.1.9. SQLite 91

A.1.10. Pickle 91
A.1.11. Flask 91
A.1.12. GitHub 91
B.1.1 Software Developers 92
B.1.2 Quality Assurance Analysts 92
B.2.1 Project Managers 93
C.1. HomePage and Project uploading 94
C.2. Code Editor 95
C.3. TraceLinks 95
C.4. Issues Files Display 97
C.5. Bug Localization 98
C.6. Maintainability Score (Referred to in appendix E.2) 99
D.1. Technical Feasibility 101
D.2. Economic Feasibility 102
D.3. Legal Feasibility 103
D.4. Operational Feasibility 103
D.5. Schedule Feasibility 103
E.1. Features Extraction Module 104
E.2. Maintainability Score Module 105

List of Figures

Figure 3.1 : CallMeMaybe's workflow ………………………………………………………….24

Figure 3.2 : Fault Prediction workflow diagram
………………………………………………………….27

Figure 3.3 : Software metrics description
………………………………………………………….28

Figure 3.4 :Constructing TestCases using NLP workflow diagram
………………………………………………………….29

Figure 3.5 :using AST in classification ………………………………………………………….30

Figure 3.6 :control flow graph in convolutional neural networks………………………………………………………….31

Figure 3.7 :embedding layer full flow program
………………………………………………………….31

Figure 3.8 :AST nodes types ………………………………………………………….32

Figure 3.9 :Ant Colony Shortest path algo………………………………………………………….34

Figure 3.10 : FSACO flowchart ………………………………………………………….35

Figure 3.11 : transformers unit test generation
………………………………………………………….36

Figure 3.12 : Flow of TLCV ………………………………………………………….38

Figure 3.13 : software metrics methodology………………………………………………………….39

Figure 3.14 :tracking layers in GRU ………………………………………………………….41

Figure 4.1 : stages block diagram ………………………………………………………….53

Figure C.1.1 : Home Page ………………………………………………………….75

Figure C.1.2 Loading Page ………………………………………………………….75

Figure C.2.1 :Code editor ………………………………………………………….76

Figure C.3.1 :TraceLinks Page………………………………………………………….77

Figure C.3.2 :TraceLinks Page-Invalid ……………………………………………………………78

Figure C.4.1 :Files Display Page - Quality Risk Example……………………………………………………………78

Figure C.4.2 :Files Display Page - Bug Example ……………………………………………………………79

Figure C.5.1 :Bug Localization Page……………………………………………………………80

Figure C.6.1 :Maintainability score page……………………………………………………………81

Figure C.6.2 :Maintainability score page-cards……………………………………………………………82

List of Tables

Table 2.1: cost analysis………………………………………………… 10
Table 3.1 :Complexity metrics used dataset………………………………………………… 32
Table 3-2 :Software prediction tool approach resulted table
………………………………………………… 37
Table 3-3 :redundant file software metrics from mozilla firefox
………………………………………………… 40
Table 3-4 :Comparative study on the results of different ML
algorithms[………………………………………………… 41
Table 3-5 :Mapping of the preprocessing
techniques………………………………………………… 45

Table 3-6 :F1 scores on different vectors using different preprocessing techniques using
SVM ………………………………………………… 47
Table 3-7 :F1 scores on different vectors using different preprocessing techniques using
MLP ………………………………………………… 47
Table 5.3 :Testing Schedule ………………………………………………… 80
Table 5.4 :Testing Schedule ………………………………………………… 81
Table 5.5 :Testing Schedule ………………………………………………… 81
Table D.2 :Feasibility study benefits ………………………………………………… 82

List of Abbreviation

API	Application Programming Interface
AST	Abstract Syntax Tree
AUC	Area Under the Curve
CI/CD	Continuous Integration/Continuous Deployment
CS	Coherence Score
CSV	Comma-Separated Values
DL	Deep Learning
GUI	Graphical User Interface
ICTF	Inverse Collection Term Frequency
IDE	Integrated Development Environment
IDF	Inverse Document Frequency
JVM	Java Virtual Machine
LDA	Latent Dirichlet Allocation
LSA	Latent Semantic Analysis
LSTM	Long Short Term Memory
ML	Machine Learning
NLP	Natural Language Processing
PMI	Pointwise Mutual Information
RAM	Random Access Memory
ROC	Receiver Operating Characteristic
SCQ	collection-query similarity
SVM	Support Vector Machine

List of Symbols

D	The set of documents in the collection.
Dt	The set of documents containing term t.
q	A term in the query.
Q	The set of query terms.
td	A document in the document collection D.
tf(t, D)	The frequency of term t in all documents.
tf(t, d)	The frequency of term t in d.
tf(t, Q)	The frequency of term t in the query.

Contacts

Team Members

Name	Email	Phone Number
Yasmin Hashem Niazy	Yasmin.hashem201@email.com	+2 01069926303
Sohad Hossam Edlin	sohad95husam@gmail.com	+2 01142280622
Yasmeen Zaki Bassiouny	yasmeen.zaki01@gmail.com	+2 01221832258
Bassant Hisham Mohamed	bassentkhafagi@gmail.com	+2 01155909550

Supervisor

Name	Email	Number
Dr.Nevin Darwish	ndarwish@eng.cu.edu.eg	+20 122 224 7364

This page is left intentionally empty

Chapter 1: Introduction

In the following sections of this chapter the essential question is spelled out along with the problem definition tackled by our project. In addition, expected approaches and outcomes are presented. The chapter then ends with this document organization.

1.1. Motivation and Justification

When it comes to big software projects, being able to trace project documentation such as requirements, feature requests, enhancements, and bugs to the files in the source code can be daunting. Using bug documentation as an example, the appropriate files that need to be updated ought to be identified and modified, as any faults detected by a user in the software could jeopardize customer satisfaction and damage the company's reputation. As a result of the changes that constantly happen to the codebase, the usual way of backtracking through the entire project to locate the files that should be changed is tedious. More importantly, it is time-consuming and costly.

The essential question is then “How can the process of tracing documentation to source code files in big software projects become more efficient and cost-effective?” The specific question handled by this work is “ How can we quickly trace source code files with related documentation expected without using traditional methods by making use of new technologies such as machine learning?”

The usual process involves backtracking from documentation to potential source code files, which is a tough process and very time-consuming. On the other hand,by blending statistics, classifiers, machine learning models, and natural language processing (NLP), we expect to create a faster and cheaper way to trace documentation with files. It evolves the tracking process, making the whole process of tracing documentation in complex software projects systematically simple and straightforward.

1.2. The Essential Question

When it comes to big software projects, tracing documentation to source code can be daunting. Taking bugs as an example any faults detected by a user in a software

could jeopardize customer satisfaction and damage the company's reputation. As a result of the changes that constantly happen to the codebase, the usual way of backtracking through the entire project to trace documentation to source code files is a tedious process. It is time-consuming and can involve consuming a lot of resources.

So, how can this process become more efficient and cost-effective? How can we quickly trace documentation to files in the source code without using traditional methods? How can machine learning facilitate the process?

The usual process involves backtracking from documentation to potential files in the source code, which is a tedious process and very time-consuming. It is expected that by blending statistics,classifiers, machine learning models, and natural language processing (NLP), we would create a faster and cheaper way for tracing. It evolves the tracking process, making the whole process systematically simple and straightforward.

1.3. Project Objectives

The objectives of this project are as follows:

1. Implement Trace Link Prediction: Use artificial intelligence to predict trace links between requirements and code, improving the traceability and maintainability of the software project.
2. Develop a Bug Prediction Model: Utilize machine learning techniques to predict potential bugs in large software projects, aiming to make the process of identifying faults more efficient and less resource-intensive compared to traditional methods.
3. Optimize Resource Allocation: By predicting bugs and predicting tracelinks correlation scores between the code files and use case documents, optimize the allocation of resources, ensuring that efforts are focused on the areas most likely to contain faults.
4. Calculate Maintainability Scores: Develop a system to calculate maintainability scores for the software project, providing an objective measure of the code's maintainability and helping to identify areas for improvement.
5. Improve Overall Testing Efficiency: Streamline the entire testing process, making it faster, cheaper, and more straightforward, ultimately leading to higher quality software and improved customer satisfaction.

1.4. Problem Definition and Project Outcomes

The problem addressed by this work is defined as "Given a large software project code in Java it is required to show the links between the source code files and requirements documentation ". Our proposed solution uses artificial intelligence (AI) to trace documentation to source code files as well as predicting potential bugs. As a result, speed up of the software development cycle, and optimization with respect to the allocation of resources will be achieved.

In order to satisfy the main goal of the project the expected outcomes are 4 main deliverables namely

- Gathering Software metrics
- Predicting Trace Links
- Bug Localization
- Calculating Maintainability Scores

1.5. Document Organization

This report is organized into six main chapters, each detailing a different aspect of the project. Below is a brief description of each chapter:

Chapter 1: Introduction provides the motivation and justification for the project, presents the essential question, and outlines the project objectives and expected outcomes.

Chapter 2: Visibility Study covers the target customers and includes a market survey. It also presents a business case and financial analysis.

Chapter 3: Literature Survey provides background information on the relevant topics, compares previous work, and details the implemented approach.

Chapter 4: System Design and Architecture gives an overview of the system design and assumptions, describes the system architecture, and breaks down the functional descriptions and design constraints of the modules.

Chapter 5: System Testing and Verification discusses the testing setup, plan, and strategy, including module and integration testing. It also compares the results to previous work.

Chapter 6: Conclusions and Future Work outlines the challenges faced, the experience gained, and provides conclusions and suggestions for future work.

Additional sections include references and appendices that cover development platforms and tools, use cases, user guides, code documentation, and a feasibility study.

Chapter 2: Market Visibility Study

In this chapter we will talk about the market visibility study and analysis for our project. We discuss who are our targeted customers, show our market survey, and finally, we present the business model.

2.1. Target Audience

Our target audience comprises software developers, quality assurance teams, and project managers involved in software development projects. We aim to cater to the needs of both individual developers and enterprise-level organizations seeking to improve their software testing processes.

2.2. Market Survey

The market has some unit testing generating tools. Although those tools support different languages and frameworks they are generic and lack the capability of generating unit tests based on static code analysis. The following subsections review some of the mostly used tools illustrating pros and cons of each.

2.2.1. VisualStudio [28]

Developers can use Visual Studio to generate code metrics data that measure the complexity and maintainability of their managed code. Code metrics data can be generated for an entire solution or a single project.

Pros:

- calculate the scores for the whole project.
- support various programing language

Cons:

- There is no clear explanation for the specific derived formula.

- The set of programs used to derive the metric and evaluate it was small, and contained small programs only.

2.2.2. Requirements Traceability Links[25]

Requirements traceability links represent relationship between requirements and other project artifacts. Consistent traceability in the project allows tracking these relationships to understand information from the related objects

Pros:

- Apart from traceability links in the context of one project, you can also link requirements from different projects
- You can apply a powerful filter to manage traceability. For instance, find missing links, filter links to the origin, and track changes in the linked object by filtering suspect flags
- create traceability links between document objects within a single or multiple projects

Cons:

- Application is Not user friendly

2.2.3. SymFlower[26]

SymFlower is a unit test generation tool that uses symbolic execution which provides a way to analyze a piece of software to identify the inputs that lead to the execution of every path of the program.

Pros:

- Ensures high test coverage.(High test coverage refers to a high percentage of the codebase being exercised by automated tests, ensuring that most functions, statements, branches, and paths are tested. This improves code reliability and helps in early bug detection)
- Ensures all changes in codebase are reflected in the test cases.
- Automatically re-runs all writing Java unit tests as new code is added.

Cons:

- Support for just Java and Go at the moment.
- Has relatively high memory requirements (minimum of 8 cores and 4 GB RAM.)

2.3. Business Model

This section starts by identifying distinguished features that give Intellitest its competitive advantage and then proceeds to give the business case.

2.3.1 Competitive Advantage:

Intellitest distinguishes itself as an advanced software development tool, offering users a myriad of advantages that set it apart in the competitive landscape; specifically, its integrity, time efficiency in addition to its user friendly environment. We briefly justify each of the three features.

1. **Integrity:**
    - Intellitest has a comprehensive code handling mode. It excels at systematically processing extensive source code and meticulously predicting potential faulty modules. It methodically navigates through each module, identifying specific bugs.

2. **Time Efficiency:**
    - Intellitest has a deep learning (DL) -powered bug detection facility. learning (DL) to swiftly detect bugs. This innovative approach of utilizing machine learning eliminates the need for laborious backtracking of source code modules, enhancing overall efficiency in software detection and debugging. In fact, this is its most distinguished key strength feature.

3. **User-Friendly:**
    - Intellitest prioritizes the user experience by featuring an intuitive and user-friendly interface. Users expect a seamlessly navigable platform that delivers clean, easily formatted bug reports. The reports are thoughtfully articulated, ensuring clarity and facilitating a streamlined understanding of identified issues.

Having identified our competitive advantages we proceed to present the business case.

2.3.2 Business Case:

Intellitest presents a compelling business case by introducing a cost-efficient paradigm that surpasses existing testing methods, be they automated tools or the traditional human-centric ones. In the context of current market demands and the rapid evolution of code, the imperative for automation in testing is undeniable.

Key Business Benefits:

1. **Company's Reputation:**

    ○ Intellitest has a low risk mitigation. safeguards the company's reputation by minimizing the likelihood of developers releasing products with hidden bugs. This proactive approach prevents potential issues that could adversely impact user experience, trust, and the overall standing of the company or developer in the market.

2. **Cost of the Process:**

    ○ Intellitest allows for financial optimization. It streamlines the testing process, consolidating code parsing, statistical analysis, faulty module prediction, and bug report generation into a singular tool. This integration significantly reduces the financial costs associated with testing by eliminating the need for multiple tools and minimizing the necessity for human intervention in the process.

3. **Shift Towards Automation:**

    ○ Intellitest is definitely future-ready. As technology advances, the trajectory towards automated code formulation becomes evident. Intellitest positions itself as a solution aligned with this trend, ensuring that testing practices evolve in tandem with the changing landscape of software development. By embracing automation, Intellitest future-proofs testing methodologies, are thus ahead in an environment where code generation itself may become automated.

2.3.3 Subscriptions Planning:

1. Target Market Size:

- Leveraging the extensive Java developer community.
- Considering the widespread use of Java in software development.

2. Market Share and Growth:

- Aiming for a conservative market share of 5% in the Java developer segment.
- Accounting for the demand for advanced testing tools in the Java ecosystem.

3. User Base Growth:

- Initiating with an initial user base of 8,000 users in the first year.(Initiating with an initial user base of 8,000 users in the first year, the projected growth rate is estimated at 15% annually through effective marketing strategies and positive user adoption. This projection is based on industry standards and assumptions regarding market trends.)
- Projecting a growth rate of 15% annually through effective marketing and positive user adoption.

4. Revenue Projection:

- Pricing Structure:
    - Standard Plan: $15/month
    - Professional Plan: $150/year
- Projected Subscriptions Over the Next Three Years:
    - Year 1: 8,000 users
    - Year 2: 9,200 users
    - Year 3: 10,580 users

2.3.4 Financial Analysis:

Revenue Projection:

- Year 1:
    - Standard Plan (SP): 7,000 users * $15/month = $105,000/month
    - Professional Plan (PP): 1,000 users * $150/year = $150,000/year
    - Total Revenue = SP + PP = $105,000/month + $150,000/year = $1,410,000/year
- Year 2:
    - Assuming 15% user growth
    - Standard Plan: 8,050 users * $15/month = $120,750/month
    - Professional Plan: 1150 users * $150/year = $172,500/year
    - Total Revenue = SP + PP = $120,750/month + $172,500/year = $1,621,500/year
- Year 3:
    - Assuming another 15% user growth
    - Standard Plan: 9258 users * $15/month = $138,862/month
    - Professional Plan: 1322 users * $150/year = $198,300/year
    - Total Revenue = SP + PP = $138,862/month + $198,300/year = $1,864,644/year

Cost Analysis:

- Development and Operational Costs:
    - Allocate funds for ongoing software development, infrastructure maintenance, and operational expenses.
- Marketing Expenses:
    - Allocate a portion of revenue for marketing to sustain and increase user growth.
- Profitability:
    - Subtract the total costs from the total revenue for each year to determine the net profit.

Summary:

- Year 1:
  - Total Revenue: Calculated above
  - Total Costs: Estimated development, operational, and marketing expenses
  - Net Profit: Total Revenue - Total Costs
- Year 2:
  - Total Revenue: Calculated above
  - Total Costs: Estimated development, operational, and marketing expenses
  - Net Profit: Total Revenue - Total Costs
- Year 3:
  - Total Revenue: Calculated above
  - Total Costs: Estimated development, operational, and marketing expenses
  - Net Profit: Total Revenue - Total Costs

Year	Revenue from Standard Plan (Monthly)	Revenue from Professional Plan (Yearly)	Total Revenue
Year 1	$105,000	$150,000	$1,320,000
Year 2	$120,750	$172,500	$1,621,500
Year 3	$138,862	$198,300	$1,864,644

Table 2-1 cost analysis

Chapter 3: Literature Survey

We will provide a brief overview of the subjects in this chapter so that you can grasp the project as a whole. First, we will go over the technical topics we understood from various researches. Second, the summarized papers and research from a collection of recent conferences

3.1 Technical Background Knowledge

A brief overview of the technical subjects of interest in the project will be provided in this section.

3.1.1 Natural Language Processing (NLP) [1]

Natural language processing (NLP) is a field of artificial intelligence (AI) concerned with the interaction between computers and human language. NLP deals with the processing, understanding, and generation of human language, including tasks such as machine translation, speech recognition, and natural language generation. NLP techniques can be used to extract temporal constraints from natural language text, such as Javadoc comments or requirements documents. These temporal constraints can then be used to generate test cases that cover all possible sequences of events that could violate a temporal property.

3.1.2 Temporal Properties [1]

Temporal properties are properties of software systems that describe the order in which events must occur. For example, a temporal property might state that a method must be called before another method. Temporal properties are often difficult to test because they can be violated by a variety of different sequences of events. For example, the temporal property "method1 must be called before method2" can be violated if method 2 is called before method1, or if both methods are called concurrently.

3.1.3 Dynamic selection [9]

Dynamic selection is used in the evaluation of the test dataset. It determines the region of competence based on the examples that are mostly similar, dynamically choosing classifiers that are best suited for each specific query instance. It is then used for the final prediction.

3.1.4 Naïve Bayes [8]

Naïve Bayes is a machine learning classifier that is used in solving classification problems. One important aspect of this classifier is that it assumes that all features are independent. This classifier is based on Bayes' theorem and uses the following equation for prediction.

$$p(c|x) = \frac{p(x|c)p(c)}{p(x)}$$

Where p(c|x) is the probability that class c will be predicted given the feature x.

3.1.5 Logistic Regression [9]

Logistic regression is a supervised machine learning model that is used for solving classification problems as well. The output of logistic regression is a category or a discrete value. The output of logistic regression can be divided into 3 types. Binomial: there are only 2 possible outputs corresponding to 0 or 1. Multinomial: more than 2 possible outputs (unordered) corresponding to the probability of each output. Ordinal: more than 3 possible outputs (ordered)

The following formula is used for prediction:

$$p(c|x) = \frac{1}{1+e^{-(w.x+b)}}$$

Where p(c|x) is the probability that class c will be predicted given the feature x, W is the weight and b is the bias term.

3.1.6 K-Nearest Neighbor [9]

“K-nearest neighbors (K-NN) is a non-parametric supervised machine learning classifier primarily employed for classification tasks.”[9] When confronted with new data, the algorithm seeks the most analogous class, classifying the data point accordingly. This process involves calculating the Euclidean distance between the new data point and its neighboring points. The K-nearest neighbors are then chosen based on the smallest distances. Subsequently, the algorithm tallies the number of data points in each category among the selected neighbors. The category with the highest count is assigned to the new data point, effectively determining its classification. This method of classifying based on proximity makes K-NN a straightforward yet potent approach for pattern recognition in various applications.

3.1.7 Area under ROC(Receiver Operating Characteristic) curve [22]

Area under the ROC curve is used in measuring the performance of binary classifiers. The rate of true positives is plotted against the false positives. The area under the curve represents the probability that a chosen positive instance would be randomly ranked higher than a randomly chosen negative one by the classifier.

3.1.8 Software Fault Prediction [9]

Software fault prediction (SFP) is the process of identifying software components that are more likely to contain faults. SFP models are typically trained on historical data of software releases, such as the number of faults found in each component and the code metrics of each component. Once trained, an SFP model can be used to predict the probability of faults in new software releases.

The section then explores the two primary types of SFP techniques: static analysis and dynamic analysis.

- Static Analysis: Static analysis techniques examine the source code of a software system to identify potential fault-prone modules. Code metrics, such as the number of lines of code (LOC), function count, and cyclomatic complexity, are commonly used in static analysis to assess code complexity and fault-proneness.

