Deterministic Classification of Accounting Functions in Code Contributions

MASTERS THESIS

Arni Islam

Submitted on 10 June 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Julian Hirsch

Declaration of Originality

Erlangen, 10 June 2025

License

Erlangen, 10 June 2025

Abstract

This thesis proposes a proof of concept for identifying Development, Enhancement, Maintenance, Protection, and Exploitation (DEMPE) functions through code contributions using machine learning techniques. Utilizing Design Science Research Methodology (DSRM), this study aims to create and assess a deterministic classification engine (artifact) for commit messages. Various supervised machine learning models: Logistic Regression, Random Forest, XGBoost, Neural Networks, and classifier chains were trained and systematically evaluated. The dataset, sourced from eight selected GitHub repositories, was manually labeled based on the DEMPE function definitions and conventional commit tag guidelines. After labeling, the data were cleaned and analyzed for class imbalance. To address this imbalance and improve model generalization, the Multi-label Synthetic Minority Oversampling Technique (MLSMOTE) was applied. Subsequently, Sentence-BERT (SBERT) using the all-MiniLM-L6-v2 model was employed to generate semantically meaningful vector representations of the commit messages. These embeddings capture the contextual meaning of sentences, enabling the models to learn from the underlying semantics rather than relying solely on surface-level text features of the text. We also developed a command-line interface (CLI) tool to reproduce the results. The tool supports data fetching from sources, data extraction, data preprocessing, model training, and real-time prediction, accommodating both conventional and non-conventional commit messages, thus providing a practical solution for classifying commits. Among the models tested, the Logistic Regression classifier using a One-vs-Rest strategy delivered the best performance, achieving an *average F1-score of 0.916* across DEMPE classes.

Contents

List of Figures

List of Tables

Acronyms

MNE	Multinational Enterprises
BEPS	Base Erosion and Profit Shifting
DEMPE	Development, Enhancement, Maintenance, Protection, and Exploitation
DSRM	Design Science Research Methodology
IS	InnerSource
ISSD	Inner Source Software Development
OECD	Organisation for Economic Co-operation and Development
NLP	Natural Language Processing
SemVer	Semantic Versioning
API	Application Programming Interface
GBM	Gradient Boosting Machine
RF	Random Forest
SVM	Support Vector Machine
ML	Machine Learning
IDE	Integrated Development Environments
MLSMOTE	Multi-label Synthetic Minority Oversampling Technique
SMOTE	Synthetic Minority Oversampling Technique
LR	Logistic Regression
CC	Classifier Chains
OvR	One-vs-Rest
OvA	One-vs-All

XGBoost	Extreme Gradient Boosting
NN	Neural Network
JSON	JavaScript Object Notation
CSV	Comma Separated Values
ReLU	Rectified Linear Unit
HDF5	Hierarchical Data Format version 5
CLI	Command Line Interface
MLP	Multilayer Perceptron
FNN	Feedforward Neural Network
GridSearchCV	Grid Search Cross-Validation
KNN	K-Nearest Neighbors
SBERT	Sentence-BERT
TF-IDF	Term Frequency-Inverse Document Frequency

1 Introduction

Modern developments in transfer pricing increasingly focus on intangibles, as they have become critical strategic assets for multinational enterprises (MNEs) (B. H. et al., 2021; O. T. et al., 2022; OECD, 2014). Intangible assets drive value creation, competitive advantages, and sustainable growth in the current interconnected global economy. Software products within inner-source communities are prime examples of such intangible assets. For tax purposes, these assets are governed by comprehensive principles and recommendations established by the Base Erosion and Profit Shifting (BEPS) Project (OECD, 2014). Modern organizations face challenges in connecting software development contributions to their strategic goals, leading to planning and evaluation inefficiencies. Collaborative development environments, in which contributions span teams and functions, exacerbate these challenges. The lack of a deterministic system to classify these contributions hinders effective resource management and performance analysis.

This research work focuses on Machine learning approach to classifying the code contributions of software in accounting functions, specifically using the DEMPE functions. They refer to the Development, Enhancement, Maintenance, Protection, and Exploitation of intangible assets within a multinational enterprise. These functions help determine which entity within the group is entitled to share in the returns derived from exploiting intangibles and which should bear the costs and investments associated with these assets (Exactera, 2021; RoyaltyRange, 2018).