Some of the most common static analysis techniques for SFP include:

- Code metrics: Code metrics are quantitative measures of the characteristics of a software system, such as the number of lines of code, the number of functions, and the cyclomatic complexity. Code metrics have been shown to be correlated with the fault-proneness of software components.
- Machine learning algorithms: Machine learning algorithms can be used to train models to predict the fault-proneness of software components based on code metrics and other features. Some of the most common machine learning algorithms used for SFP include logistic regression, decision trees, and support vector machines.

- Dynamic Analysis: Dynamic analysis techniques execute the software system and monitor its behavior to detect potential faults. Mutation testing, which involves injecting artificial faults into the code and observing the system's response, and coverage analysis, which measures the extent to which test cases cover the code, are two prominent dynamic analysis techniques.

3.1.9 Source Code Modeling [4]

Source code modeling is the process of converting source code into a form that can be processed by a computer. This is an important step in many software engineering tasks, such as code completion, program synthesis, and code debugging.

There are many different ways to model source code. Some of the most common approaches include the following:

- Tokenization: Tokenization is the process of breaking code into a sequence of tokens, where each token is a single word, punctuation mark, or identifier.
- Parsing: Parsing is the process of analyzing the grammatical structure of code.
- Abstract syntax tree (AST): An AST is a tree-based representation of the grammatical structure of code.
- Intermediate representation (IR): An IR is a low-level representation of code that is closer to machine code than the source code.

Source code models can be used for a variety of tasks, including the following:

- Code completion: Code completion is the task of suggesting code snippets to complete a partially typed code snippet.
- Program synthesis: Program synthesis is the task of automatically generating code from a natural language description.
- Code debugging: Code debugging is the task of identifying and fixing bugs in code.
- Code documentation generation: Code documentation generation is the task of automatically generating documentation for code.
- Code migration: Code migration is the task of converting code from one programming language to another.

3.1.10 Support vector machine(SVM) [7]

SVM is a type of supervised learning model used to operate on small and complex datasets, in classification tasks. Its aim is to find a decision boundary that separates the data points of different classes, maximizing the margin between every pair of classes, to increase the classification accuracy. There are two basic types of SVM, namely. Linear and non-linear. Linear SVM are used when the data can be classified into two classes using a single straight line (2D) while non-linear SVM are used when the data is not linearly separable, which means when the data points cannot be separated into 2 classes by using a straight line (2D). SVM uses kernel methods to transform data features by employing orthogonal kernel functions, where kernel functions map complex datasets to higher dimensions in a manner that enables data point separation.

3.1.11 Vectorization methods [17]

We identify some of the most commonly used vectorization methods below:

- **TF-IDF:** stands for Term Frequency Inverse Document Frequency
    - ❖ TF of a term or word is the number of times the term appears in a document compared to the total number of words in the document.
    - ❖ IDF of a term reflects the proportion of documents in the corpus that contains the term.
- **Word2Vec:** Word2Vec creates vectors of the words that are distributed numerical representations of word features. These word features could comprise of words that represent the context of the individual words present in our vocabulary. Word embedding eventually help in establishing the association of a word with another similar meaning word through the created vectors.
- **Count vector:** Count vector is used to transform a given sentence into a vector on the basis of the frequency (count) of each word that occurs in the entire text. This is helpful when we have multiple sentences, and we wish to convert each word in each sentence into vectors.

3.1.12 Multi-layer Perceptron [21]

A Multilayer Perceptron has input and output layers and one or more hidden layers with many neurons stacked together. However, while in the single Perceptron the neuron must have an activation function that imposes a threshold, like ReLU or sigmoid, neurons in a Multilayer Perceptron can use any arbitrary activation function. It is a feed forward artificial neural network.

3.1.13 Inverse Collection Term Frequency(ICTF) [24]

The ICTF is the opposite of TF which denotes how many times a term appears throughout the whole collection of documents. It is calculated using the following formula: $ictf(t) = log\left(\frac{|D|}{tf(t,D)}\right)$

3.1.14 vector space model [23]:

A term-by-document matrix is used in the VSM to represent software artifacts and documents. The term frequency-inverse document frequency (TF-IDF) of each member of the matrix expresses the term's relevance within the document and corpus. Remember that every document is a vector, hence we can use cosine similarity to calculate how similar two documents are to one another as follows:

$$sim_{cosine}(d_1, d_2) = \frac{\sum_{i=1}^{t} d_{1,i} d_{2,i}}{\sum_{i=1}^{t} d_{1,i}^2 \cdot \sum_{i=1}^{t} d_{2,i}^2}$$

3.1.15 Latent Semantic Analysis(LSA)(referred to in code in appendix E.1) [23]:

As the name implies LSA is a sort of semantic analysis. It captures the co-occurrence of terms which solves the issues with polysemy and synonymy that VSM is unable to do. Simply said, LSA breaks the term-by-document matrix into three matrices using singular value decomposition. The first is a matrix of singular values (S0). A new term-by-document matrix with minimal dimensionality and information about the associations between terms can then be generated by selecting the k biggest values from S0. The rebuilt matrix's document representation and cosine similarity are finally utilized to calculate the degree of similarity between documents.

3.1.16 Latent Dirichlet Allocation(LDA) (referred to in code in appendix E.1) [23]:

LDA is a generative probabilistic model that depicts each subject in the corpus as a distribution over words and each document as a mixture of latent themes. Following the use of LDA, every document is represented as a vector of probabilities, each of which indicates the likelihood that a given topic will be included in the text. We compute the similarity using this representation and the Hellinger distance.to compute the similarity between documents. Specifically, we define the similarity between two LDA document representations d1 and d2 as

$$sim_{LDA}(d_1, d_2) = 1 - \frac{1}{\sqrt{2}} \|(\sqrt{d_1} - \sqrt{d_2})\|$$

3.1.17 Jensen-Shannon [23]:

The JS model also represents software artifacts as probability distributions over terms in the corpus via hypothesis testing techniques [6]. Given the JS representation of two documents d1 and d2, their similarity is computed as follows:

$$sim_{JS}(d_1, d_2) = 1 - \left[H\left(\frac{d_1 + d_2}{2}\right) - \frac{H(d_1) + H(d_2)}{2} \right]$$

$$H(d) = \sum_{w \in W} -P(w) \cdot \log P(w) \qquad (3)$$

3.1.18 Entropy [24]

Entropy is calculated using the following formula

$$entropy(t) = \sum_{d \in D_t} \frac{tf(t, d)}{tf(t, D)} \cdot \log_{|D|} \frac{tf(t, d)}{tf(t, D)},$$

where D_t is the set documents containing the term t and $\frac{tf(t,d)}{tf(t,D)}$ represents the probability that the term t is in the document d. which is the ratio between the number of occurrences of the term t in the document d over the total number of occurrences of the term t in all the documents D.

3.1.19 Okapi BM25 [23]:

The Okapi BM25 model scores each document in the corpus based on the query terms appearing in it. The scoring function is the following

$$score_{BM25}(q,d) = \left(\sum_{t \in q} \log\left[\frac{N}{df_t}\right]\right) \cdot \frac{(\lambda+1)\cdot tf_{t,d}}{tf_{t,d}\cdot\lambda\left((1-b)+b\cdot\left(\frac{L_d}{L_{ave}}\right)\right)}$$

where q is the query, d is a document in the corpus, N is the number of documents in the corpus, df_t is the number of documents the term t appears in, $tf_{t,d}$ is the term frequency of term t in document d, L_d is the length of document d (in number of words), L_{ave} is the average document length in the corpus, b is a parameter used to control how much effect field-length normalization should have, and λ is a positive parameter that calibrates the document term frequency scaling. We used Lucene's2 default implementation of BM25, which utilizes λ = 1.2 and b = 0.75.

3.1.20 Language Model with Dirichlet (LM-Dirichlet), and Jelinek-Mercer smoothing (LM-JM) [23]:

These two models first define a language model for each document in the corpus, and then rank the documents according to the probability that the language model of each document d generates a query q. Both models use smoothing to improve accuracy by adjusting the maximum estimator of a language model.They use the following equation:

$$p(q,d) = \prod_{w \in q} p(q_i|d)$$

3.1.21 Correlation-based Feature Subset Selection (CFS-Subset)[23]:

CFS-Subset is a feature selection technique that selects subsets of features that are highly correlated with the target variable but have low correlation with each other. The idea behind CFS is to identify a subset of features that can provide the maximum amount of information about the target variable while minimizing redundancy among the features.

3.1.22 Pearson's correlation [23]:

Pearson's Correlation is a test statistic that measures the statistical relationship, or association, between two continuous variables. It is known as the best method of measuring the association between variables of interest because it is based on the method of covariance. It gives information about the magnitude of the association, or correlation, as well as the direction of the relationship.

3.1.23 Gain Ratio [23]:

Gain Ratio is an alternative to Information Gain that is used to select the attribute for splitting in a decision tree. It is used to overcome the problem of bias towards the attribute with many outcomes. Gain Ratio is a measure that takes into account both the information gain and the number of outcomes of a feature to determine the best feature to split on. We use Gain Ratio to normalize the Information Gain by the Split information as follows:

$$-\sum_{i=1}^{n} Di \log_2 Di$$

3.1.24 information gain [23]:

Information gain is the reduction in entropy produced from partitioning a set with attributes and finding the optimal candidate that produces the highest value:

$$\mathrm{IG}(T,a) = \mathrm{H}(T) - \mathrm{H}(T|a),$$

where is a random variable and **H(T|A)** is the entropy of **T** given the value of attribute **a**.

The information gain is equal to the total entropy for an attribute if for each of the attribute values a unique classification can be made for the result attribute. In this case, the relative entropies subtracted from the total entropy are 0.

3.1.25 Symmetrical Uncertainty [23]:

Is the product of a normalization of the information gain (**IG**) with respect to entropy. SU(X,Y) is a value in the range [0,1], where *SU(X,Y) = 0* if X and Y are totally independent and *SU(X,Y) = 1* if X and Y are totally dependent.

3.1.26 Synthetic Minority Oversampling Technique [23]:

SMOTE is a technique that generates new data points near the original data points, in order to increase the number of datapoints of the minority class by

- Taking the difference between a sample and its nearest neighbor
- Multiplying the difference by a random number between 0 and 1
- Adding then the difference to the sample to generate a new synthetic example in feature space
- Continuing on with next nearest neighbor up to a user-defined number

3.1.27 Random Majority Undersampling [23]:

RMU randomly eliminates a subset of data points from the majority class to create a more balanced dataset.

3.1.28 Pointwise mutual information (PMI) [24]:

PMI is calculated using the following formula:

$PMI(t_1, t_2) = log \frac{p_{t_1, t_2}(D)}{p_{t_1}(D) \cdot p_{t_2}(D)}$, where $p_{t_1, t_2}(D) = |D_{t_1} \bigcap D_{t_2}|/|D|$, and $p_t(D) = |D_t|/|D|$.

where D_t is the set of documents containing the term t and D is the set of documents in the collection. It is used to calculate the probability that two terms appear together in the corpus.

3.1.28 Similarity Collection Query (SCQ) [24]:

SCQ is calculated using the following formula

$SCQ(t) = (1 + log(tf(t, D))) \cdot idf(t)$

where D is the set of documents in the collection and t is the query term.

3.2 Comparative Study of Previous Work

The following subsections review some of the most recent and most relevant published research of interest to this work.

3.2.1 “Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints” [1]

The Call Me Maybe technique uses the Enhanced English Universal Dependencies parsing library to extract temporal constraints from Javadoc summaries. The semantic dependency graph produced by the parser is then traversed to identify temporal constraints. Temporal constraints are identified by looking for specific dependencies inside the graph. These dependencies identify adverbs and adverbial clauses, which fall into several categories, one of which is time. By referring to time dependencies, the Call Me Maybe Constraint Finder identifies clauses that express the occurrence of an event with respect to another event.

Once the Call Me Maybe Constraint Finder has identified the temporal constraints in the Javadoc summary, it translates them into Java expressions. The Java expressions are then used to generate test cases that cover all possible sequences of events that could violate the temporal constraints.

The Call Me Maybe Constraint Finder is able to handle a wide range of temporal constraints, including:

- Before
- After
- During
- Overlaps
- Finishes
- Starts

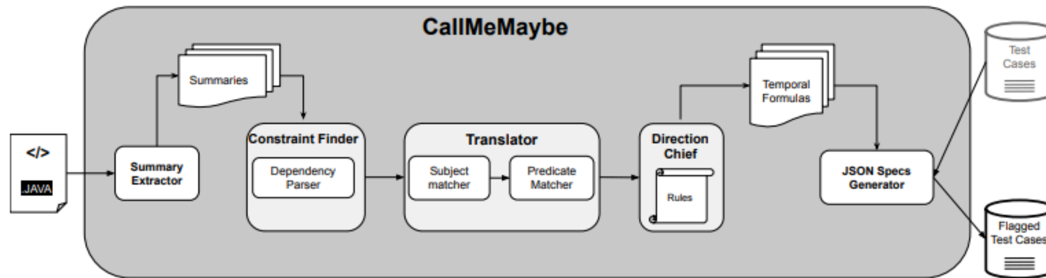


Figure 3-1 : CallMeMaybe's workflow [1]

The Call Me Maybe Constraint Finder is also able to handle temporal constraints that involve multiple events and temporal constraints that involve the state of the system.

The Call Me Maybe Constraint Finder is a powerful tool that can be used to automatically generate test cases for temporal properties. It is able to handle a wide range of temporal constraints, and it is also computationally efficient.

3.2.2 "An Overview of Quality Metrics Used in Estimating Software Faults" [6]

In this study, three software metrics were evaluated based on their values and calculations, namely CK, MODD, and QMOOD.

1. Ck(Chidamber and Kemerer metrics suite) contains the following:
   - WMC (Weighted Methods per Class): WMC is calculated by the complexity of each function in the class; if two classes have the same complexity, we can count the number of methods in each of them.
   - DIT (Depth of Inheritance):DIT is calculated by counting the number of classes between the given class and its root; if it is the root, then the value is 0.
   - NOC (Number of Children):NOC is calculated by counting the number of direct children of a given class; if it does not have children, then the value is 0.
   - CBO (Coupling between Objects):CBO is calculated by counting the number of methods or variables used in a given class plus the number of classes that used the methods or variables of the given class.
   - RFC (Response for a Class): RFC is calculated by counting the number of local methods in a given class plus the methods that are directly called in these local class methods.
   - LCOM (Lack of Cohesion of Methods): LCOM is calculated by subtracting the methods that have zero similarity from those with

non-zero similarity (similarity is the count of the number of attributes that are used in both methods).

2. MODD (Abreu and Carapuca metrics suite)
    - MHF (Method Hiding Factor): MODD is calculated by the division of the number of invisible methods (private methods) and the number of methods in the system.
    - AHF (Attribute Hiding Factor): calculated by the division of the number of invisible attributes (private attributes) and the number of methods in the system.
    - MIF (Method Inheritance Factor): MIF is calculated by dividing the number of inherited methods by the number of all the methods defined in a given class.
    - AIF (Attribute Inheritance Factor): AIF is calculated by dividing the number of inherited attributes by the number of all the attributes defined in a given class.
    - CF (Coupling Factor): CF is calculated by dividing the number of non-inheritance couplings by the maximum number of couplings.
    - PF (Polymorphism Factor): calculated by the division of the number of overriding methods by the maximum number of possible overriding methods.
3. QMOOD (Bansiya and Davis metrics suite): QMOOD enhances both the structural and behavioral properties of the classes
    - DSC (Design Size in Classes): DSC is calculated by counting the number of classes in a given design.
    - NOH (Number of Hierarchies): NOH is calculated by counting the number of hierarchies in a given design.
    - DAM (Data Access Metric): DAM is calculated by a ratio between the number of private and protected attributes and all the attributes in a class.
    - ANA (Average Number of Ancestors): ANA is calculated by a ratio between the numbers of private and protected attributes in all the classes.
    - CAM (Cohesion among Methods of Class): CAM is calculated by counting the number of intersections of parameters in a class with the maximum independent parameters.
    - DCC (Direct Class Coupling) is calculated by counting the number of classes directly associated with other classes.
    - MOA (Measure of Aggregation): MOA is calculated by counting the total number of data declarations that are defined by the user.

- MFA (Measure of Functional Abstraction): MFA is calculated by a ratio between the numbers of inherited methods and all the methods defined in a class.
- NOP (Number of Polymorphic Methods): NOP is calculated by counting the number of all polymorphic methods in a system.
- NOM (Number of Methods): NOM is calculated by counting the number of defined methods in a given class.
- CIS (Class Interface Size): CIS is calculated by counting the number of public methods in a given class.

3.2.3 “Software fault prediction (SFP) based on the dynamic selection of learning technique: findings from the eclipse project study” [9]

This paper first talks about the threat of faults in the software system. Faults in the software system decrease both the reliability and quality of code, resulting in code failures more frequently, which increases costs. Therefore, the paper proposes a software fault prediction model that helps developers detect faults faster and easier. The paper also critiques previous attempts that were made to address the threat of faults problem as they used a single machine learning model, which gave relatively poor performance and higher misclassification errors. The work notices that SFP models are greatly dependent on the software project. The paper then addresses the problem by using two approaches: ensemble methods and dynamic classifier selection.

Ensemble methods mean combining multiple machine learning models to improve accuracy and robustness. The difference between ensemble methods and dynamic classifiers is that in case of ensembles learners and weights selection is done at training time, whereas in dynamic selection it is done at test time, dependent on the test instance that was given. In this way, the idea of using multiple learners is fully utilized.

The algorithm used in the paper is as follows: Firstly, the dataset was divided into 60% for training, 20% for validation, and 20% for testing. The learning techniques that were used were Naïve Bayes, Logistic Regression, and K-Nearest Neighbor. Secondly, they split the validation dataset into disjoint subsets, where each subset contains modules that are similar. Thirdly, evaluating the performance of the models based on the validation data set using the area under the ROC curve (AUC). Lastly, the testing dataset is used to test the predictions of the machine learning models

with the help of the results from the validation dataset.the following figure is a diagram of the flow that was used.

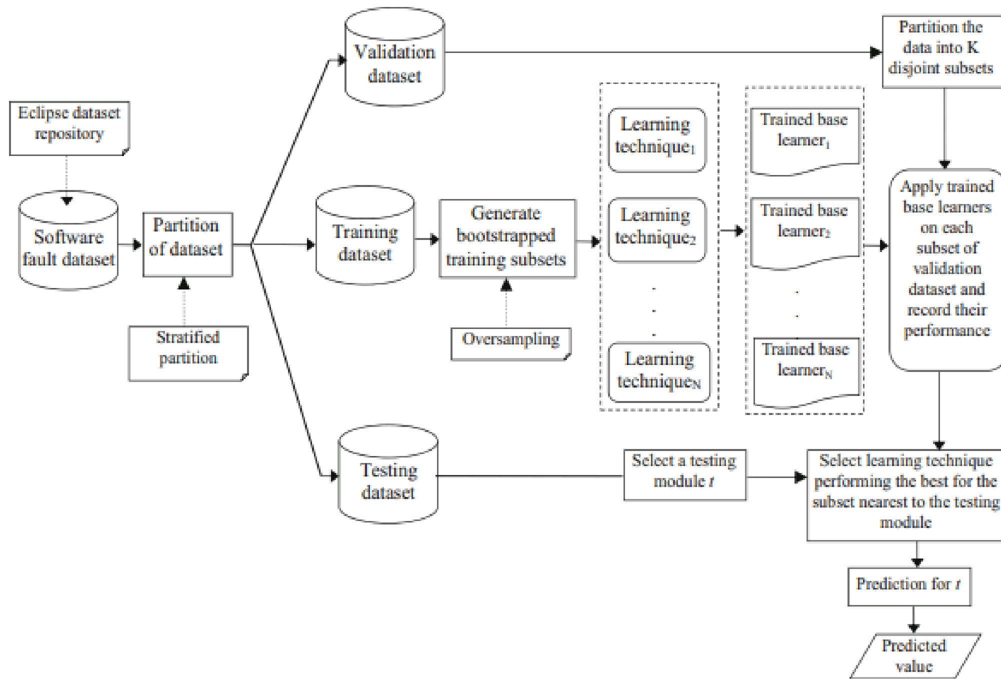


Figure 3-2 : Fault Prediction workflow diagram[9]

The datasets used in the paper are “Eclipse JDT Core, Eclipse PDF UI, Equinox Framework, Lucene, and Mylyn,” as well as nine other object-oriented software datasets.

For performance evaluation, the following were used: AUC analysis, accuracy (– (True positive + True negative) / (Total number of modules)), sensitivity (True Positive / Total number of positive modules), and specificity (True negative / Total number of negative modules).

The following figure is a description of the software metrics used in the paper.

"CBO	Coupling between objects. It count the number of classes coupled to class.
DIT	Depth of inheritance tree. It calculates the depth of a class within the class hierarchy from the root of inheritance.
FANIN	It counts the number of calls to that module from other modules.
FANOUT	It counts the number of calls from that module to other modules.
LCOM	Lack of cohesion among methods. It shows the dependency of two methods on each other by counting the method-pairs that do not share a field to the method-pairs that do.
NOC	Number of children. It counts the number of immediate descendants of a class.
NumberOfAttributes	It shows the number of data members of a class.
NumberOfAttributesInherited	It shows the number of data members a class inherited from its ancestor classes.
NumberOfLinesOfCode	It counts the number of lines of the code a class have.
NumberOfMethods	It counts the number of member functions defined in a class.
NumberOfMethodsInherited	It shows the number of methods a class inherited from its ancestor classes.
NumberOfPrivateAttributes	It shows the number of data members defined in a class in the private access scope.
NumberOfPrivateMethods	It shows the number of member functions defined in a class in the private access scope.
NumberOfPublicAttributes	It shows the number of data members defined in a class in the public access scope.
NumberOfPublicMethods	It shows the number of member functions defined in a class in the public access scope.
RFC	Response of a class. It counts the number of distinct methods invoke by a class in response to a received message.
WMC	Weighted method count. It counts the number of methods defined in a class
CA	Afferent couplings. It measures the number of classes that depend upon the measured class.
CE	Efferent couplings. It measures the number of classes that the measured class is depended upon.
NPM	Number of public methods. It counts all the publicly declared methods in a class.
LCOM3	It is a variation of LCOM metric and measures the cohesion of a class.
LOC	Lines of code. It counts the number of non-commented lines of code a class.
DAM	Data access metric. It shows the ratio of the number of private or protected attributes to the total number of attributes declared in the class.
MOA	Measure of aggregation. It measures the number of class fields whose types are user defined classes.
MFA	Measure of functional abstraction. It counts the total number of methods accessible by the member methods of the class. The constructors are ignored.
IC	Inheritance coupling. It measures the number of parent classes to which a given class is coupled.
CBM	Coupling between methods. It measures total number of new/redefined methods to which all the inherited methods are coupled.
AMC	Average method complexity. It measures the average method size for each class.
CC	Cyclomatic complexity. It is equal to the number of different paths in a method (function) plus one."

Figure 3-3 : Software metrics description [9]

3.2.4 “Constructing Test Cases Using Natural Language Processing” [10]

The paper deals with the topic of automatic test case generation using natural language processing. Firstly, it describes the problem that inspired the paper, which is that the current method of generation of test cases by testers takes quite some time, which could be dramatically decreased if the proposed method is used (using natural language processing for the generation of test cases). The proposed approach in the paper is feeding the functional requirement documentation to the NLP model to extract certain keywords that would help in creating scenarios for tests. The following figure is a block diagram that shows the flow of the module.

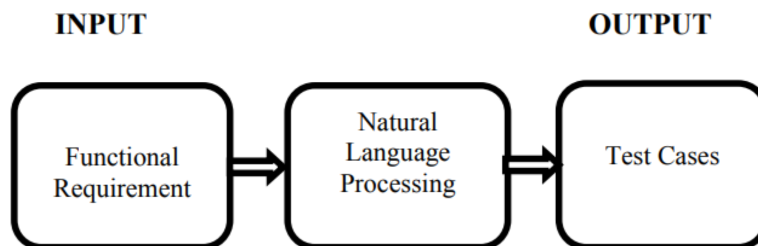


Figure 3-4 :Constructing TestCases using NLP workflow diagram [10]

3.2.5 “Fast Static Analyses of Software Product Lines — An Example With More Than 42,000 Metrics” [11]

The main idea of this paper is the extraction of different software metrics. The problem that this paper handles is that previous approaches did not consider the variability of the software product. The paper proposes a solution, which is the partial parsing solution, as well as using a reduced abstract syntax tree, but the main output from this paper is the ability to use abstract syntax trees in the extraction of the software metrics.

3.2.6 “Unified Abstract Syntax Tree Representation Learning for Cross-Language Program Classification” [12]

The problem that this paper deals with is the classification of programs that are written in different programming languages. The main takeaway from this paper is how they use a single abstract syntax tree to feed it to the machine learning models that they use, even though each programming language produces different abstract syntax trees since each programming language has its own rules and characteristics. They approach this problem using a mechanism called “unified

vocabulary," which is used to normalize node names in the AST. This will cause the AST to become unified. This approach can help in the extraction of the software metrics to be used in software fault prediction.

In the following diagram in the figure is an example of how the unified representation of the AST can be used in classification of software programs.

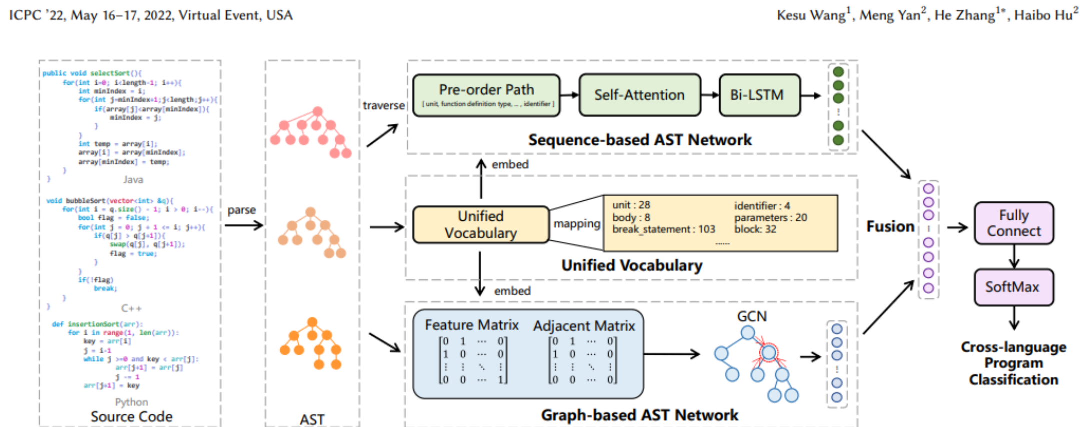


Figure 3-5 :using AST in classification [12]

3.2.7 "Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction" [13]

This paper approaches the problem of software fault prediction in a different way. It approaches the problem using graph-based convolutional neural networks instead of using software metrics.

The approach is as follows:

- Generate a control flow graph
- Apply a graphical model to the control flow graph dataset.

Control flow graphs are directed graphs, G = (V, E). V represents the vertices, and E represents the edges. A vertex in the control flow graph represents a code block with a single entry and exit point, while the edges represent the flow between the code blocks. The following is an example of a control flow graph.

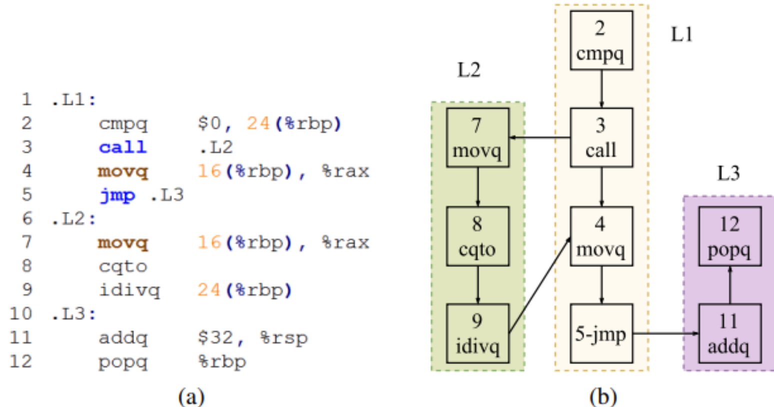


Figure 3-6 :control flow graph in convolutional neural networks [13]

The second step is using a graph-bases convolutional neural network

The first layer in the network is called the embedding layer, where a vertex in the CFG is represented as a vector. To be able to extract local features, fixed-sized circular windows were used to slide over the graph. To be able to collect all the extracted features, a dynamic pooling layer is applied. The feature vector is then given to the network to be able to predict the output. The following figure is a diagram that shows the full flow of the program

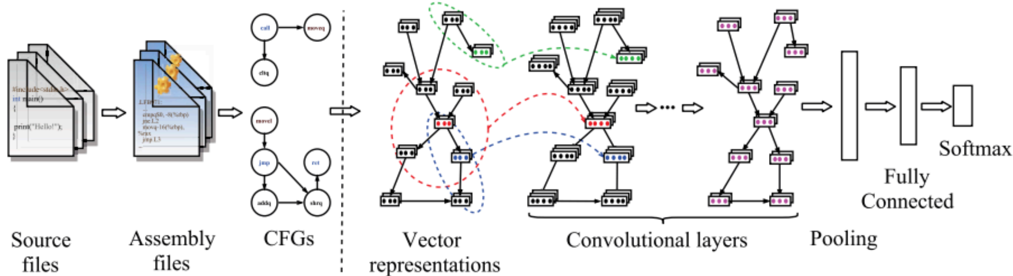


Figure 3-7 :embedding layer full flow program [13]

The dataset that was used was obtained from a programming contest site called CodeChef. For performance evaluation AUC was used.

3.2.8 “Predicting Defects for Eclipse” [14]

In the aim of finding out why software products have defects, finding software products that have defects is the first step in the analysis. This paper explains the dataset created by the writer as well as the research done to reach the choice of its features. The dataset is then used to create a defect prediction model to classify and rank files and packages in the Eclipse project which are the most likely to have bugs.

The dataset uses both the version database (e.g. CVS) which is the contribution history to the source code, and the bug database (e.g. BUGZILLA) which is used by the testers to report and keep track of bugs. These two databases are used to relate bug reports to fixes and thus determine the number of defects of a component.

Complexity metrics are also calculated for each file in the code as well as the number of syntactic elements. The paper used Java Parser for Eclipse and Abstract Syntax Tree for these calculations.

The following table shows the complexity metrics used by the authors in this dataset.

		Metric	**File level**	**Package level**
methods	FOUT	Number of method calls (fan out)	avg, max, total	avg, max, total
	MLOC	Method lines of code	avg, max, total	avg, max, total
	NBD	Nested block depth	avg, max, total	avg, max, total
	PAR	Number of parameters	avg, max, total	avg, max, total
	VG	McCabe cyclomatic complexity	avg, max, total	avg, max, total
classes	NOF	Number of fields	avg, max, total	avg, max, total
	NOM	Number of methods	avg, max, total	avg, max, total
	NSF	Number of static fields	avg, max, total	avg, max, total
	NSM	Number of static methods	avg, max, total	avg, max, total
files	ACD	Number of anonymous type declarations	value	avg, max, total
	NOI	Number of interfaces	value	avg, max, total
	NOT	Number of classes	value	avg, max, total
	TLOC	Total lines of code	value	avg, max, total
packages	NOCU	Number of files (compilation units)	N/A	value

Table 3-1 :Complexity metrics used dataset [14]

The final form of the dataset features thus include Name of File or Package, pre-release defects, post-release defects, complexity metrics and the frequency of each type of node in the AST.

This figure shows the nodes of the AST.

AnnotationTypeDeclaration
AnnotationTypeMemberDeclaration
AnonymousClassDeclaration
ArrayAccess
ArrayCreation
ArrayInitializer
ArrayType
AssertStatement
Assignment
Block
BlockComment
BooleanLiteral
BreakStatement
CastExpression
CatchClause
CharacterLiteral
ClassInstanceCreation
CompilationUnit
ConditionalExpression
ConstructorInvocation
ContinueStatement
DoStatement
EmptyStatement
EnhancedForStatement
EnumConstantDeclaration
EnumDeclaration
ExpressionStatement
FieldAccess
FieldDeclaration
ForStatement
IfStatement
ImportDeclaration
InfixExpression
Initializer
InstanceofExpression
Javadoc
LabeledStatement
LineComment
MarkerAnnotation
MemberRef
MemberValuePair
MethodDeclaration
MethodInvocation
MethodRef
MethodRefParameter
Modifier
NormalAnnotation
NullLiteral
NumberLiteral
PackageDeclaration
ParameterizedType
ParenthesizedExpression
PostfixExpression
PrefixExpression
PrimitiveType
QualifiedName
QualifiedType
ReturnStatement
SimpleName
SimpleType
SingleMemberAnnotation
SingleVariableDeclaration
StringLiteral
SuperConstructorInvocation
SuperFieldAccess
SuperMethodInvocation
SwitchCase
SwitchStatement
SynchronizedStatement
TagElement
TextElement
ThisExpression
ThrowStatement
TryStatement
TypeDeclaration
TypeDeclarationStatement
TypeLiteral
TypeParameter
VariableDeclarationExpression
VariableDeclarationFragment
VariableDeclarationStatement
WhileStatement
WildcardType

Figure 3-8 :AST nodes types [14]

Experimental model created using this dataset consists of two modules: Classification to answer the question which files/packages have defects and Ranking module to answer which has the most defects. Classification module is implemented using a logistic regression model while the ranking module is implemented with a linear regression model.

3.2.9 “Software Fault Prediction using Wrapper based Ant Colony Optimization Algorithm for Feature Selection” [15]

This paper proposes the use of feature selection (FS) algorithms to reduce the features of high dimensional datasets, i.e. datasets that contain a large quantity of features that make the learning process slower as well as reduce the accuracy of the model. FS filters features down to the most important features and removes the repeated or unnecessary ones. The approach here uses Ant Colony Optimization (ACO) that uses ant colony intelligence and pheromone secretion to reach the shortest path. The figure below illustrates the way ants could reach food using the shortest path.

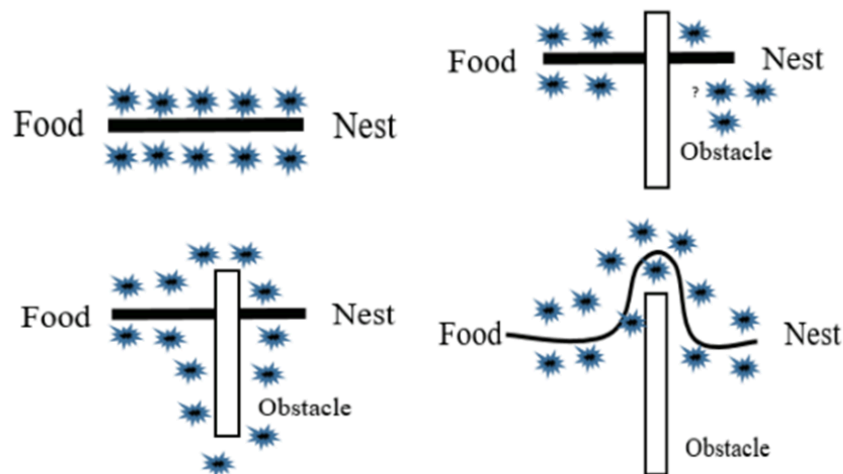


Figure 3-9 :Ant Colony Shortest path algo [15]

With the evaporative nature of pheromones, the pheromones in the longer path would evaporate more than on the shorter one, causing the ants to follow the path with most pheromones, i.e. the shortest path. The algorithm here is applied to software fault prediction datasets that contain both complexity metrics and historical data. Twelve of the SFP datasets were used as well as 3 different classification modules in order to observe the pure effect of feature selection using ant colony optimization (FSACO) without the bias of the classification algorithms.

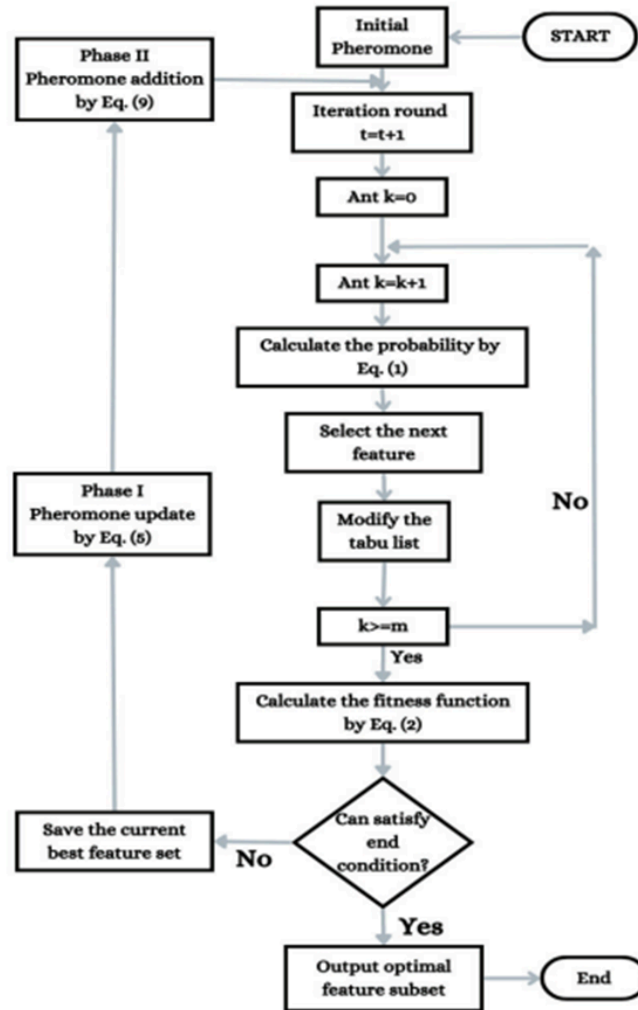


Figure 3-10 : FSACO flowchart [15]

The flowchart in the figure describes the applied algorithm step by step.

For initialization, all pheromone values are set to 0 and the following equation is applied:

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{\tau_{ij}^{\alpha}(t)\eta_{ij}^{\beta}(t)}{\sum\limits_{s \notin tabu_k} \tau_{is}^{\alpha}(t)\eta_{ij}^{\beta}(t)}, & j \notin tabu_k \\ 0, & \end{cases}$$

Where p(t) is the probability that the ant moves from feature i to j during the t iteration, Ƞij is the heuristic information. The heuristic information here is the Euclidean distance between two different nodes of features and the performance of the classifier used.

The path of the ant is considered using a fitness function to evaluate the feature subset of every ant. This fitness function is:

$$F = \frac{\sum W}{Pd}$$

W is the number of the wrong predictions. Pd is the sum of all predictions. The less the value of F the better the classification performance.

The pheromone values are then updated in two stages. The first is where the concentration of the pheromone of all the paths traveled by the ant is updated in iteration t. The second stage is where the concentration is increased randomly in some paths to avoid getting stuck at the local optimum and diversifying the paths.

As a result, the 12 datasets that have an average of 30 features now have an optimized feature list with an average of 8 features. The predicted accuracy also increased for all datasets with all the classification algorithms.

3.2.10 "Unit Test Case Generation with Transformers" [16]

The current implementation for automatic unit test generation is lacking in areas such as readability, code quality, and bug detection capability. Here the writer's approach is to train a sequence-to-sequence transformation model on a dataset of unit tests created by developers to generate proper readable test cases.

The paper introduces "ATHENATEST" with the following modules:

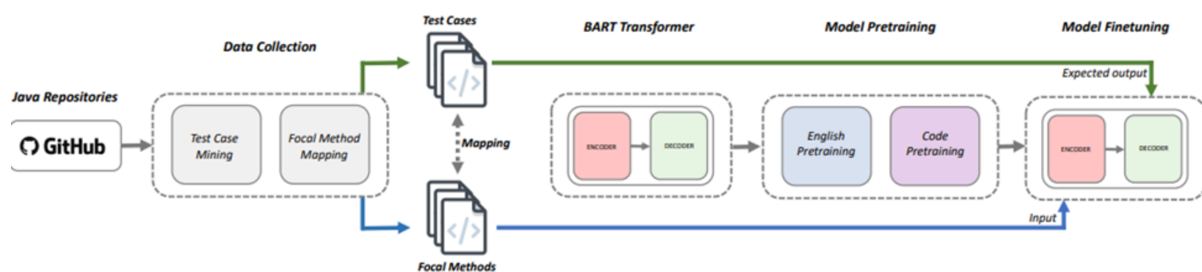


Figure 3-11 : transformers unit test generation [16]

The Data Collection module parsers Java repositories on GitHub to extract pairs of methods under test, i.e. focal methods and corresponding unit tests. The next step is pre training a BART transformer on English and Java code; this is the model pre-training module. Lastly, model is trained with the collected pairs to generate unit test cases, i.e. Model Fine Tuning.

In the process of Data Collections, the developers created the biggest public Java unit test generation dataset to this day, calling it METHODS2TEST, and provided it publically through a GitHub repository. Pretraining included English Pretraining to learn semantic and statistical properties of natural language. The technique was corrupting English documents and training the module to reconstruct it. Similarly, Java Coding pre training was also done.The fine tuning stage used the pairs collected in a translation task with the use of cross entropy loss and Adam optimizer. The source here is the focal method and the target is its unit test case extracted.

3.2.11 “Software Fault Prediction Tool” [2]

The paper provides a comprehensive overview of software fault prediction (SFP), a crucial aspect of software development that aims to identify fault-prone modules or files early in the development cycle. The implemented approach utilizes a hybrid model that combines code metrics and machine learning algorithms. The model employs a variety of code metrics to capture the characteristics of software modules and employs a logistic regression algorithm, a simple yet effective machine learning algorithm, to predict fault-proneness.