We utilized the Design Science Research Methodology (DSRM) to achieve our research goals. The DSRM for information systems is a popular model for solving engineering problems. Since the model is employed to solve this research problem, it is briefly explained in Section 1.1.

Subsequently, based on the activities of DSRM for IS proposed by Peffers et al., 2007, this thesis is organized as follows: *Chapter 2 contains the identification of the problem. Chapter 3 focuses on defining the objectives of the solution, followed by Chapter 4, which contains the design and development of the proposed work. Chapter 5 demonstrates the use of the proposed solution in this study. Chapter 6* *evaluates the solution and finally, Chapter 7 concludes with key findings, limitations, and future work.*

The remainder of the introduction is organized as follows: Section 1.2 briefly explains the inner-source community. This is followed by the DEMPE framework guidelines in Section 1.3. Finally, Section 1.4 provides a detailed discussion on conventional commits.

1.1 Design science research methodology for information systems research

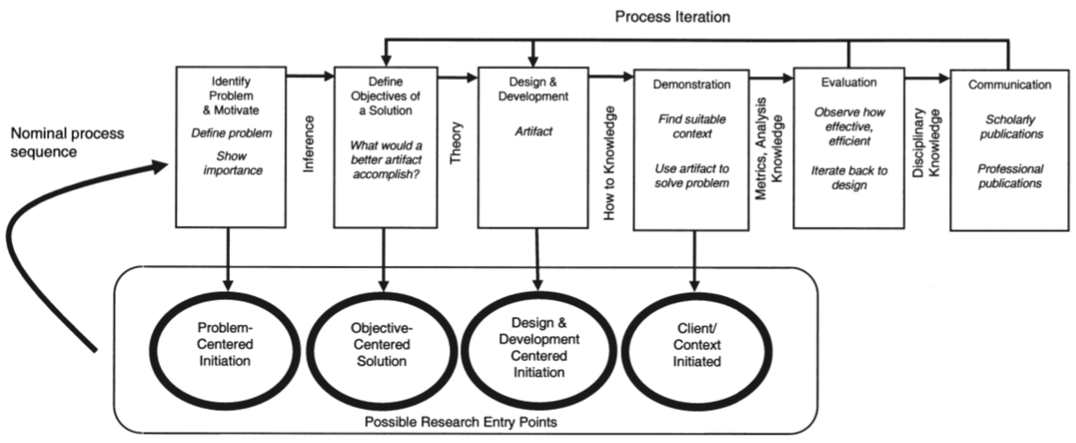

Process Iteration
Nominal process sequence
Identify Problem & Motivate
Define problem
Show importance
Inference
Define Objectives of a Solution
What would a better artifact accomplish?
Theory
Design & Development
Artifact
How to Knowledge
Demonstration
Find suitable context
Use artifact to solve problem
Metrics, Analysis Knowledge
Evaluation
Observe how effective, efficient
Iterate back to design
Disciplinary Knowledge
Communication
Scholarly publications
Professional publications
Problem-Centered Initiation
Objective-Centered Solution
Design & Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 1.1: Design Science Research Methodology (DSRM) process model (Peffers et al., 2007).

The Design Science Research Methodology (DSRM) for Information Systems (IS) focuses on creating and evaluating IT artifacts to address specific organizational challenges. This is a structured approach that combines rigorous research processes with practical problem-solving to generate meaningful contributions in the field. Artifacts developed in the DSRM can take various forms, including constructs, models, methods, or system instantiations (Hevner et al., 2004). Additionally, these artifacts may include social innovations or new properties of technical, social, and informational resources. Fundamentally, any designed object embedded with a solution to a research problem qualifies as an artifact.

The authors Peffers et al., 2007 structured the DSRM framework into six key activities, each essential for ensuring the development and evaluation of effective solutions, as outlined in Table 1.1.

Activity	Purpose
Problem Identification and Motivation	• Define the research problem and explain its significance. • Motivate stakeholders to engage with the solution. • Provide a clear rationale for understanding the problem.
Define the Objectives for a Solution	• Derive objectives from the problem and existing knowledge. • Include quantitative goals (e.g., measurable improvement). • Include qualitative goals (e.g., addressing unmet needs).
Design and Development	• Specify the artifact's architecture and features. • Develop the artifact according to these specifications.
Demonstration	• Show how the artifact solves instances of the problem. • Use simulations, case studies, or proof-of-concepts.
Evaluation	• Assess the artifact's performance against objectives. • Use quantitative metrics (e.g., speed, accuracy). • Collect qualitative feedback and empirical results.
Communication	• Used in scholarly publication. • Disseminate findings clearly and rigorously. • Present the problem, artifact, contributions, and evaluation.

Table 1.1: Overview of Design Science Research Methodology (DSRM) Activities (Peffers et al., 2007), highlighting each phase's purpose and the associated tasks for building and evaluating research artifacts.

We closely followed these phases. However, communication is for scholarly publications, so this specific step is not relevant to this thesis study at this time.

1.2 Inner-source methodology

An InnerSource community represents the adoption of open-source software development principles and practices within an organization. It enables multiple teams and departments to collaboratively develop and maintain shared software components or projects internally, as though they were part of a public open-source community (Capraro et al., 2018).InnerSource, also known as Inner Source Software Development (ISSD) (Edison et al., 2020). The core goals of InnerSource include promoting transparency, fostering collaboration, and encouraging the reuse of code across the organization.

In an IS community, the development process is guided by open-source like practices, Contributions from different teams are welcomed and reviewed according to the established guidelines (Van Lessen, 2019). Teams can leverage shared internal libraries and tools to avoid duplication of efforts and accelerate the development process (Commons, 2021). Code ownership is collective rather than tied to individual teams, fostering a culture of shared responsibility and cooperation among team members. Organizations of various sizes and sectors have embraced IS because of its adaptability and numerous advantages (Dorner, 2024). Since 2002, interest in InnerSource has grown steadily, as evidenced by the increasing number of academic publications, blog posts, and active community engagement (Capraro and Riehle, 2016).

A notable example of this growing interest IS Commons, an initiative founded in 2015 that boasts over 750 organizations and 3,000 individual contributors. Prominent companies such as IBM, Google, Microsoft, and SAP have adopted InnerSource to achieve benefits such as reduced development and maintenance costs, increased code reuse, enhanced knowledge sharing, and improved software quality (Wan et al., 2022). In addition to these technical and operational benefits, InnerSource has been linked to improved developer motivation, engagement, and retention, which contribute to project longevity (Constantino et al., 2021).

These advantages often mirror the benefits of traditional open-source software development, demonstrating InnerSource's potential to drive innovation, collaboration, and efficiency within organizations (Edison et al., 2020; Wan et al., 2022).

However, some organizations may be hesitant to adopt InnerSource because of concerns about the tax implications of transferring intellectual property (IP) across internal boundaries (Christel, 2024). The arm's length principle is a core method in transfer pricing that requires transactions between related entities to be conducted as though they were between independent parties, ensuring that costs and benefits are distributed fairly based on each entity's contribution (Dorner, 2024). Therefore, an accurate valuation of contributions is essential to maintain compliant and equitable financial practices, even for internally shared software assets.

1.3 The DEMPE Functions

For intangible assets, such as software development in Transfer Pricing, it is crucial to understand how value is created, managed, and utilized. The DEMPE framework: Development, Enhancement, Maintenance, Protection, and Exploitation, provides a structured approach to identify and classify these value-generating activities within multinational enterprises. By mapping contributions to these functions, organizations can gain deeper insights into how software-related efforts contribute to the overall value creation and distribution across the organization.

Each DEMPE function plays a distinct role: development involves creating or acquiring intangible assets, whereas enhancement focuses on improving the value or performance of these assets. Maintenance ensures continued functionality and usability over time, and protection safeguards intellectual property and ensures compliance with regulations. Exploitation, on the other hand, leverages these assets to generate economic value (RoyaltyRange, 2018).

These classifications also support the establishment of arm's-length pricing for intangibles by precisely defining the roles and contributions of various entities within the organization (OECD, 2014). A robust classification system that aligns InnerSource contributions with DEMPE functions can enhance the accuracy of the valuation and compliance with transfer pricing regulations. By categorizing contributions according to these functions, organizations can better understand the economic value of each contribution, enabling fair and compliant financial practices that reflect the true value of their internal efforts. Table 1.2 presents a structured overview of the DEMPE function classification.