The choice of a hybrid model stems from the strengths of both code metrics and machine learning algorithms. Code metrics provide a direct measure of code complexity and fault-proneness, while machine learning algorithms can capture complex patterns and relationships within the code. The combination of these approaches aims to enhance prediction accuracy.

The following is an example of the results table:

File name	Line count	Function count	Cyclomatic complexity	Predicted probability of fault
main.cpp	1000	100	20	0.75
utility.cpp	500	50	10	0.5
test.cpp	200	20	5	0.25

Table 3-2 :Software prediction tool approach resulted table [2]

3.2.12 “Transfer Learning Code Vectorizer based Machine Learning Models for Software Defect Prediction” [7]

In this work , the authors began by discussing different machine learning models used in software defect prediction, such as a model that used support vector machines (SVM), artificial neural networks (ANN), and another model that used Bayesian belief networks (BBN) for SDP. They then stated that the tree-based algorithms give better performance compared to other algorithms. They mentioned a problem that faced the SDP models: datasets have a high level of class imbalance, which means that the occurrences of non-defective data points are much higher than the defective data points; therefore, the synthetic minority oversampling technique was used.

They selected the Apache project as a dataset and the F1 score as a performance level.They proposed a model called Transfer Learning Code Vectorizer, which uses the ULMFit architecture to generate vectors from the source code of each module itself. The figure below shows the flow of the training process.

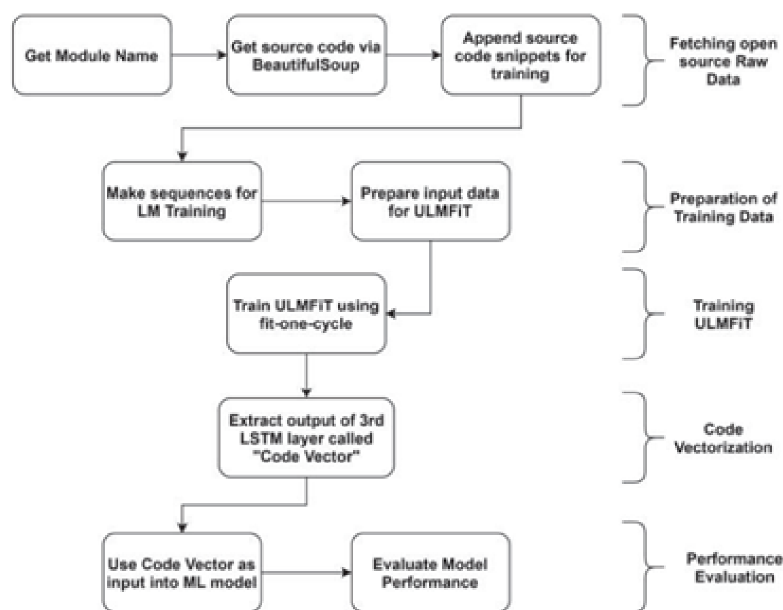


Figure 3-12 : Flow of TLCV [7]

The main results achieved were that Transfer Learning Code Vectorizer works well with neural networks, decision trees, and the Gaussian Naïve Bayes model.

3.2.13 “Vulnerable Code Detection Using Software Metrics and Machine Learning” [3]

methodology used in this approach

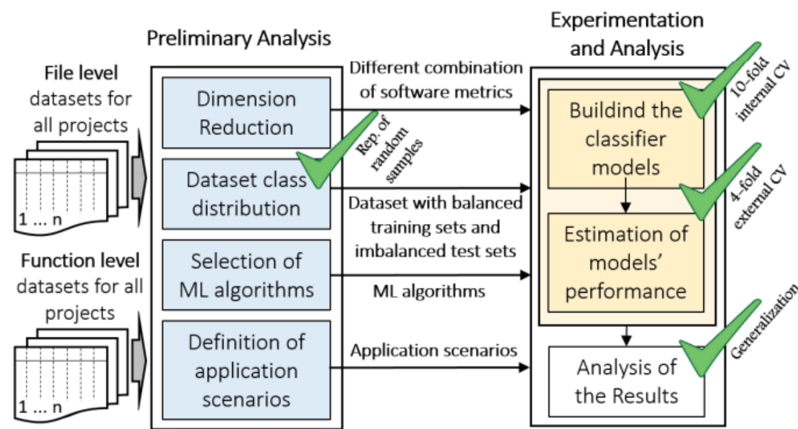


Figure 3-13 : software metrics methodology [3]

This paper presents a study of software metrics and machine learning for vulnerable code detection. The study evaluates the performance of different software metrics and machine learning algorithms for detecting vulnerabilities in a dataset of open-source software projects. The study finds that the XGBoost machine learning algorithm outperforms other machine learning algorithms, such as Support Vector Machines (SVMs) and Random Forest, in terms of F1 score. The study also finds that the Modularity software metric is the most informative software metric for vulnerable code detection. here is a summary of the project plan explained in the paper: Comparative Study of Software Metrics and Machine Learning for Vulnerable Code Detection

Project Plan

Phase 1: Data Collection

- Collect software metrics from a dataset of open-source software projects.
- Select the most informative software metrics for vulnerable code detection.
- Preprocess the data for machine learning.

(a)								(b)	
Irrelevant (I) and Redundant (R) File level metrics								# of samples	
#	Software Metrics	MOZILLA	KERNEL	XEN	APACHE	GLIBC		Irr.	Red.
1	AvgCyclomatic	I / R	I / R	I / R	I / R	I / R		9	10
2	AltAvgLineBlank	I / R	I / R	I / R	I	R		9	7
3	AvgCyclomaticModified	I / R	I	I	I / R	I / R		8	7
4	AvgCyclomaticStrict	I	I / R	I	I / R	I / R		8	1
5	AvgLine	I	I / R	R	I	I / R		9	0
6	FanOut	I / R	I	I / R	I / R	I		10	1
7	FanIn	I	I	I	I	I / R		10	0
8	SumMaxNesting	I	I	I	I	I / R		10	0
9	MaxMaxNesting	I	I	I	I	R		10	0
10	HK	I	I	I	I	I		9	0
11	LCOM	I	I	I	I	I		9	0
12	MaxFanIn	I	I	I	I	I		10	0
13	CountPath	I	I	I	I	I		10	0
14	CBO	I	I	I	I	I		10	0
15	CountLineBlank	R	R	R	R	R		0	9
16	CountLineCodeExe	R	R	R	R	R		0	10
17	CountSemicolon	R	R	R	R	R		0	10
18	CountStmt	R	R	R	R	R		0	10
19	CountStmtExe	R	R	R	R	R		0	10
20	MaxCyclomatic	R	R	R	R	R		1	9
21	MaxCyclomaticModified	R	R	R	R	R		1	6
22	AltAvgLineCode	I / R	I / R		I / R	I / R		10	9
23	AvgLineCode	I / R		R	I / R	I / R		10	8
24	AvgLineBlank	I	I	I	I / R			9	1
25	CountLineCode		R	R	R	R		0	7
26	SumCyclomatic	R	R	R		R		0	9
27	SumCyclomaticModified	R	R	R	R	R		0	8
28	AltCountLineCode	R	R		R	R		0	9
29	AltCountLineComment	R	R	R		R		0	9
30	SumCyclomaticStrict	R		R	R	R		0	8
31	RatioCommentToCode	I	I		I	I		10	0
32	CountStmtEmpty	I	I	I		I		7	0
33	AvgFanIn	I	I	I	I			10	0
34	AltAvgLineComment	I / R	I / R	I				9	10
35	AvgEssential	I			I	I / R		8	0
36	AvgLineComment	I	I	I / R				8	0
37	AltCountLineBlank		R	R	R			0	6
38	CountDeclFunction		R	R		R		1	0
39	CountLine	R		R		R		0	8
40	CountLineCodeDecl	R		R	R			0	10
41	CountStmtDecl		R	R	R			0	0
42	CountLineInactive		I	I	I			2	0
43	MaxFanOut	I		I	I			10	0
44	AvgMaxNesting	I				I / R		7	0
45	MaxCyclomaticStrict			R		R		0	2
46	AvgFanOut					R		8	0
47	CountLineComment					R		0	2
48	SumEssential				R			0	6
49	CountLinePreprocessor				I			0	0
50	MaxEssential							0	0
51	MaxNesting							0	0

Table 3-3 :redundant file software metrics from mozilla firefox [3]

Phase 2: Machine Learning Algorithm Selection

- Train and evaluate different machine learning algorithms for vulnerable code detection.
- Select the machine learning algorithm that achieves the best performance.

Options:

- Random Forest
- Extreme Gradient Boosting (XGBoost)
- Support vector machines (SVM)

Comparative study on the results of different ML algorithms:

Software Metric	Machine Learning Algorithm	Accuracy	Precision	Recall	F1 Score
Lines of Code (LOC)	Random Forest	85%	80%	75%	77%
Cyclomatic Complexity	Extreme Gradient Boosting (XGBoost)	90%	85%	80%	82%
Number of Functions	Support Vector Machines (SVMs)	88%	83%	78%	80%
Average Function Size	Random Forest	87%	82%	77%	79%
Modularity	XGBoost	91%	86%	81%	83%

Table 3-4 :Comparative study on the results of different ML algorithms[3]

Phase 3: Model Training

- Train the selected machine learning algorithm on the preprocessed data.
- Evaluate the trained model on a holdout dataset.

Phase 4: Vulnerable Code Detection

- Use the trained model to detect vulnerable code units in the software projects.
- Evaluate the effectiveness of the vulnerable code detection approach.

3.2.14 “Software Defect Prediction Using Software Metrics with Naïve Bayes and Rule Mining Association Methods” [8]

The paper first compared different software prediction models as Support Vector Machine, joint method Neural Network (NN) and ARM-based Particle Swarm Optimization (PSO). The model uses NASA MDP data set The Design of the SDP model is as follows:

- Normalize the dataset from numeric type to category using the k-means method.
- Select features from data set software metrics using association rules mining
- Make predictions of software module defects using algorithms

3.2.15 “A Novel Neural Source Code Representation based on Abstract Syntax Tree” [5]

The paper introduces Abstract syntax tree based neural networks for code representation. The usual AST is deep and large for large code bases. Thence, bottom computations from leaf to node will have so many problems like gradient vanishing problems and will miss some of the semantics. It made a comparison between Abstract Syntax Tree, Tree-based Neural Networks

- AST: is used for representing abstract syntactic structure of the source code
- Tree-based Neural Network: it takes an AST learn its vector representation by recursively computing node embeddings in a bottom-up way, those are its types:
    - ❖ Recursive Neural Network
    - ❖ Tree-based Convolutional Neural Network
    - ❖ Tree-based Long Short-Term Memory

The paper then explained its approach to build the Abstract syntax tree based neural network which is as follows

- They split the AST into smaller trees depending on the statements constructing a ST-tree.
- They design a recurrent neural network to learn representations of statements which is referred to as the encoding layer.
- They use batches to speed the algorithm by putting childrens of the same position in one batch .

- They use GRU to track the statements .

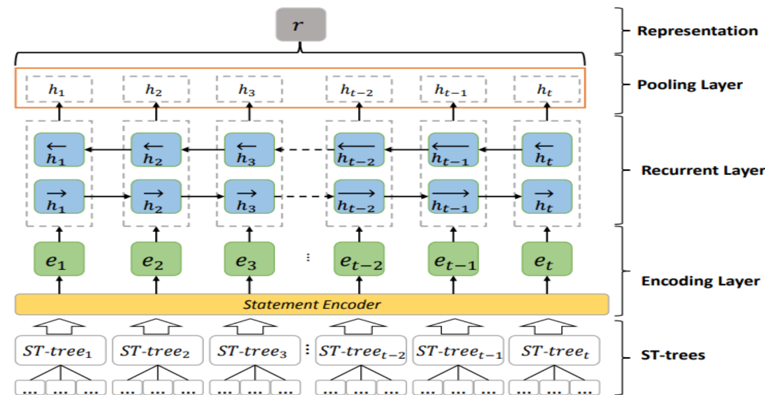


Fig. 2. The architecture of AST-based Neural Network

Figure 3-14 :tracking layers in GRU [5]

3.2.16 “Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges” [4]

This paper deals with the topics of code representation, modeling and generation.

Code representation is the process of converting code into a form that can be processed by a computer. There are many different ways to represent code, but some of the most common include:

- Tokenization: Tokenization is the process of breaking code into a sequence of tokens, where each token is a single word, punctuation mark, or identifier.
- Parsing: Parsing is the process of analyzing the grammatical structure of code.
- Abstract syntax tree (AST): An AST is a tree-based representation of the grammatical structure of code.
- Intermediate representation (IR): An IR is a low-level representation of code that is closer to machine code than the source code.

Recent years have seen a surge in research on deep learning for source code modeling and generation. Many different approaches have been proposed, with varying degrees of success. In this section, it presents a comparative study of some of the most promising models and approaches.

Seq2seq Models:

- Seq2seq models are a type of neural network that are trained to generate sequences of output tokens from sequences of input tokens. They have been shown to be effective for a variety of code modeling and generation tasks, including code completion, program synthesis, and code debugging.
- One of the most well-known seq2seq models for code generation is the DeepCodeGenerator model. DeepCodeGenerator is a seq2seq model that is trained on a large dataset of Java code. It has been shown to achieve state-of-the-art results on a variety of code generation benchmarks.
- Another promising seq2seq model for code generation is the Code2Seq model. Code2Seq is a seq2seq model that is trained on a large dataset of Python code. It has been shown to achieve competitive results with DeepCodeGenerator on a variety of code generation benchmarks.

Graph Neural Networks (GNNs):

- GNNs are a type of neural network that is designed to learn from graph-structured data. They have been shown to be effective for code modeling and generation tasks that involve reasoning about the relationships between different parts of code, such as code documentation generation and code migration.
- One of the most well-known GNNs for code generation is the DeepCodeDoc model. DeepCodeDoc is a GNN that is trained on a large dataset of Java code. It has been shown to achieve state-of-the-art results on a variety of code documentation generation benchmarks.
- Another promising GNN for code generation is the Context2Name model. Context2Name is a GNN that is trained on a large dataset of Python code. It has been shown to achieve competitive results with DeepCodeDoc on a variety of code documentation generation benchmarks.

Hybrid Approaches:

- Some researchers have explored hybrid approaches that combine seq2seq models and GNNs. These approaches aim to leverage the strengths of both types of models to achieve better results than either model alone.

3.2.17 “Output Sampling for Output Diversity in Automatic Unit Test Generation” [18]

The state of the art automatic test generation tools such as EvoSuite and CAVM are trained to hit line and branch coverage but there is proof that tools trained with this purpose have limitations in terms of discovering bugs in code. Here, the authors introduce a different approach called OutGen “the first totally automatic output sampling for output diversity unit test set generation tool”. This tool adds semantic information, i.e. including information about the data being tested as well as diversity to the test set to ensure the highest output uniqueness score which strongly correlates to bug discovery and high coverage.

The concept of output sampling used in this paper is for the purpose of identifying more inputs that map to rare software behavior where 80% of the bugs exist. The criteria here for evaluating this approach against current approaches is the mutation score and bug detection.

3.2.18 “On the “Naturalness” of Buggy Code” [19]

This paper proposes a unique approach to static bug detection which is that normal “Real” software is repetitive and predictable forming a sense of “Naturalness” which buggy code does not have leading to language models to suspect such code and highlight it for the developer to focus on.
The authors test this hypothesis using a dataset of buggy commits and bug fix commits to rate the sense of naturalness of these commits. This hypothesis can be used to create tools that compete with current static bug finders in terms of performance.
The variable that represents naturalness here is entropy. Unnaturalness leads to a higher entropy value. Entropy is measured using a cache language model. Equation for calculating the entropy for a code snippet is:

$$H_M(S) = -\frac{1}{N}\log_2 P_M(S) = -\frac{1}{N}\sum_{1}^{N}\log_2 P(t_i|h)$$

3.2.19 “Source Code Preprocessing Method Analysis in Unit Test Code Classification” [17]

This study provides an analysis to better understand the influences of several text preprocessing methods on the performance metrics of machine learning models in the case of a simple code coverage classification (branch or non-branch). Further, it aims to investigate the effect of using different preprocessing methods and their combinations on the machine learning model's performance.

It compares two research papers that used the Convolutional Neural Network model and text preprocessing in the form of lexicalized tokens and word embeddings to classify 60 programming languages in the GitHub repository. and another model that used the Long Short-Term Memory (LSTM) model and text preprocessing in the form of tokenization and word embeddings to classify programming languages, where the LSTM model gave better accuracy.

They then measure accuracy and F1 score depending on different combinations of preprocessing methods, which are alphanumeric character extraction, replacement of numeric characters into special characters, and removal of newline and tab-line characters, and using different vectorization methods, which are no vectorization, count vector, TF-IDF, and Word2Vec; they apply those methods to two models, which are support vector machine and MLP.

Below is the mapping of the preprocessing techniques. 0 means the technique is not used, and 1 means the technique is used.

Combinations	Extract alphanumeric	Numeric to special character	Remove new line and tab-line
000	0	0	0
001	0	0	1
010	0	1	0
011	0	1	1
100	1	0	0
101	1	0	1
110	1	1	1
111	1	1	1

Table 3-5 :Mapping of the preprocessing techniques [17]

In the below tables are the F1 scores for extracting different vectors using different combinations of the preprocessing techniques.

Using SVM:

Vector	Combination							
	000	*001*	*010*	*011*	*100*	*101*	*110*	*111*
no vector	0.87	0.80	0.86	0.80	**0.93**	0.82	0.92	0.85
Count	**0.94**	**0.94**	0.89	0.91	0.90	0.90	0.88	0.90
TF-IDF	0.95	0.93	0.94	0.93	**0.96**	0.95	0.95	0.93
Word2 Vec	0.81	0.81	0.79	0.79	**0.91**	0.86	0.79	0.79

Table 3-6 :F1 scores on different vectors using different preprocessing techniques using SVM [17]

Using MLP:

Vector	Combination							
	000	*001*	*010*	*011*	*100*	*101*	*110*	*111*
no vector	**0.83**	0.76	0.78	0.77	0.76	0.78	0.77	0.77
Count	0.95	0.97	0.94	0.94	**0.99**	0.97	0.94	0.94
TF-IDF	0.92	**0.96**	0.92	0.92	**0.96**	**0.96**	0.92	0.91
Word2 Vec	0.82	0.80	0.76	0.76	0.79	0.82	**0.83**	**0.83**

Table 3-7 :F1 scores on different vectors using different preprocessing techniques using MLP [17]

3.2.20 “Software Defect Prediction via Convolutional Neural Network” [20]

This paper proposes a framework called Defect Prediction via Convolutional Neural Network (DP-CNN) for software defect prediction. The framework uses deep learning to extract token vectors from Abstract Syntax Trees (ASTs) and encode them as numerical vectors. These numerical vectors are then inputted into a Convolutional Neural Network to learn semantic and structural features of programs. The framework also combines traditional hand-crafted features for accurate software defect prediction. The study evaluates the effectiveness of this framework.

The implementation is divided into several steps. First, the source code is parsed into Abstract Syntax Trees (ASTs). Representative nodes on the ASTs are selected to form token vectors, which represent each source file. These token vectors are then encoded using word embedding, converting them into numerical vectors. The encoded token vectors are inputted into the CNN, which automatically generates semantic and structural features of the source code. Weight sharing and max-pooling techniques are employed to enhance learning efficiency and reduce dimensionality.

The CNN-learned features are combined with traditional defect prediction features. The combined features are then fed into a Logistic Regression classifier for defect prediction. The performance of DP-CNN is evaluated using F-measure on seven open-source Java projects. The experimental results show that DP-CNN improves the state-of-the-art DBN-based method by 12% and traditional features-based method by 16% in terms of defect prediction accuracy. The authors also report that for future work their DP-CNN framework may be extended to other programming languages and applied to other software engineering tasks such as code completion and code clone detection.

3.2.21 "Automatic Traceability Maintenance via Machine Learning Classification" [23]

This paper proposes an approach called TRIAL that aims to deduce links between software documents and their corresponding code in order to give a detailed picture of how a system is constructed and the relationships between the system components. The features used in our implementations of TRAIL can be separated into three categories: IR-based, query quality (QQ), and document statistics features.

For IR-based Features they capture the strength of the link based on IR using two metrics. First,they use software documents as a query and the code as the corpus. After running documents as a query through an IR engine,they capture the rank at which the code appears in the list of results as the first metric. Then they repeat the procedure, this time considering the code as the query, documents as the corpus, and capturing the rank of documents in the list of results as the second metric used to capture the semantic similarity between software artifacts. They use 7 IR approaches, which are Vector Space Model,Latent Semantic Analysis,Latent Dirichlet Allocation, Jensen-Shannon, Okapi BM25, Language Model with Dirichlet (LM-Dirichlet), and Jelinek-Mercer smoothing (LM-JM).

As for Query Quality Features that are estimators for the quality of software artifacts when used as queries for IR,where they can give the classifier more contextual information about the link between two artifacts. For example, if the IR rank of an artifact in a potential link is low, they can give an indication of whether this is due to the artifact being generally hard-to-trace or to the fact that the two artifacts are indeed not related. The metrics can be split into two main categories: pre-retrieval (21 metrics), which can be applied without running the query and capture general properties of the text found in the artifact, and post-retrieval (7 metrics), which also takes into account the ranked list of results returned when running the artifact as a query.

In terms of Document Statistics Features: The three document features used are: the number of unique terms in a document, the total number of terms in a document (including duplicates), and the percentage of overlapping terms between the two documents in a candidate link.

The author then discussed the techniques used for lowering the dimensionality of the features and reducing the probability of overfitting. They are correlation-based feature subset selection, Pearson's correlation, gain ratio, information gain, and symmetrical uncertainty. He then explained the problem he faces with the number of valid and invalid links between artifacts, where the number of invalid links is much larger than the number of valid ones, so he used rebalancing techniques such as the Synthetic Minority Oversampling Technique (SMOTE) and Random Majority Undersampling.

Finally, he uses more than one classifier to deduce which one gives the highest accuracy: k-Nearest Neighbors with k = 5 (5NN), Näive Bayes (NB), Logistic Regression, Random Forest (RF), Support Vector Machines (SVM), and a voting ensemble classifier that combines all of the other algorithms. He also used F1 score to deduce which model gave him highest accuracy , where he found that random forest gave highest accuracy with SMOTE data balancing technique.

3.2.22 "Predicting Query Quality for Applications of Text Retrieval to Software Engineering Tasks" [24]

The main aim of this paper is to predict the quality of the queries presented to text retrieval based approaches. There are mainly 2 types of query quality measures which are pre-retrieval and post-retrieval query measures. Firstly, for the pre-retrieval query measures, there are 21 pre-retrieval measures mentioned in the paper. These are AvgIDF, MaxIDF, DevIDF which are the average, max and standard deviation of the inverse document frequency over all the query terms; AvgICTF, MaxICTF, DevICTF which are the average, max, and standard deviation of the inverse collection term frequency over all the query terms; AvgEntropy, MedEntropy, MaxEntropy, DevEntropy which are the average, median, max and standard deviation of the entropy values over all the query terms; Query Scope which is the percentage of documents that contain at least one term in the query; Simplified clarity score which is the Kullback-Leiber divergence of the language model of the query and the document language model; AvgVAR, maxVAR, SumVAR which are the average, maximum and the sum of the query terms weight variances over the documents that contain the query term for all the query terms; Coherence score which is the average of the similarity between pairs of the documents which contain at least one of the query terms;

AvgSCQ, MaxSCQ, SumSCQ which are the average, max and sum of the document-query similarity score for all the query terms and AvgPMI, MaxPMI which are the average and max pointwise mutual information for all pairs of terms in the query.

Secondly, the post-retrieval query measures and these include Subquery Overlap which is the degree of overlap between the result sets retrieved from each query term in the query and the result set retrieved from the query as a whole; Robustness Score which measures the correlation between the rank before and after modification to the top documents relative to the query terms; First Rank change which captures the likelihood that a document that was located in the top spot in the results list will stay there even after a disturbance is made to it; Clustering Tendency which calculates the textual similarity of the top-retrieved documents to determine how cohesive they are;
and Spatial Auto-correlation which modifies each top-ranked relevant document's retrieval score by taking the average of the scores of the documents that are most similar to it. Subsequently, the new scores are correlated linearly with the old scores. Finally, Weighted information gain which calculates the difference between the average retrieval score of the highest-ranked documents and the total corpus and Normalized query commitment which evaluates the retrieval scores' standard deviation, normalized by the document's overall score are used..

3.3 Implemented approach

In our project we decided to follow two paths , the machine learning path as well as the deep learning path to be used as our benchmark. The machine learning approach starts by applying intensive preprocessing on the use cases of each of the project documents as well as the code documents preparing them for further preprocessing handled by each feature accordingly. Next, apply the following steps :

1. All the interpunction is removed.
2. All numeric characters are removed.
3. All sentences were tokenized with NLTK.
4. The stop words corpus from NLTK is used to eliminate all stop words.
5. All remaining terms were stemmed using the Porter Stemming Algorithm
6. remove keywords in java
7. All words are made lowercase.

Then, integrate a total of 131 features mentioned in the “Predicting Query Quality for Applications of Text Retrieval to Software Engineering Tasks” paper [24] and “Automatic Traceability Maintenance via Machine Learning Classification” paper[23] to ensure the best results. These are composed of :

- 21 Pre-Retrieval Query Measures

- 7 Post-Retrieval Query Measures
- 14 different IR-based features for each possible link
- the Document statistics measurements .

The previous operations are followed by the steps of feature selection and mapping and ending with the data imbalancing. At this point, the features become ready to be fed to the model. We use a Random Forest machine learning model . On the other hand, the deep learning approach starts by calculating the maintainability score of each of the project code documents. It is noted this is a contribution of this work. It is achieved by constructing a set of equations. Using tree setter, we are able to construct a tree to access the code operands and operators and detect the following parameters :

- Program vocabulary: $\eta=\eta_1+\eta_2$
- Program length: $N=N_1+N_2$
- Calculated program length: $\hat{N}=\eta_1\log_2\eta_1+\eta_2\log_2\eta_2$
- Volume: $V=N\log_2\eta$ # Difficulty: $D=\frac{\eta_1}{2}\cdot\frac{N_2}{\eta_2}$
- Effort: $E=D\cdot V$
- Time required to program: $T=E/18$ seconds
- Number of delivered bugs: $B=V/3000$.

and using these parameters to calculate the maintainability score of each document .

Chapter 4:System Design and Architecture

We will begin this chapter with a summary of our project and a list of all the assumptions that were made in the design phase. After that, we discuss our system's design and provide a block diagram to explain the steps required to establish a functioning model and the manner in which Intellitest's modules communicate with one another. Subsequently, we will go into the specifics of every Intellitest module, emphasizing its submodules, capabilities, smaller block diagrams, and design limitations.

4.1. Overview and Assumptions

We will give a summary of the system's underlying architecture and design process in this part. Furthermore, all assumptions made throughout the development process will be acknowledged and supported.

4.1.1 System Inputs Assumptions

In order to be able to construct trace links for the project, which in turn enables the calculation of the performance of the code relevant to the original requirements; we assume the presence of well written and thorough use cases describing the path the code must take. Thus after preprocessing, we can construct trace links between the code documents and the use case documents and accordingly calculate if they did match. Hence, each project repository fed into our project shall be combined with its software requirements documents. Other input assumptions are that the projects are in java.

4.2. System Architecture

The architecture of IntelliTest is covered in this section. We will review each module and display the block diagram of the system to illustrate how the modules work together.

4.2.1. Block Diagram

The expected block diagram of this project consists of 3 Main Parts:

- The Maintainability Score Feature:
  The module is Fed a Java Code Document and a set of calculations is done on the abstract syntax tree for the code using Tree-Sitter , a range of parameters is calculated and combined to be used to output the Maintainability Score of that Code Document

- The Machine Learning Approach for Traceability link generation
  Our approach is composed of 3 stages: Data preprocessing, feature extraction and model building and prediction. Main Input is a Project Source code in Java combined with its set of use case (UC) documents describing the software requirements in the form of bug reports or functional

specifications used to calculate the correlation percentage of these Code documents with the original requirement. Each of the code and the UCs are parsed separately, removing any common keywords, stemming the tokens and removing any punctuation.

- The Deep Learning Approach for Traceability link generation

  For the sake of benchmarking, an equivalent approach for traceability link generation is used. The preprocessed tokens are inserted into a word2vec embedding model to create the feature vector. The feature vector is then passed through embedding, LSTM and linear layers to learn the complex relations between the code files and UCs.

- Bug Localization

  Our Last feature is a bug localization feature that recommends the functions that are faulty based on the bug report imported into the tool. It is implemented as a deep learning approach.

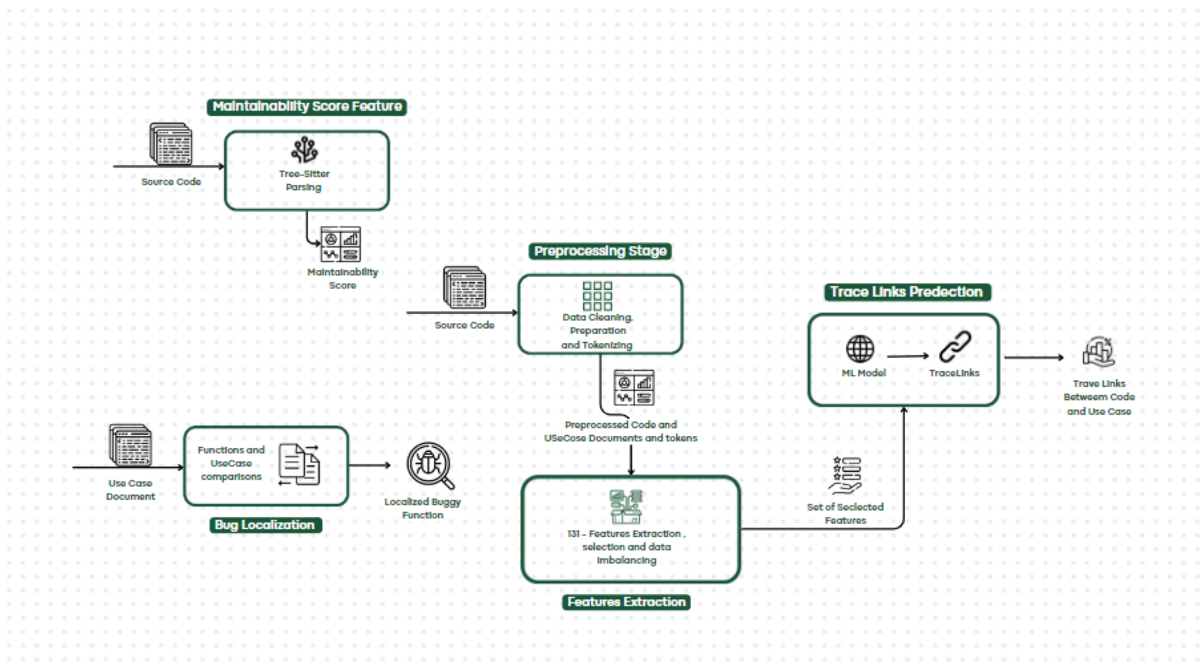


Figure 4-1 : stages block diagram

4.2. Preprocessor Module

This section outlines the functionality, decomposition, and design constraints of the Preprocessor module, which is responsible for preparing source code files and use case (UC) documents for further analysis by cleaning, tokenizing, and filtering content.

4.2.1. Functional Description

The Preprocessor module is designed to process both source code files and use case (UC) documents to prepare them for further analysis. Its primary functionalities include:

Cleaning and tokenizing source code files:

- Removing punctuation marks, numeric characters, and stopwords.
- Lowercasing all words.
- Stemming words using the Porter Stemming Algorithm.
- Filtering out Java keywords to focus on relevant content.

Cleaning and tokenizing use case (UC) documents:

- Removing numeric characters and stopwords.
- Lowercasing all words.
- Stemming words using the Porter Stemming Algorithm.
- Filtering out specific domain-related keywords.

Setting up documents and tokens:

- Reading files from provided paths.
- Storing tokenized documents.
- Generating sets of unique tokens for both source code and UC documents.

4.2.2. Modular Decomposition

The Preprocessor module is composed into the following functions:

- CodePreProcessor: Processes source code files by cleaning, tokenizing, and filtering out irrelevant information based on Java keywords.
- UCPreProcessor: Processes UC documents by cleaning, tokenizing, and filtering out irrelevant information based on specific domain-related keywords.

- setup: Sets up documents and tokens by iterating through files in provided directories, applying preprocessing functions, and collecting tokenized documents and unique tokens.

4.2.3. Design Constraints

Language Constraint : The preprocessing steps are tailored for Java source code, considering its syntax , common conventions Key Words that must differ in other languages the preprocessor is adjusted to work with java codes.

4.3. Maintainability Score (Referred to in Appendix E.2)

This section explains the functionality, decomposition, and design constraints of the Maintainability Score module, which calculates a maintainability score for source code files based on various software metrics.

4.3.1. Functional Description

The Maintainability Score Module is responsible for calculating a maintainability score for source code files based on various metrics.

4.3.2. Modular Decomposition

Parsing source code: Utilizes a Tree-sitter parser for Java to parse the source code files. Identifying the following 4 parameters : operands count, operators count, unique operands count, unique operators count. Using the calculated parameters to compute the following parameters:

Program vocabulary: $\eta=\eta 1+\eta 2$
Program length: $N=N1+N2$
Calculated program length: $\hat{N}=\eta 1 \log 2 \eta 1+\eta 2 \log 2 \eta 2$
Volume: $V=N \log 2 \eta$ # Difficulty: $D=\eta 12 \cdot N2\eta 2$
Effort: $E=D \cdot V$
Time required to program: $T=E/18$ seconds
Number of delivered bugs: $B=V/3000$.

Apply the Maintainability score equation:
Maintainability Score = max(0, (100*(171 - 5.2*np.log(V) - 0.23*G - 16.2*np.log(SLOC))) / 171)

4.3.3. Design Constraints

Dependent on tree sitter as a parsing technique , we tried using another technique called “javalang” to do the parsing but comparing the resulting output we found that tree sitter output matches the needed parameters better .

4.4. Features Extraction Module

This section describes the functionalities, decomposition, and processes involved in the Features Extraction module, which is designed to extract a comprehensive set of features from source code files and use case (UC) documents to support further analysis and machine learning models.

4.4.1. Functional Description

We integrated a total of 131 features mentioned in the “Predicting Query Quality for Applications of Text Retrieval to Software Engineering Tasks” paper and “Automatic Traceability Maintenance via Machine Learning Classification” paper to ensure the best results.

4.4.2. Modular Decomposition

The model starts at using TF-IDF vectorizer in addition to Count Vectorizer to process both source code files and use case (UC) documents to prepare them for further analysis giving us the following :

- TF-IDFMatrix
- IDF Dictionary
- TF Dictionary
- FeatureNames
- DF Dictionary
- CountMatrix

We utilize the generated list to start computing the 131 features as follows:
14-IR-Based Features :

- Vector space model
- Latent semantic analysis
- Latent Dirichlet Allocation
- Jensen-Shannon(JS)

- Okapi BM25
- Language Model with Dirichlet

21 Pre-retrieval Features , each set of features has its own preprocessing function as we use the IDF Arrays multiple times as well as ICTF , Entropy , Variance ,SCQ and PMI :

- AvgIDF
- MaxIDF
- DevIDF
- AvgICTF
- MaxICTF
- DevICTF
- AvgEntropy
- MedEntropy
- MaxEntropy
- DevEntropy
- QueryScope
- SimplifiedClarityScore
- AvgVAR
- MaxVAR
- SumVAR
- CoherenceScore
- AvgSCQ
- MaxSCQ
- SumSCQ
- AvgPMI
- MaxPMI

7 post-retrieval Features :

- SubqueryOverlap using VSM and Jensen Shannon
- RobustnessScore
- FirstRankChange
- ClusteringTendency
- SpatialAutocorrelation
- ScoreDistributionWeightedInformationGain
- NormalizedQueryCommitment

and finally the Document Statistics Features.

Further processing done on the generated features to be normalized and ready to be fed into the model :

1. Remove hardcoded sizing with the size captured from the data set
2. Feature Selection

3. Feature Mapping
4. Data Imbalancing

4.5. Model Prediction Module

This section details the functionality, decomposition, and processes of the Model Prediction module, which uses machine learning to predict trace links between use case (UC) documents and Java source code files.

4.5.1. Functional Description

After applying the feature selection , mapping and data imbalancing to the 131 features extracted we were able to feed the data to the machine learning model for which we picked a random forest model whom we expected matches our scope of data the best and started the prediction process reaching accuracy of 70% and F1-score of 0.7.

4.5.2. Modular Decomposition

Feature Selection:

Feature selection is a critical step in machine learning aimed at identifying and retaining only the most relevant features to improve model performance and reduce computational complexity.

Process Overview

1. **Matrix Shape:** The feature matrices representing the relationship between links and features are examined to understand the dataset's dimensions.
2. **Algorithm Steps:**
    - Reshape: The feature matrix is reshaped to a format suitable for analysis.
    - Correlation Analysis: Correlation coefficients among features are computed to identify highly correlated pairs. Threshold-based Exclusion: Features with correlation coefficients above a predefined threshold (e.g., 0.9) are excluded to mitigate redundancy.
    - Selected Features: The remaining features form the optimized feature set, enhancing model performance.

Results

The feature selection process results in a refined feature set tailored to the dataset's characteristics. Selected features exhibit low correlation, enhancing predictive power while reducing overfitting risk.

Feature Mapping

Feature mapping involves associating each data point with its corresponding features based on predefined indices. This step ensures that the dataset is appropriately aligned with the selected features, facilitating subsequent model training and evaluation.

Process Overview

1. **Data Loading:** The dataset for both training and testing is loaded from CSV files (train_modified.csv and test_modified.csv, respectively) using pandas.
2. **Feature Extraction:**
    - Training Dataset: For each data point in the training dataset, the indices corresponding to its code complexity (CC) and use case (UC) are extracted. These indices are used to access the selected features from the reshaped feature matrix (features_links_selected_reshaped_train). The extracted features are then appended to the Features_train list.
    - Testing Dataset: Similar to the training dataset, the indices for code complexity and use case are extracted for each data point in the testing dataset. The corresponding features are accessed from the reshaped feature matrix (features_links_selected_reshaped_test) and appended to the Features_test list.

Results

- Features_train: A list containing the selected features mapped to each data point in the training dataset.
- Features_test: A list containing the selected features mapped to each data point in the testing dataset.

Data Imbalance

To address data imbalance, BorderlineSMOTE from imblearn library is used. This technique generates synthetic samples for the minority class to balance the class distribution.

Process Resampling:

BorderlineSMOTE is applied to the training dataset to create synthetic samples, achieving a balanced class distribution.

Results

- Features_SMOTE_train: Resampled features with balanced class distribution.
- Labels_SMOTE_train: Corresponding labels for the resampled features.

Model Training and Prediction

A RandomForestRegressor model from the scikit-learn library is employed for training and prediction tasks.
Process

- Data Splitting: The dataset is not split, as indicated by the commented-out train_test_split line, suggesting the use of the entire dataset for training.
- Model Initialization: A RandomForestRegressor model is initialized with specified parameters, including the number of estimators, random state, verbosity, and number of parallel jobs for computation.
- Model Training: The RandomForestRegressor model is trained using the resampled features (Features_SMOTE_train) and corresponding labels (Labels_SMOTE_train).

Results
Model: RandomForestRegressor model trained with the provided dataset providing 70% prediction accuracy and 0.7 F1-score for tracelinks between any use case files and java code files in any project.

4.5. Deep Learning Module

This section describes the functionalities, decomposition, and design constraints of the Deep Learning module, which trains and evaluates a neural network model for traceability link prediction and benchmarking machine learning model outputs.

4.5.1. Functional Description

The Deep Learning Module is designed to train and evaluate a model that processes and analyzes dataset entries for traceability link prediction and benchmarking of the machine learning model outputs. The module performs several key functions:

1. **Data Retrieval and Preparation:** It retrieves and processes datasets from a specified SQLite database. This involves parsing the database, splitting the dataset into training and testing sets, and preprocessing the data for deep learning.
2. **Data Preprocessing:** It prepares the textual data by tokenizing and setting up word embeddings using the Word2Vec model. This includes handling function names, function segments, descriptions, and summaries from the dataset.
3. **Model Training:** It defines and trains a neural network model using the preprocessed data. The model includes an embedding layer initialized with Word2Vec embeddings, LSTM layers for sequence processing, and dense layers for classification.
4. **Model Evaluation:** It evaluates the trained model's performance on a test dataset, calculating accuracy, recall, precision, and F1 score.

5. **Embedding and Vocabulary Management:** It manages the embedding matrix and vocabulary to index mappings, ensuring that the textual data is correctly converted to numerical form for model input.

4.5.2. Modular Decomposition

The Deep Learning Module is decomposed into the following components:

1. **Data Retrieval Component:**
    - DatasetRetrieval: Responsible for retrieving and parsing the dataset from an SQLite database.
    - parseSqlite: Parses the specified SQLite database to extract the dataset.
    - getGlobalDatasetData: Processes the parsed dataset to generate necessary mappings and data structures.
    - splitTestTrain: Splits the dataset into training and testing sets.
2. **Data Preprocessing Component:**
    - PreProcessor: Handles the preprocessing of textual data, including tokenization, embedding setup, and data indexing.
    - setupDeepLearning: Prepares the data for deep learning by tokenizing and segmenting the text.
    - setUpUnknown: Handles unknown tokens and prepares the data accordingly.
    - word2VecProcessor: Generates word embeddings using the Word2Vec model.
    - vocabToIndex and dataSetToIndex: Convert vocabulary and datasets to index-based numerical form.
3. **Model Definition Component:**
    - DLModel: Defines the deep learning model architecture, including embedding layers, LSTM layers, and dense layers for classification.
4. **Model Training Component:**
    - train: Handles the training process of the model, including data loading, loss calculation, backpropagation, and weight updates.
5. **Model Evaluation Component:**
    - evaluate: Evaluates the model's performance on a test dataset, calculating accuracy, recall, precision, and F1 score.
6. **Utilities Component:**
    - customCollate: Custom collate function for data loading.
    - weighted_binary_cross_entropy: Custom loss function for weighted binary cross-entropy.