Function	**Description**
Development	The development of intangibles refers to everything that is associated with coming up with ideas for the brand and products, and putting plans and strategies in place for their creation.
Enhancement	Involves continuing to work on aspects of intangibles to ensure they can perform well at all times and are constantly improved.
Maintenance	Maintaining intangibles involve everything that is possible to ensure they continue to perform well and generate revenue for a business.
Protection	Involves securing IP legal rights, ensuring that no one can copy the ideas, and monitoring competitor's activities.
Exploitation	The term 'exploitation' refers to how intangibles are used to generate profits

Table 1.2: Descriptions of the DEMPE functions, adapted from (RoyaltyRange, 2018)

1.4 Conventional Commits

In software development, commit classification is crucial for understanding the nature of code contributions and their implications for project management and for version control. However, there is no universally accepted method to classify commits, nor is there a consensus on the categories to which they belong (dos Santos and Figueiredo, 2020). Various approaches exist for commit classification, such as specifications that guide developers in systematically documenting their changes or leveraging Natural Language Processing (NLP) for automated categorization.

One popular specification is Conventional Commits, which defines a set of rules for creating an explicit and meaningful commit history. This specification categorizes commits into predefined types, such as fix, feat, BREAKING CHANGE, etc. These categories provide insights into the purpose of the changes and help streamline the development and review processes (Conventional Commits Community, 2019).

The Conventional Commits are based on Semantic Versioning (SemVer) 2.0.0 (Semantic Versioning Community, 2013), where versioning follows the format MAJOR.MINOR.PATCH. A MAJOR version indicates incompatible API changes, a MINOR version signifies new features added while maintaining backward compatibility, and a PATCH version represents backward-compatible bug fixes. Another similar specification, Semantic Commits, also categorizes commits based on their impact on the codebase and aims to create clear and structured commit histories.

A significant advantage of these specifications is their ability to automate release processes. For instance, Semantic Release integrates SemVer principles to automatically determine the next version number, generate release notes, and publish updates, relieving developers of the manual task of tracking categorized commits (InnerSource Commons Community, 2020). Table 1.3 presents a detailed summary of the commonly adopted keywords in the conventional commit specification, elucidating their semantic meanings and intended use cases in software development practices.

Keyword	**Purpose**
`feat`	Introduces a new feature to the codebase.
`fix`	Fixes a bug in the codebase, often linked to a PATCH-level release in semantic versioning.
`BREAKING CHANGE`	Introduces non-backward-compatible changes. Appears in the footer or with a `!` after the type/scope.
`build`	Modifies the build system or external dependencies.
`chore`	Routine changes that do not affect source or test files.
`ci`	Updates to continuous integration configurations or scripts.
`docs`	Documentation.
`style`	Code formatting or styling changes that do not affect functionality.
`refactor`	Code restructuring that improves clarity or maintainability without changing behavior.
`perf`	Performance improvements.
`test`	Adds or modifies test cases.

Table 1.3: Overview of conventional commit keywords and their purposes, based on the Conventional Commits v1.0.0 specification. These standardized keywords facilitate structured version control and semantic release automation.

2 Problem Identification

2.1 Problem Identification and Motivation

MNEs increasingly rely on intangible assets, such as software products developed within InnerSource communities, as strategic resources for competitive advantage and sustainable growth (O. T. et al., 2022). However, the collaborative and decentralized nature of InnerSource development presents significant challenges in attributing contributions to specific business functions and aligning software development activities with strategic objectives.

The absence of a deterministic system for classifying contributions makes it difficult to evaluate performance, allocate resources efficiently, and ensure compliance with transfer pricing regulations (OECD, 2014). The following section outlines the key challenges in applying the DEMPE classification to InnerSource environments.

2.2 Challenges and Importance of DEMPE Classification

While InnerSource fosters innovation and collaboration, it also introduces unique challenges related to management, resource allocation, and regulatory compliance, particularly when intangible assets such as software cross organizational boundaries without a structured classification system (Dorner, 2024).

A major issue is the difficulty in accurately attributing contributions and costs across departments or product lines. When multiple teams contribute to a shared codebase, it becomes unclear how to allocate the value of these contributions or their associated costs (InnerSource Commons Community, 2020). This ambiguity complicates resource allocation and hampers performance evaluation (Dorner, 2024). Without a clear structure, organizations struggle to align their contributions with their strategic goals, resulting in inefficiencies in project execution and resource use (Edison et al., 2020; Wan et al., 2022).

Another significant challenge is the tax and accounting implications of InnerSource practices. The transfer of intellectual property across internal organizational boundaries often raises concerns regarding compliance with transfer pricing regulations (OECD, 2014). For instance, MNE may hesitate to adopt InnerSource models because of fears of unintended tax liabilities or regulatory scrutiny (Dorner, 2024). The open nature of InnerSource can also lead to accounting difficulties when contributions cannot be traced to specific organizational entities. Without proper tracking, organizations may find it difficult to determine how to distribute costs or profits across business units (Capraro and Riehle, 2016; InnerSource Commons Community, 2020).

Management and oversight are also affected. In the absence of a classification system, it becomes difficult for leadership to direct development efforts efficiently or ensure alignment with long-term strategic objectives (Edison et al., 2020). High-frequency contributions make it even harder to distinguish between activities, such as adding new features or conducting routine maintenance. Without clear differentiation, efforts to improve performance may appear indistinguishable from general upkeep (Wan et al., 2022).

Classifying contributions using the DEMPE framework is essential for overcoming these obstacles. The DEMPE model offers a structured approach to categorizing contributions based on their role in value creation (OECD, 2014). It also aids in complying with transfer pricing laws by enabling organizations to demonstrate adherence to the arm's length principle, treating internal transactions as if they were between independent entities (Dorner, 2024). For example, if a software module developed in one region substantially enhances a global product, the DEMPE classification helps allocate profits and costs appropriately (Rudzika, 2018).

Moreover, the DEMPE categorization improves transparency, resource planning, and performance measurement. It enables organizations to identify areas requiring investment, such as "Protection" (such as legal rights), and maintain a balance between "Development" and "Maintenance" activities, in line with strategic goals. By distinguishing new contributions from improvements, organizations can better assess their impact, make informed decisions, and streamline communication (Edison et al., 2020; Wan et al., 2022).

3 Objective Definition

3.1 Defining Objectives for a Solution

The objective of this thesis is to develop a robust artifact that classifies code contributions to DEMPE functions. This artifact aims to provide a structured approach to categorizing code contributions within the context of transfer pricing, where practices intersect with the principles of DEMPE. We chose machine learning-based approach to achieve this objective.

As ML is an inductive learning system, it learns from examples and provides predictions for unseen data. In recent years, Commit Classification has become a growing area of interest in software engineering research, with numerous approaches developed to categorize code contributions. Many researchers have utilized various machine learning techniques and methodologies to classify commits.

Hindle et al., 2008 were among the first to explore machine learning methods for commit classification. They applied decision trees, Naive Bayes, SVM (Support Vector Machine), and nearest-neighbor algorithms to classify commits into maintenance categories.

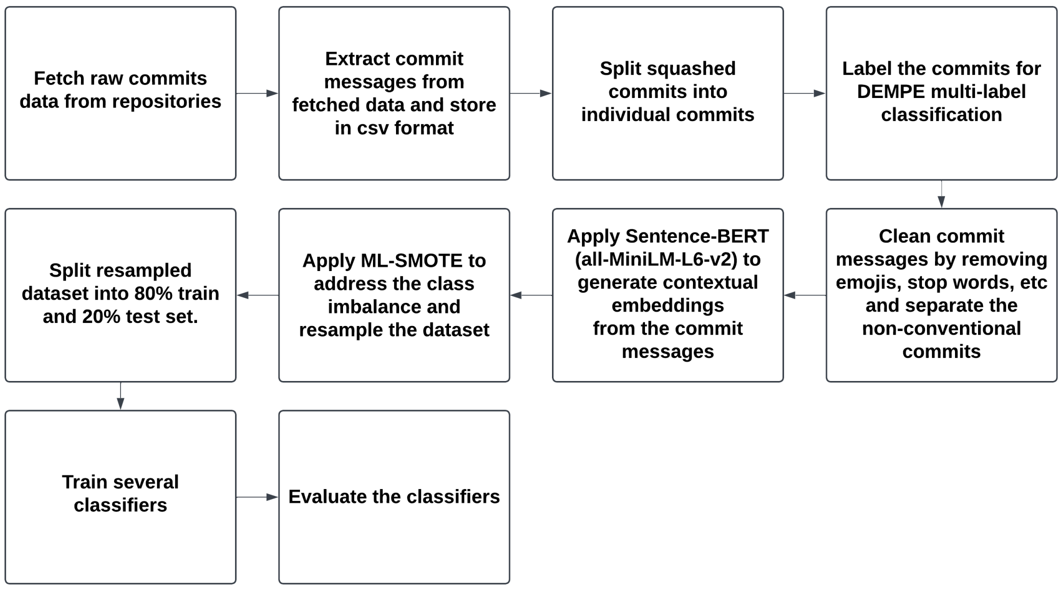
Levin and Yehudai, 2017 extended this work by utilizing word frequencies in commit messages and code changes as features for decision trees, Gradient Boosting Machines (GBM), and Random Forest (RF) algorithms. They focused on classifying commits into maintenance activities.