4.5.3. Design Constraints

The design of the Deep Learning Module is subject to the following constraints:

1. Data Dependency: The module relies on the availability of a specific dataset stored in an SQLite database. The structure and content of this database must be consistent with the expected format.
2. Computational Resources: Training deep learning models, especially with LSTM layers, is computationally intensive. The module requires access to significant computational resources, including GPU support for efficient training.
3. Model Complexity and Performance: The design of the model, including the number of layers and hidden units, must balance complexity and performance. Overly complex models may lead to overfitting, while simpler models may underfit the data.
4. Handling of Unknown Tokens: The module must handle unknown tokens effectively to ensure that the model can process previously unseen data without errors.
5. Embedding Quality: The quality of the Word2Vec embeddings significantly impacts the model's performance. The embeddings must be trained or fine-tuned on relevant data to capture meaningful semantic information.
6. Loss Function and Class Imbalance: The module must use an appropriate loss function and handle class imbalances in the dataset. The use of weighted binary cross-entropy helps mitigate the impact of imbalanced classes.
7. Evaluation Metrics: The module must implement robust evaluation metrics to assess the model's performance comprehensively, including accuracy, recall, precision, and F1 score.

4.6.Bug Localization Module

This section outlines the functionalities, decomposition, and design constraints of the Bug Localization module, which identifies and locates specific code changes associated with resolved bug reports in a software project.

4.6.1. Functional Description

The Bug Localization module identifies and locates the specific code changes associated with resolved bug reports in a software project. It processes Jira issues to extract relevant commit information, identifies the modified code between comits, and compares different versions of the code to determine the exact changes. These changes are then tokenized and prepared for further analysis, which helps in predicting and understanding bug-prone areas in the codebase.

4.6.2. Modular Decomposition

The Bug Localization module can be decomposed into the following sub-modules:

1. **Bug Report Retrieval**
    - **Description**: Connects to the SQLite database and retrieves summary and description of bug reports marked as resolved or closed.
    - **Inputs**: SQLite database containing the issues information
    - **Outputs**: DataFrame of resolved bug reports, containing its description , summary, issue IDs.
2. **Commit Hash Retrieval**
    - **Description**: Fetches commit hashes associated with the resolved bug reports.
    - **Inputs**: SQLite database containing
    - **Outputs**: DataFrame containing issue IDs and corresponding commit hashes.
3. **Commit Data Extraction**
    - **Description**: Uses the GitHub API and GitPython to fetch commit details and access the local repository clone. It extracts the content from the commited file before and after solving the bug.
    - **Inputs**: Commit hashes, GitHub API token, local repository path.
    - **Outputs**: List of tuples containing issue ID, the file before and after solving the bug.
4. **Code Parsing**
    - **Description**: Parses the Java files to extract method-level changes using the Tree-Sitter library. It identifies segments of code that have been modified between the old and new versions of the file.
    - **Inputs**: List of tuples containing issue ID, the file before and after solving the bug.
    - **Outputs**: List of methods that are changed when solving the bug with labels indicating that they are buggy, and the unchanged methods with label indicating they are not buggy.
5. **Tokenization and Preprocessing**
    - **Description**: Tokenizes the extracted code segments and other relevant information (e.g., method name, method body, summaries, and descriptions) using predefined preprocessing functions. It prepares the data for deep learning models.
    - **Inputs**: List of methods changed, issues description and summary.
    - **Outputs**: Tokenized data ready for model training and testing.
6. **Word Embedding Training**

- **Description**: Trains a Word2Vec model on the tokenized data to create word embeddings for the vocabulary used in the code and bug reports.
- **Inputs**: Tokenized data.
- **Outputs**: Trained Word2Vec model, embedding matrix.

7. **Data Indexing**
    - **Description**: Converts the tokenized data into numerical indices using the vocabulary generated from the Word2Vec model. This prepares the data for input into deep learning models.
    - **Inputs**: Tokenized data, Word2Vec vocabulary.
    - **Outputs**: Indexed data.

4.6.3. Design Constraints

Data Availability: The module relies on the availability of a comprehensive dataset containing detailed bug reports and commit information.
API Rate Limits: The use of the GitHub API is subject to rate limits, which can constrain the frequency of commit data extraction.
Computational Resources: Parsing and processing large codebases and training word embedding models require significant computational resources.
Accuracy of Tokenization: The effectiveness of the module depends on the accuracy of the tokenization and preprocessing functions, which must handle various programming constructs and natural language descriptions.
Security: Access to the GitHub repository and database must be secure to protect sensitive information and prevent unauthorized access.

Chapter 5: System Testing and Verification

5.1. Testing Setup

A customized testing environment that closely mimics the production environment makes up our testing configuration. It comes with all the dealing with complete source code projects with the use case and code files structure required to replicate actual use cases. To help with testing and deployment procedures, we make use of containerisation technologies and virtualized environments.

5.2. Testing Plan and Strategy

5.2.1. Module Testing

We use a methodical methodology to test each project module individually when it comes to module testing. This involves

- Making sure that the output of the preprocessing functions conforms to the standards required, which is ensuring that the data is clean and tokenized enough to enable extracting features.
- Verifying that our vocabulary is constructed from our dataset as expected and that the unknown words are set correctly.
- Ensuring that every feature follows a logical distribution depending on its nature.

Findings:

- Removing all necessary unwanted tokens.
- Cosine similarity feature outputs values ranging from -1 to 1, RobustnessScore feature outputs 0 or 1, and this applies for he remaining features
- A resonable percentage of unknwon words is set in the dataset.

5.2.2. Integration Testing

In order to verify smooth communication and compatibility across various modules, integration testing concentrates on testing the integrated system as a whole. This includes:

- verifying how the results from the preprocessing act along with the features being extracted functions as inputs and outputs and how they are reshaped to be inputted to the machine and deep learning models and how all is integrated to the desktop app working fine with the APIs and the UI, This is done by:
  - Testing our ML and DL model on 0.2 of our dataset, and calculated its precision, recall, and f1score
  - we made a validation dataset to calibre the model hyperparamters, to maximize the model f1score.

Results:

- DL Model:

Project Name	precesion	Recall	F1score
Errai	0.85	0.86	0.85

Jboss	0.78	0.80	0.79
RestEasy	0.85	0.89	0.87
All Projects Integrated	0.84	0.87	0.85

Table 5.3 :Testing Schedule

- ML Model:

Project Name	precesion	Recall	F1score
Errai	0.70	0.82	0.75
Jboss	0.68	0.78	0.72
RestEasy	0.75	0.8	0.77

Table 5.4 :Testing Schedule

5.3. Testing Schedule

Modules Tested	Date
PreProcessing	28th of april
Features Module	2nd of May
Model Prediction	12th of May
Whole Pipeline	20th of May
App UI and APIs	28th of May

Table 5.5 :Testing Schedule

5.4. Comparative Results to Previous Work

In this section we are discussing the previous researches approaches achieving similar modules to ours

Module 1: Maintainability Score Module

Current Implementation: The Maintainability Score Module currently uses Tree-sitter to parse Java source code and calculates maintainability based on various code metrics such as operands count, operators count, and program volume.

Alternative Approach: Using Deep Learning for Maintainability Prediction Triet et al. (2020) reviewed deep learning methods for source code modeling and generation. Implementing their approach could enhance the maintainability prediction by leveraging deep learning models to understand and evaluate code maintainability.

Reflections:

- **Automation:** Deep learning models can automate the extraction of features and identification of code patterns associated with maintainability, reducing the manual effort involved in defining metrics.
- **Accuracy:** Deep learning can improve the accuracy of maintainability predictions by learning from a large dataset of code samples and associated maintainability scores.
- **Scalability:** Deep learning models can efficiently handle large codebases, making them suitable for projects with extensive and complex code.

Source: Triet et al. (2020). "Deep learning for source code modeling and generation: Models, applications, and challenges."

Module 2: Preprocessor Module

Current Implementation: The Preprocessor module processes both source code files and use case (UC) documents by cleaning, tokenizing, and filtering out irrelevant information.

Alternative Approach: Using NLP for Code and Requirement Traceability Blasi et al. (2022) demonstrated the use of NLP techniques for generating trace links between requirements and code. Implementing their approach could enhance the preprocessor module's ability to establish traceability links.

Reflections:

- **Efficiency:** Using NLP can streamline the preprocessing of both code and use case documents, automating the generation of trace links and reducing manual effort.
- **Consistency:** NLP techniques ensure consistent preprocessing and trace link generation, improving the reliability of the traceability analysis.

- **Temporal Constraints:** By incorporating NLP, the preprocessor can handle temporal constraints more effectively, ensuring the traceability of code changes over time.

Source: Blasi et al. (2022). "Using NLP to Automatically Generate Unit Test Cases that Respect Temporal Constraints."

Module 3: Bug Localization Module

Current Implementation: The Bug Localization module identifies and locates specific code changes associated with resolved bug reports by extracting commit information and parsing code changes.

Alternative Approach: Applying Machine Learning for Fault Prediction Medeiros et al. (2020) used machine learning models combined with software metrics to detect vulnerable code. Implementing their approach could enhance the bug localization module's ability to predict and identify bug-prone areas.

Reflections:

- **Accuracy:** Machine learning models can improve the accuracy of bug localization by identifying complex patterns and correlations in code changes and bug reports.
- **Preventive Action:** Enhanced fault prediction can facilitate proactive bug fixing, reducing the time and effort required for debugging.
- **Adaptability:** Machine learning models can be continuously trained and updated with new data, ensuring they remain effective in detecting new types of bugs and vulnerabilities.

Source: Medeiros et al. (2020). "Machine Learning Model to Detect Vulnerable Code."

Chapter 6: Conclusions and Future Work

6.1. Faced Challenges

Throughout the project, we faced significant issues, mainly because our machines struggled to handle the huge amount of data we were working with.

Processing all that data took a lot of time, and the more complex tasks like extracting features and training models made things even slower. We had to find ways to balance doing thorough analysis while also not waiting forever for our computers to finish. It was tough, but by planning carefully and finding ways to make things run smoother, we managed to overcome these challenges and get good results in the end.

6.2. Gained Experience

6.2.1. Learning from Research:

Our capacity to comprehend and evaluate the most advanced research papers and articles was one of the most important lessons we learned. Our learning of different methods to problem-solving was expanded, and we also learned useful reading techniques to fill in comprehension gaps. With the use of this ability, we were able to compile and contrast study data and select the best strategy for our project.

6.2.2. Dealing with Features(referred to in code in appendix E.1) :

We obtained practical feature engineering knowledge by merging 131 features from prestigious research articles. By using document processing techniques like TF-IDF and Countvectorizer, we were able to overcome obstacles like feature selection and normalization. We have gained a deeper grasp of feature importance through experimenting with different statistical methods. Furthermore, resolving problems such as data imbalance and hardcoded size improved our ability to refine feature extraction processes. This experience helped us better understand how important feature engineering is to machine learning, which will help us with future work.

6.2.3. Promoting Teamwork and Engagement:

Our project required good communication and collaboration from all involved parties. We improved our cooperation abilities by exchanging ideas, actively listening, and giving helpful criticism. We were able to maximize our productivity and success as a team by dividing up the work, staying in constant communication, and getting advice from Dr. Nevin.

6.3. Conclusions

The project successfully demonstrated the application of machine learning techniques to trace requirements documentation to source code files and vice versa and locate bugs in large software projects. By combining various technologies, including statistics, classifiers, machine learning models, and natural language processing, we developed a system that streamlines the tracing and bug detection process, making it faster and more cost-effective. However, the system's reliance on extensive computational resources and the challenges in feature extraction were notable limitations. Despite these challenges, the project provided valuable insights and a solid foundation for future improvements.

6.4. Future Work

We want to improve our system in the future by adding features related to **bug localization**, **error repair** and **error detection**. Through the application of the knowledge gathered from our research, we hope to create reliable methods for locating problems in the codebase and recognising them, which will enable more effective debugging procedures. Furthermore, our goal is to incorporate **mistake correcting features** in order to effectively address problems that are found. In addition, we hope to provide in-place code editing capabilities so that developers may make quick edits right in the code environment. These improvements will help to improve software systems' overall maintainability and dependability in addition to streamlining development operations. We work to improve our system through constant iteration and improvement so that it can better serve the changing requirements of software development.

References

[1] [Blasi et al 2022], Blasi A., Gorla, A., Ernst, M., & Pezzè, M]. (2022). Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints. *ACM Transactions on Software Engineering and Methodology (ACM TOSE). https://doi.org/10.1145/3551349.3556961 (last accessed on December 27, 2023)*

[2] [Ostrand et al 2010], Ostrand, T. J., & Weyuker, E. J. (2010). *Software fault prediction tool. Proceedings of the 19th International Symposium on Software Testing and Analysis - ISSTA . https://dl.acm.org/doi/abs/10.1145/1831708.1831743 (last accessed on December 27, 2023)*

[3] [Medeiros et al 2020], Medeiros, N., Ivaki, N., Costa, P., & Vieira, M. (2020). Vulnerable Code Detection Using Software Metrics and Machine Learning. IEEE Access, 8, 219174-219198. *https://doi.org/10.1109/ACCESS.2020.3041181 (last accessed on December 27, 2023)*

[4] [Triet et al 2020], Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. ACM Comput. Surv. 53, 3, Article 62 (June 2020), 38 pages. *https://doi.org/10.1145/3383458 (last accessed on December 27, 2023)*

[5] [J.Zhang et al 2019], J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 783–794. *https://doi.org/10.1109/ICSE.2019.00086 (last accessed on December 27, 2023)*

[6] [Talha et al 2019], Talha Burak Alakus, Resul Das, Ibrahim Turkoglu, 2019. An Overview of Quality Metrics Used in Estimating Software Faults , International Artificial Intelligence and Data Processing Symposium (IDAP) , 6 pages. *https://ieeexplore.ieee.org/document/8875925 (last accessed on December 27, 2023)*

[7] [Rituraj et al 2020], Rituraj Singh, Jasmeet Singh, Mehrab Singh Gill , Transfer Learning Code Vectorizer based Machine Learning Models for Software Defect Prediction, 2020 International Conference on Computational Performance Evaluation (ComPE) North-Eastern Hill University, Shillong, Meghalaya, India, 6 pages *https://ieeexplore.ieee.org/document/9200076 (last accessed on December 27, 2023)*

[8] [Wikan et al 2019], Wikan Danar Sunindyo, Fernando Maruli Tua, Software Defect Prediction Using Software Metrics with Naïve Bayes and Rule Mining Association Methods, 2019 5th International Conference on Science and Technology (ICST), Yogyakarta, Indonesia,5 pages https://ieeexplore.ieee.org/document/9166448 *(last accessed on December 27, 2023)*

[9] [Rathore et al 2021], Rathore, S.S., Kumar, S. Software fault prediction based on the dynamic selection of learning technique: findings from the eclipse project study. Appl Intell 51, 8945–8960 (2021). 16 pages, https://link.springer.com/article/10.1007/s10489-021-02346-x#citeas *(last accessed on December 27, 2023)*

[10] [Ansari et al 2017], Ansari, A., Shagufta, M. B., Sadaf Fatima, A., & Tehreem, S. . Constructing Test cases using Natural Language Processing. 2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB),Chennai, India,5 pages. https://doi.org/10.1109/aeeicb.2017.7972390 *(last accessed on December 27, 2023)*

[11] [Sascha et al 2020], Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid . Fast static analyses of software product lines: an example with more than 42,000 metrics. 2020 In Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20). Association for Computing Machinery, New York, NY, USA, 9 pages . https://doi.org/10.1145/3377024.3377031 *(last accessed on December 27, 2023)*

[12] [Wang et al 2022]Wang, K., Yan, M., Zhang, H., & Hu, H.. Unified abstract syntax tree representation learning for cross-language program classification. 2022 Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. Pittsburgh, PA, USA, 11 pages ,https://doi.org/10.1145/3524610.3527915 *(last accessed on December 27, 2023)*

[13] [A. Viet et al 2017] , A. Viet Phan, M. Le Nguyen and L. Thu Bui, Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction,2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI), Boston, MA, USA, 8 pages , https://ieeexplore.ieee.org/document/8371922 *(last accessed on December 27, 2023)*

[14] [T. Zimmermann et al 2007], T. Zimmermann, R. Premraj and A. Zeller, Predicting Defects for Eclipse,2007 Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007), Minneapolis, MN, USA,7 pages , https://ieeexplore.ieee.org/document/4273265 *(last accessed on December 27, 2023)*

[15] [S. Mondal et al 2023]S. Mondal, A. K. Sahu, H. Kumar, R. M. Pattanayak, M. K. Gourisaria and H. Das, Software Fault Prediction using Wrapper based Ant Colony Optimization Algorithm for Feature Selection, 2023 6th International Conference on Information Systems and Computer Networks (ISCON),6 pages Mathura, India, pages *(last accessed on December 27, 2023)*

[16] [Tufano et al 2021], Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S.K., & Sundaresan, N. (2021). Unit Test Case Generation with Transformers. 15 pages , https://www.semanticscholar.org/paper/Unit-Test-Case-Generation-with-Transformers-Tufano-Drain/b82287ae96f67ff12daebb59d953b4b08a9c1e02 . *(last accessed on December 27, 2023)*

[17] [A. Rasyid et al 2022] A. Rasyid, M. J. Alibasa, N. Selviandro and Y. Priyadi, Source Code Preprocessing Method Analysis in Unit Test Code Classification, 2022 1st International Conference on Software Engineering and Information Technology (ICoSEIT), Bandung, Indonesia, 5 pages, https://ieeexplore.ieee.org/document/10030023 . *(last accessed on December 27, 2023)*

[18] [H. Menendez et al 2022] H. D. Menendez, M. Boreale, D. Gorla and D. Clark, Output Sampling for Output Diversity in Automatic Unit Test Generation, in IEEE Transactions on Software Engineering,14 pages, 1 Jan. 2022,https://ieeexplore.ieee.org/document/9068446 . *(last accessed on December 27, 2023)*

[19] [Baishakhi et al 2016], Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "naturalness" of buggy code. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16). Association for Computing Machinery, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848 *(last accessed on December 27, 2023)*

[20] [Jian et al 2017], Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu, Software Defect Prediction via Convolutional
Neural Network. 2017 IEEE International Conference on Software Quality, Reliability and Security, 11 pages Software Defect Prediction via Convolutional Neural Network | IEEE Conference Publication | IEEE Xplore *(last accessed on December 27, 2023)*

[21] [Anbuazhagan et al 2022] Anbuazhagan, S. (2022, September 19). A Complete Guide to train Multi-Layered Perceptron Neural Networks | Medium. Medium. https://paarthasaarathi.medium.com/a-complete-guide-to-train-multi-layered-perceptron-neural-networks-3fd8145f9498 *(last accessed on December 27, 2023)*

[22] [Bhandari et al 2023] , Bhandari, A. (2023, December 27). Guide to AUC ROC Curve in Machine Learning : What is specificity? Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning *(last accessed on December 27, 2023)*

[23] C. Mills, S. Haiduc, J Escobar-Avila "Automatic Traceability Maintenance via Machine Learning Classification",12 pages , 2018 https://ieeexplore.ieee.org/document/8530044/citations?tabFilter=papers#citations *(last accessed on March 27, 2023)*

[24] C. Mills, G. Bavota, S. Haiduc, R. Oliveto, A. Marcus, and A. D. Lucia, "Predicting query quality for applications of text retrieval to software engineering tasks," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 26, no. 1, p. 3, 2017. https://dl.acm.org/doi/10.1145/3078841 *(last accessed on March 27, 2023)*

[25] Requirements Traceability Links (n.d.). Requirements. Traceability Links Retrieved from https://www.reqview.com/doc/requirements-traceability-links/ *(last accessed on [June 7,2024])*

[26] Symflower. (n.d.). SymFlower. Retrieved from https://www.symflower.com/ *(last accessed on [June 6,2024])*

[28] Visual Studio (n.d.). Visual Studeio matrices Retrieved from https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022 *(last accessed on [June 7, 2024])*

[29] Leach, R. J. (1997). A software metric for logical errors and integration testing effort. In Computer Assurance, 1997. COMPASS '97. 'Are We Making Progress Towards Computer Assurance?'. Proceedings of the

12th Annual Conference on. DOI: 10.1109/CMPASS.1997.613302. Retrieved from IEEE Xplore *(last accessed on June 7, 2024).*

[30] Workneh, H., & Reddivari, S. (2023). A Machine Learning based Traceability Links Classification: A Preliminary Investigation. In 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC). DOI: 10.1109/COMPSAC57700.2023.00141. Retrieved from IEEE Xplore *(last accessed on June 7, 2024).*

[31] Lin, J., Liu, Y., Zeng, Q., Jiang, M., & Cleland-Huang, J. (2021). Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). Retrieved from IEEE Xplore *(last accessed on June 7, 2024).*

[32] Keim, J., Corallo, S., Fuchß, D., Hey, T., Telge, T., & Koziolek, A. (n.d.). Recovering Trace Links Between Software Documentation and Code. Karlsruhe Institute of Technology, Karlsruhe, Germany.*(last accessed on June 7, 2024).*

[33] Lin, J., Poudel, A., Yu, W., Zeng, Q., Jiang, M., & Cleland-Huang, J. (2022). Enhancing Automated Software Traceability by Transfer Learning from Open-World Data. arXiv:2207.01084v1 [cs.SE]. Retrieved from https://arxiv.org/abs/2207.01084 *(last accessed on June 7, 2024).*

[34] Keim, J., Corallo, S., Fuchß, D., Hey, T., Telge, T., & Koziolek, A. (n.d.). Recovering Trace Links Between Software Documentation and Code. Karlsruhe Institute of Technology, Karlsruhe, Germany. *(last accessed on June 7, 2024).*

[35] Fuchß, D., Corallo, S., Keim, J., Speit, J., & Koziolek, A. (n.d.). Establishing a Benchmark Dataset for Traceability Link Recovery between Software Architecture Documentation and Models. KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Karlsruhe, Germany. *(last accessed on June 7, 2024).*

Appendix A: Development Platforms and Tools

A.1. Software Tools

A.1.1. Visual Studio Code

We will use Visual Studio Code as our IDE

A.1.2. Python

We are expected to use Python programming language in all our stages for the development

A.1.3. Jupyter Notebook

We will use Jupyter Notebooks in order to test our python code

A.1.4. NumPy

We are expected to use NumPy Library. ”NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.”

A.1.5. Pandas

We are expected to use Pandas Library , “pandas is a software library written for the Python programming language for data manipulation and analysis.”

A.1.6. TensorFlow

We are expected to use the Tensorflow Library, “TensorFlow is a free and open-source software library for machine learning.”

A.1.7. Tree-sitter

Tree-sitter is a parser generator tool and an incremental parsing library. It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited.

A.1.8. Requests

Requests is a Python library used for making HTTP requests. It simplifies the process of sending HTTP requests and handling responses, making it easier to interact with web services and APIs for tasks such as retrieving data or posting information.

A.1.9. SQLite

SQLite is a self-contained, serverless, and zero-configuration database engine. It is used for managing the project's database, providing a lightweight and reliable solution for storing and retrieving data. SQLite is particularly suited for small to medium-sized applications and for use in development and testing environments.

A.1.10. Pickle

Pickle is a Python module used for serializing and deserializing Python object structures. It is used for saving the state of a program or a model to a file, allowing it to be restored and reused later, which is helpful for saving machine learning models and other data structures.

A.1.11. Flask

Flask is a lightweight web framework for Python. It is used for developing web applications and APIs, providing a simple yet flexible platform for creating web-based interfaces and services.

A.1.12. GitHub

GitHub is a web-based platform that provides hosting for Git repositories. It is used for version control, collaboration, and project management. GitHub facilitates code review, issue tracking, and integration with other development tools, thereby enhancing team collaboration and project transparency.

A.2. Hardware Platforms

Since software tools are the primary component of our concept, no hardware platforms are required.

Appendix B: Use Cases

As mentioned, our targeted users are free-lancing software developers and software developers belonging to a software development corporation, yet the use cases of benefit that can be utilized from the application differ.

B.1. Automating Trace Link Generation for Java Documentation and Use Case Documents

B.1.1 Software Developers

Primary Users: Software Developers involved in maintaining and verifying traceability between requirements and code.

Purpose: To enable software developers to automatically generate trace links between use case documents and Java code, ensuring that all requirements are effectively traced to their implementation. This helps in maintaining consistency and validating that all requirements are met in the codebase.

Scenario: The software developer avoids the need to backtrack the code step by step as the trace links module will be completely responsible for applying feature extraction and selection,Data imbalancing and prediction of the correlation scores provide the developer with an easy path into detecting the Java file responsible for the bug, where he can start his fixing process.

B.1.2 Quality Assurance Analysts

Primary Users: Quality Assurance (QA) Analysts responsible for validating software functionality and compliance.

Purpose: To assist QA analysts in validating that all requirements are correctly implemented and traceable, ensuring the overall quality and compliance of the software product.

Scenario: A QA analyst is responsible for assessing the maintainability of a software project's codebase to ensure it meets quality standards and is easy to manage. The analyst uses the maintainability score feature to automatically evaluate each Java file in the project. The feature calculates scores based on various maintainability metrics, such as code complexity, readability, and adherence to coding standards. The analyst receives a detailed report highlighting files with low scores, indicating potential maintenance issues. The analyst then works with developers to address these issues, improving the overall quality and maintainability of the codebase and reducing future maintenance efforts.

B.2. Maintainability Scores use (Referred to in appendix E.2)

B.2.1 Project Managers

Primary Users: Project Managers overseeing the software development process.

Purpose: To provide project managers with insights into the maintenance of each code file in the whole project, giving them bugger insights on which files need to be resent to the developers to be restructured to have a maintenance structure for further code updates and new features.

Scenario: directly accessing the maintainability score feature, automating the process of calculating the score, followed by reassigning the documents to the Java developers instead of manually revising all the source code files one by one to see which of them doesn’t follow a proper maintainable architecture.

Appendix C: User Guide

In this section, we will guide you through how to use our application and how, through Intellitest, you can get your required calculations and utilize our features step by step.

C.1. HomePage and Project uploading

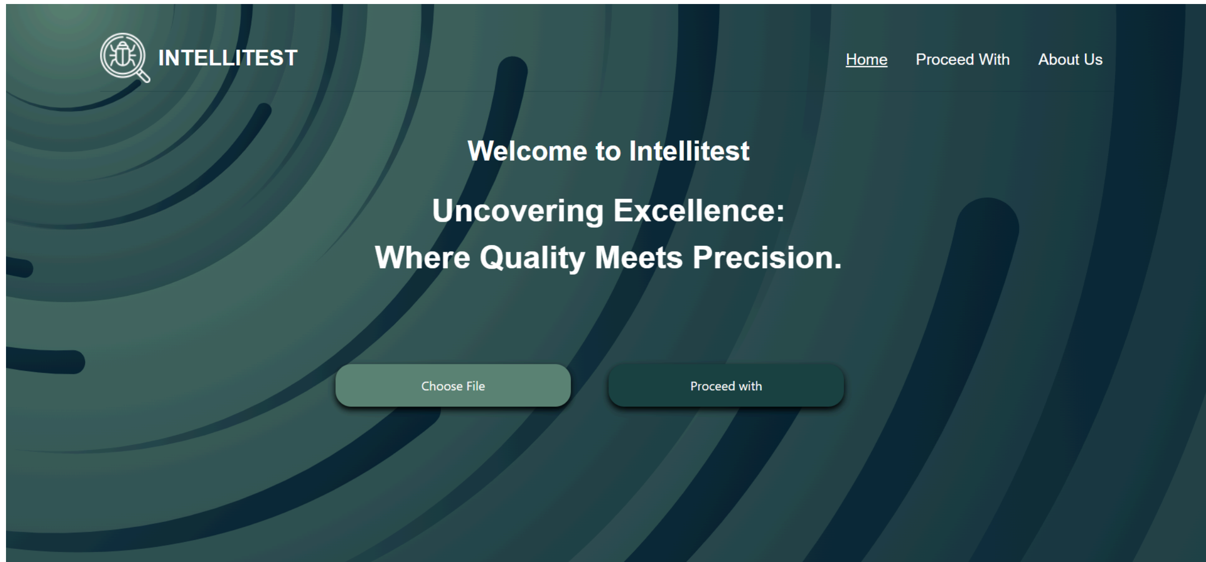


Figure C-1.1 : Home Page

The program starts at the home page, where you have the option of uploading your

source code folder using the “Choose File” button or proceeding with the last uploaded project using the “Proceed With” button, the projects shall be uploaded in the zipped format they get downloaded from the github repos, and the program is responsible for

- Creating the needed directories for further processing
- Creating the use case documents from the sqlite file
- Getting the structure prepared for further features

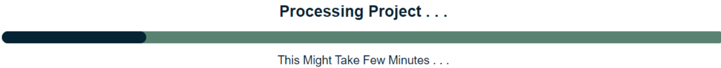
and that uploading process might take few minutes

INTELLITEST

Home Import Project Proceed With

INTELLITEST

Processing Project . . .



This Might Take Few Minutes . . .

Figure C-1.2 : Loading Page

C.2. Code Editor

After the project is successfully uploaded, you are automatically directed to the code editor, where you can view the file structure and the file content. From there, you can proceed with any of the program features listed in the header since the project has been uploaded.

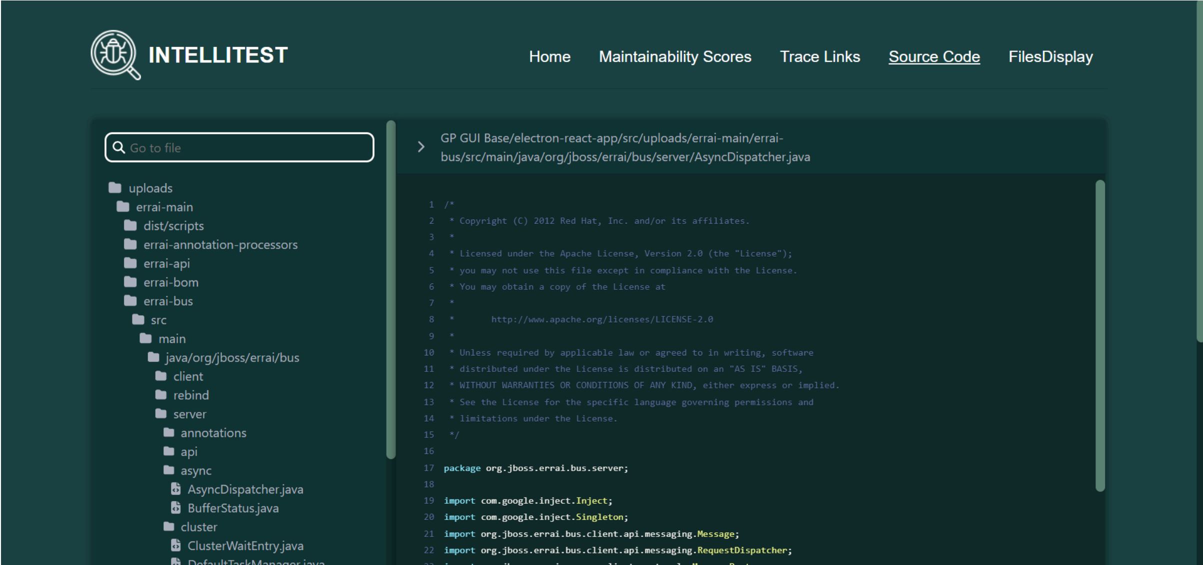


Figure C-2.1 :Code editor

C.3. TraceLinks

Starting with the main feature, compute the trace links between the Java code files and corresponding use case documents to produce a correlation score indicating if the Java code is highly correlated with the software document.

The first option shown in the following figure is choosing a local existing use case file from the ones created using the sqlite file.

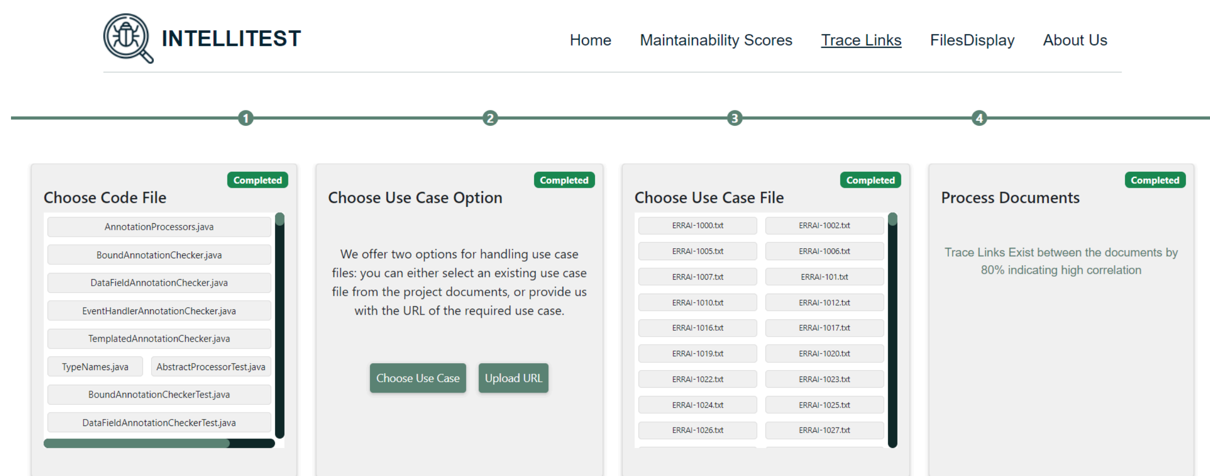


Figure C-3.1 :TraceLinks Page

The second option is uploading a URL to an issue where it gets parsed with our program and the information needed to be extracted from it as

- description
- summary
- issue type
- version

are parsed and passed to the trace links computing module.

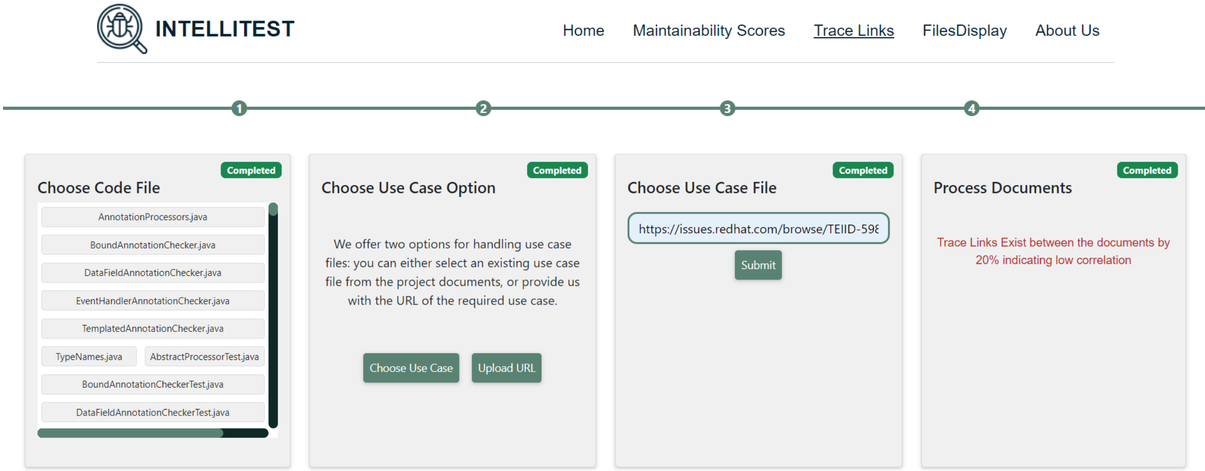


Figure C-3.2 :TraceLinks Page-Invalid

C.4. Issues Files Display

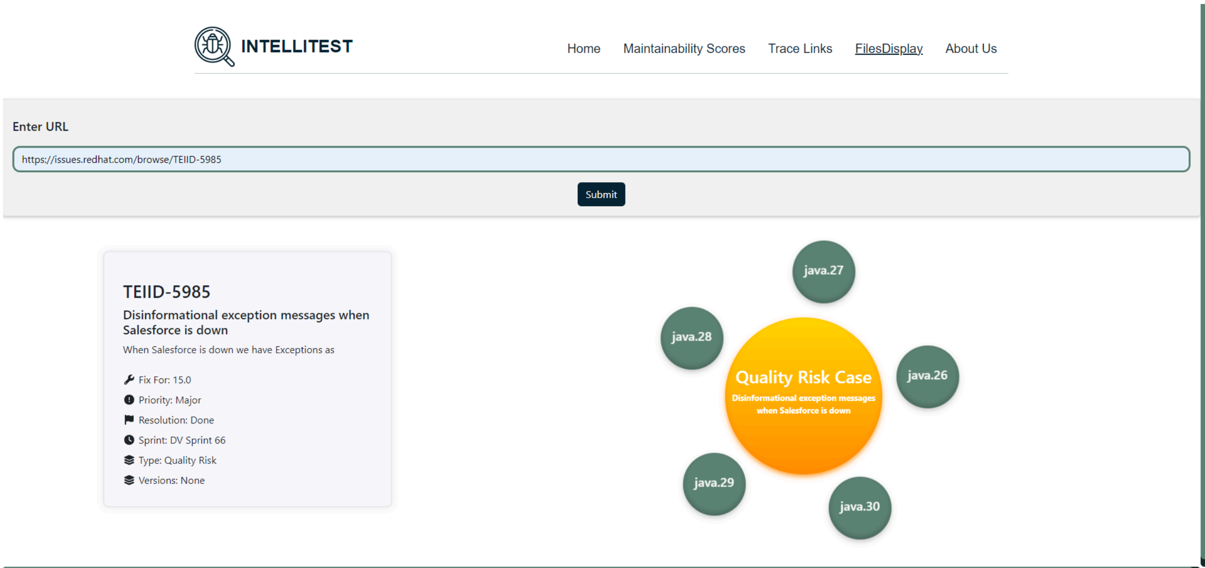


Figure C-4.1 :Files Display Page - Quality Risk Example

Now we come to the processing of tracking down each issue using the uploading urls functionality and extracting the required information as mentioned before, where another model is responsible for computing the correlation between that use case file

and all the corresponding Java code files, returning the highest 5 correlated Java files to be checked according to the use case type.

We have five issue types.

- Bug
- Quality Risk
- Enhancement
- Feature Request
- Sub-Task

Figure C-4.2 :Files Display Page - Bug Example

In cases of bug issues, we will have the ability to click on the highest-correlated Java files to be redirected to their source code and utilize the bug localization feature as shown in the figure above.

C.5. Bug Localization

as shown in the image below

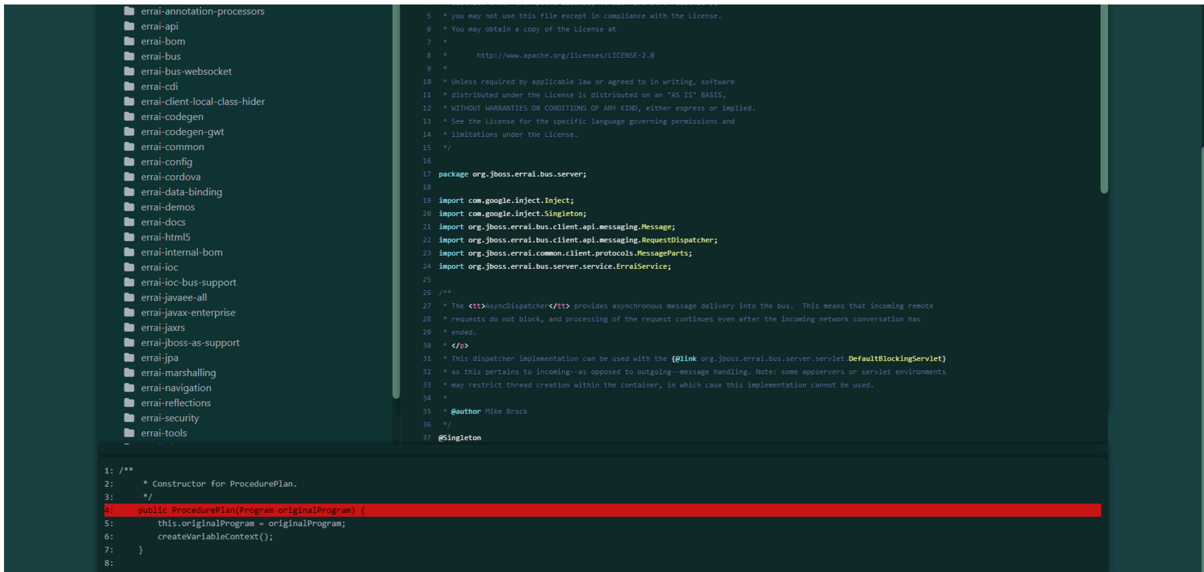


Figure C-5.1 :Bug Localization Page

After getting redirected into the source code of the correlated Java code, the bug localization feature was able to point out the bugged function that originally caused the issue.

C.6. Maintainability Score (Referred to in appendix E.2)

Moving to our final feature, computing the maintainability score for each Java file, we can easily detect the files that are not maintainable and decide if they need enhancement based on the computed score, as shown in the image below.

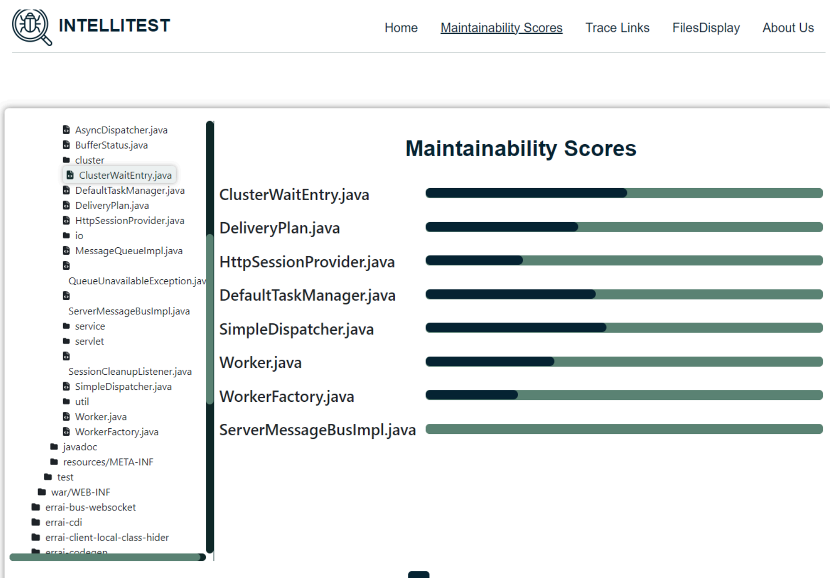


Figure C-6.1 :Maintainability score page

The tree file structure is available for users to surf and choose any Java file by clicking on it, and a progress bar with the computed score will be displayed.in addition to viewing a card by hovering over that progress bar with all the calculated parameters for that specific Java file, including

- program vocabulary
- program length
- Volume
- difficulty
- effort
- time required to compute the program
- number of delivered bugs
- SLOC
- Comment line ratio
- cyclomatic complexity

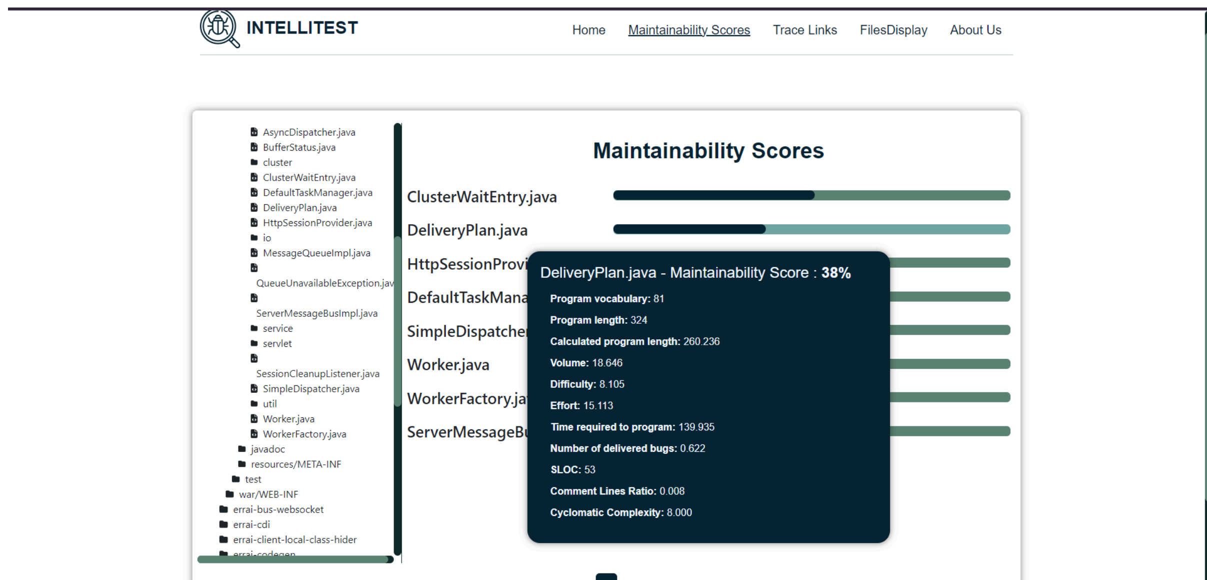


Figure C-6.2 :Maintainability score page-cards

Appendix D: Feasibility Study

This feasibility study examines whether we can successfully develop a tool to automate the creation of trace links between Java code and use case documents. The goal is to save time and improve accuracy in tracking how requirements are implemented in the code. We will look at the technical requirements, costs, legal considerations, and whether we have the resources and time to complete the project. This study will help us decide if this project is worth pursuing and how to go about it.

D.1. Technical Feasibility

Current Technologies and Tools:

- **Machine Learning Algorithms:** Tools like RandomForestRegressor for prediction tasks.
- **Data Processing Libraries:** Python libraries such as pandas and scikit-learn for data handling and machine learning.
- **Feature Extraction Techniques:** Automated methods for extracting relevant features from documents.

- **Development Frameworks:** Java for codebase handling and Python for machine learning tasks.

Required Skills:

- Proficiency in Python and Java.
- Experience with machine learning and data processing.
- Knowledge of software development and documentation practices.

Technical Challenges:

- Integrating machine learning models with existing development workflows.
- Handling diverse and complex datasets from use case documents and Java code.
- Ensuring the accuracy and reliability of generated trace links.

D.2. Economic Feasibility

Category	Details	Estimated Cost	Potential Benefits
Development Costs	Includes salaries, software licenses, and equipment.	$50,000	
Operational Costs	Annual costs for maintenance, updates, and support.	$10,000	
Training Costs	Training staff to use and maintain the system	$5000	
Short-term Benefits	Increased efficiency and reduced manual efforts.	-	Reduced time and effort in generating trace links. Lower costs for manual traceability.
Long-term Benefits	Continued savings and productivity	-	Increased accuracy and

	improvements.		compliance, reducing risks of delays and rework.

Table D.2 :Feasibility study benefits

D.3. Legal Feasibility

Compliance Requirements:

- Ensuring that the application complies with software development standards and best practices.
- Adhering to data privacy regulations, especially when handling sensitive documentation.
- Intellectual property considerations regarding the use of third-party libraries and tools.

Potential Legal Issues:

- Ensuring no violation of data protection laws (e.g., GDPR).
- Securing licenses for any third-party software used.

D.4. Operational Feasibility

User Requirements:

- Software developers, QA analysts, and project managers need a tool that is easy to integrate with their existing workflows.
- The application should provide intuitive interfaces and clear documentation.

Operational Challenges:

- Training users to effectively use the new system.
- Ensuring compatibility with existing software development tools and platforms.
- Providing ongoing support and updates.

D.5. Schedule Feasibility

Project Timeline:

- **Planning and Requirements Gathering:** 3 months
- **Design and Prototyping:** 2 month

- **Development and Integration:** 2 months
- **Testing and Quality Assurance:** 1 month
- **Deployment and Training:** 1 month
- **Total Duration:** 9 months

Appendix E: Code Documentation

E.1. Features Extraction Module

Features extraction module was implemented from scratch including 121 features implemented from scratch

```
# Latent Semantic Analysis.
def LSA(self,tfidf_matrix_uc: np.ndarray,tfidf_matrix_code: np.ndarray,train_or_test: str = 'train') -> np.ndarray:
    num_components = min(tfidf_matrix_uc.shape[0], tfidf_matrix_code.shape[0])
    num_components = min(num_components, 100)
    LSA_model = TruncatedSVD(n_components=num_components)

    if train_or_test == "train":
        LSA_data_useCases = LSA_model.fit_transform(tfidf_matrix_uc)
        LSA_data_codes = LSA_model.fit_transform(tfidf_matrix_code)
        LSA_similraity_matrix = cosine_similarity(LSA_data_useCases, LSA_data_codes)
        return LSA_similraity_matrix
    else:
        LSA_data_useCases = LSA_model.transform(tfidf_matrix_uc)
        LSA_data_codes = LSA_model.transform(tfidf_matrix_code)
        LSA_similraity_matrix = cosine_similarity(LSA_data_useCases, LSA_data_codes)
        return LSA_similraity_matrix
```

```
def LDA(self,UC_documents:list,code_documents:list,TotalTokens:set):


    id2word = corpora.Dictionary([list(TotalTokens)])
    corpus_uc = [id2word.doc2bow(doc.split()) for doc in UC_documents]
    corpus_code = [id2word.doc2bow(doc.split()) for doc in code_documents]

    num_topics = 200
    lda_model_uc = LdaMulticore(corpus=corpus_uc, id2word=id2word, num_topics=num_topics)
    lda_model_code = LdaMulticore(corpus=corpus_code, id2word=id2word, num_topics=num_topics)

    DocumentTopicDisUC = lda_model_uc[corpus_uc]
    DocumentTopicDisCode = lda_model_code[corpus_code]

    DocumentTopicDisUC_dense = gensim.matutils.corpus2dense(DocumentTopicDisUC, num_terms=num_topics).T
    DocumentTopicDisCode_dense = gensim.matutils.corpus2dense(DocumentTopicDisCode, num_terms=num_topics).T

    cosine_similarities = cosine_similarity(DocumentTopicDisUC_dense, DocumentTopicDisCode_dense)

    return DocumentTopicDisUC_dense, DocumentTopicDisCode_dense, cosine_similarities
```

```
def CountVectorizerModel(self, UC_documents: list, code_documents: list, train_or_test: str = "train"):

    self.code_count_matrix = np.zeros((len(code_documents), len(code_documents[0])))
    self.UC_count_matrix = np.zeros((len(UC_documents), len(UC_documents[0])))

    if train_or_test == "train":
        self.UC_count_matrix = self.count_vectorizer.fit_transform(UC_documents)
        self.code_count_matrix = self.count_vectorizer.fit_transform(code_documents)
    else:
        self.UC_count_matrix = self.count_vectorizer.transform(UC_documents)
        self.code_count_matrix = self.count_vectorizer.transform(code_documents)

    self.count_vocab_index = self.count_vectorizer.vocabulary_

    UC_total_number_of_occurences_in_corpus = np.sum(self.UC_count_matrix, axis=0)
    code_total_number_of_occurences_in_corpus = np.sum(self.code_count_matrix, axis=0)

    # dict of word: # of occurences in the corpus
    tf_uc_dict = {word: UC_total_number_of_occurences_in_corpus[0,i] for word,i in self.count_vocab_index.items()}
    tf_code_dict = {word: code_total_number_of_occurences_in_corpus[0,i] for word,i in self.count_vocab_index.items()}

    UC_words_count = self.UC_count_matrix.sum(axis=1)
    code_words_count = self.code_count_matrix.sum(axis=1)

    self.UC_count_matrix /= UC_words_count
    self.code_count_matrix /= code_words_count

    self.UC_count_matrix = self.UC_count_matrix.tolist()
    self.code_count_matrix = self.code_count_matrix.tolist()

    return self.UC_count_matrix, self.code_count_matrix,tf_uc_dict,tf_code_dict
```

E.2. Maintainability Score Module

```
def computeHalsteadVolume(self) -> tuple:
    curr_node = self.tree.root_node

    arethmatc_operators={'+','-','*','/','%'}
    assignment_operators={'=','+=', '-=', '*=', '/=', '%='}
    relational_operators={ '==', '!=', '<', '>', '<=', '>='}
    logical_operators={'&&', '||', '!',"instanceof"}
    ternary_operators={'?',':'}
    unary_operators={'+','-','++','--','!'}
    bitwise_operators={'~','<<','>>','>>>','^','&'}
    data_types={"byte","short","int","long","float","double","boolean","char","class","superclass","object","string","array",
    "interface"," super_interfaces"}
    loops_conditions={"for","while","return","if"," method_invocation"}
    operators_count=0  #N1

    unique_operators=set()
    queue = list()
    id = 0
    queue.append(curr_node)

    while(len(queue)):
        curr_node = queue.pop(0)
        if(curr_node.type in arethmatc_operators ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        elif (curr_node.type in assignment_operators ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        elif (curr_node.type in relational_operators ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        elif (curr_node.type in logical_operators ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        elif (curr_node.type in ternary_operators ):
            unique_operators.add(curr_node.type)
```

```
            operators_count+=1
        elif (curr_node.type in ternary_operators ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        elif (curr_node.type in unary_operators ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        elif (curr_node.type in bitwise_operators ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        elif (curr_node.type in data_types ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        elif (curr_node.type in loops_conditions ):
            unique_operators.add(curr_node.type)
            operators_count+=1
        queue.extend(curr_node.children)


    CycleComplexity = 1
    queue = list()
    curr_node = self.tree.root_node
    queue.append(curr_node)
    while(len(queue)):
        curr_node = queue.pop(0)
        if curr_node.type == "if_statement" or curr_node.type == "elif_statement" or curr_node.type == "for_statement" or
        curr_node.type == "while_statement" or curr_node.type == "except_clause" or curr_node.type == "with_statement" or
        curr_node.type == "assert_statement" or curr_node.type == "boolean_operator":
            CycleComplexity += 1
        queue.extend(curr_node.children)

    print("Cyclomatic Complexity: ", CycleComplexity)
```

```
unique_operators_count=len(unique_operators)  #n1
###print(unique_operators_count)

unique_operands =set()
operands_count=0
operators_count=0
unique_operators=set()
operators = {'+', '-', '*', '/', '%', '&&', '||', '!', '&', '|', '^', '~', '<<', '>>', '>>>', '==', '!=', '<', '>',
'<=', '>=', '=', '+=', '-=', '*=', '/=', '%=', '&=', '|=', '^=', '<<=', '>>=', '>>>=', '?'}


curr_node = self.tree.root_node

queue = list()
queue.append(curr_node)
while(len(queue)):
    curr_node = queue.pop(0)
    ###print(curr_node)
    if(curr_node.type in operators):
        unique_operators.add(curr_node.type)
        operators_count+=1
    queue.extend(curr_node.children)

###########################################      operators    ############################################
#access modfiers??, extends, implments ?? , class ?? , function calls , function definations ,
unique_operators=set()
queue = list()
curr_node = self.tree.root_node
queue.append(curr_node)
excluded_operators=["}","]",")","import", "package"]
while(len(queue)):
    curr_node = queue.pop(0)
    if (not curr_node.is_named and curr_node.type not in excluded_operators):
        unique_operators.add(curr_node.type)
```

```
            operators_count+=1

        if (curr_node.type == "method_invocation"):
            operators_count+=1
            method_name=self.source_code[curr_node.children[-2].start_byte:curr_node.children[-2].end_byte]
            unique_operators.add(method_name)

        if (curr_node.type == "type_identifier" and curr_node.parent.type!="type_list"  and  curr_node.parent.
        type!="superclass"):
            operators_count+=1
            class_name=self.source_code[curr_node.start_byte:curr_node.end_byte]
            unique_operators.add(class_name)
        if(curr_node.type == "void_type"):
            operators_count+=1
            unique_operators.add("void_type")

        if(curr_node.type != "import_declaration" and curr_node.type != "package_declaration"):
            queue.extend(curr_node.children)

    ###print(unique_operators)

    ############################################     operands    ############################################
    # package and import statements are not included in operands
    #
    unique_operands =set()
    queue = list()
    curr_node = self.tree.root_node
    queue.append(curr_node)
    while(len(queue)):
        curr_node = queue.pop(0)
        if (curr_node.type == "identifier" and curr_node.parent.type != "method_invocation"):
            var_name=self.source_code[curr_node.start_byte:curr_node.end_byte]
            unique_operands.add(var_name)
            operands_count+=1
        if (curr_node.type == "string_fragment"):
```

```
            unique_operands.add(curr_node.type)
            operands_count+=1
        if ("literal" in curr_node.type and curr_node.type != "string_literal" ):
            value=self.source_code[curr_node.start_byte:curr_node.end_byte]
            unique_operands.add(value)
            operands_count+=1
        if (curr_node.type == "true" and curr_node.type == "false"):
            unique_operands.add(curr_node.type)
            operands_count+=1
        if (curr_node.type == "type_identifier" and (curr_node.parent.type =="type_list"  or  curr_node.parent.
        type=="superclass")):
            class_name=self.source_code[curr_node.start_byte:curr_node.end_byte]
            unique_operands.add(class_name)
            operands_count+=1
        if(curr_node.type != "import_declaration" and curr_node.type != "package_declaration"):
            queue.extend(curr_node.children)

    ##print(unique_operands)

    unique_operators_count=len(unique_operators)  #n1
    unique_operands_count=len(unique_operands)  #n2

    # Program vocabulary: η=η1+η2
    # Program length: N=N1+N2
    # Calculated program length: Nˆ=η1log2η1+η2log2η2
    # Volume: V=Nlog2η
    # Difficulty: D=η12·N2η2
    # Effort: E=D·V
    # Time required to program: T=E18 seconds
    # Number of delivered bugs: B=V3000.

    n=unique_operands_count+unique_operators_count
    N=operators_count+operands_count
    N_hat=unique_operators_count*math.log2(unique_operators_count)+unique_operands_count*math.log2(unique_operands_count)
    V=N*math.log2(n)
```

```
D=(unique_operators_count/2)*(operands_count/unique_operands_count)
E=D*V
T=E/18
B=V/3000

return n, N, N_hat, V, D, E, T, B
```

```
def computeSLOCAndCommentLines(self) -> tuple:
    # calculate SLOC
    SLOC = 0 #number of source line codes without comments and spaces and imports
    flag = False
    LOC = 0
    blank_lines = 0
    with open(self.file_dir, "r") as file:
        for line in file:
            line = line.replace(" ", "")
            LOC += 1
            if not len(line):
                blank_lines += 1
            #lw fy awel el line multiline comment f=t
            if line.strip().rstrip('\n')[0:2] == '/*':
                flag = True

            #lw ft7en multiline comment bs aflto msh fy akhr el line yb2a f=F w hn3d sloc
            match = re.search(r"\*\/",line)
            if flag and match:
                flag=False
                if line[match.end():match.end()+2] == "//":
                    continue
                elif (line[match.end():match.end()+2])!="/*" and (line[match.end():match.end()+2])!="\n":
                    SLOC+=1
                    continue
            #lw fy akher el line aflt el multiline comment f=F, w mfesh iteration
            if line.strip().rstrip('\n')[-2:] == '*/':
                flag = False
                continue
            #lw ft7en multiline comment yb2a msh hn3d el sloc
            if flag:
                continue

            #lw 3dena kol el fo2a yb2a aked el flag msh btrue , w el comment msh fy awel el line
            #hncheck lw comment asln w w2tha hndwr 3la aflto , bs kda kda hyt7seb sloc
            if re.search(r"\/\*",line) != None and re.search(r"\/\*",line).start() != 0:

                if re.search(r"\*\/",line)!=None:
                    flag = False
                else:
                    flag = True

            if line.strip().rstrip('\n') != "" and not line.strip().startswith(('package', 'import','//')):
                SLOC += 1

    # print(SLOC)
    comment_lines = LOC - SLOC - blank_lines
    return SLOC, (comment_lines/LOC) * (math.pi/180)
```

```
def computeCyclomaticComplexity(self) -> int:
    ################################### Cyclomatic Complexity  ########################################

    # if    +1  An if statement is a single decision.
    # elif  +1  The elif statement adds another decision.
    # else  +0  The else statement does not cause a new decision. The decision is at the if.
    # for   +1  There is a decision at the start of the loop.
    # while +1  There is a decision at the while statement.
    # except    +1  Each except branch adds a new conditional path of execution.
    # finally   +0  The finally block is unconditionally executed.
    # with  +1  The with statement roughly corresponds to a try/except block (see PEP 343 for details).
    # assert    +1  The assert statement internally roughly equals a conditional statement.
    # Comprehension +1  A list/set/dict comprehension of generator expression is equivalent to a for loop.
    # Boolean Operator  +1  Every boolean operator (and, or) adds a decision point.
    curr_node = self.tree.root_node
    G = 1
    queue = list()
    queue.append(curr_node)
    types_to_check = {'if', 'else if', 'for', 'while', '&&', '||', '!', 'assert', 'catch'}
    while(len(queue)):
        curr_node = queue.pop(0)
        if(curr_node.type.lower() in types_to_check):
            G += 1
        queue.extend(curr_node.children)
    return G
```

```
def computeMaintainabilityScore(self) -> float:
    n, N, N_hat, V, D, E, T, B = self.computeHalsteadVolume()
    SLOC, comment_lines = self.computeSLOCAndCommentLines()
    G = self.computeCyclomaticComplexity()

    if not V:
        V = 1
    if not SLOC:
        SLOC = 1

    # print("V: ", V)
    # print("SLOC: ", SLOC)
    # print("G: ", G)

    MI= max(0, (100*(171 - 5.2*np.log(V) - 0.23*G - 16.2*np.log(SLOC))) / 171)
    return MI
```