Sabetta and Bezzi, 2018 introduced a dual SVM approach, one for analyzing commit messages and another for code changes, to classify security-relevant commits.

Thus, classifying commit messages using ML approaches is a reliable and well-accepted method.

4 Design and development

4.1 Methodology



Fetch raw commits data from repositories
Extract commit messages from fetched data and store in csv format
Split squashed commits into individual commits
Label the commits for DEMPE multi-label classification
Clean commit messages by removing emojis, stop words, etc and separate the non-conventional commits
Apply Sentence-BERT (all-MiniLM-L6-v2) to generate contextual embeddings from the commit messages
Apply ML-SMOTE to address the class imbalance and resample the dataset
Split resampled dataset into 80% train and 20% test set.
Train several classifiers
Evaluate the classifiers

Figure 4.1: The step-by-step methodology used in this study is as follows: The process begins by fetching raw commit data from the selected GitHub repositories. The commit messages were extracted and stored in CSV format, followed by splitting the squashed commits into individual records. Subsequently, the commits were manually labeled, as shown in Table 4.3. The commit messages were then cleaned by removing noise (such as emojis and stop words), and non-conventional commits were separated. Sentence-BERT (all-MiniLM-L6-v2) was used to convert the commit messages into contextual vector embeddings. To address the class imbalance in the labeled data, MLSMOTE was applied to generate a balanced dataset. The data are then split into 80% training and 20% testing subsets, and several classification models are trained on them. Finally, the performance of the models is evaluated to determine their effectiveness.

4.2 Fetch raw commit data from repositories

To build a high-quality dataset for our study, we selected software projects from Github, one of the largest and most widely used platforms for open-source development. GitHub provides extensive repositories with rich metadata, making it an ideal source for collecting structured commit data.

To ensure that we collected data from well-maintained high-impact projects, we established a set of selection criteria that prioritized repositories with significant community involvement and a history of structured commit messages. The selection criteria are listed in Table 4.1.

Repository Selection Criteria

- Used conventional commit message format (at least most of the commits are conventional).
- Contained more than 300 commits.
- Had over 100 forks (indicating community interest).
- Actively maintained, with recent updates.
- Created before 2018-01-01 (indicating maturity and stability).

Table 4.1: Criteria used to select high-quality, community-driven repositories from GitHub for building the commit dataset. These filters ensured relevance, activity, and structured commit practices.

By applying these selection criteria, we identified and curated a set of eight repositories that are well-known in the open-source community. These repositories span various software domains. Each selected repository followed the conventional commit standard (most commits), allowing for a structured commit analysis. The final list of repositories used in this study is shown in Table 4.2. To fetch the commit data from these repositories, the CLI command 2 of our CLI tool must be executed (see Appendix A for more details).

Repository	**Description**	**Language(s)**	**Stars**	**Commits**
`yargs`	Command-line argument parser.	JavaScript	11.2k	1,842
`istanbuljs`	Tools for JavaScript test coverage.	JavaScript	1k	666
`standard-version`	Versioning and changelog management.	JavaScript	7.8k	335
`Blaze UI`	Framework-free modular UI toolkit.	JavaScript	1.6k	730
`electron`	Build cross-platform desktop apps.	JavaScript, HTML, CSS	7.8k	**29,273**
`uPortal-home`	Supplemental UI for Apereo uPortal.	JavaScript	25	2,793
`uPortal-app`	Framework for uPortal applications.	JavaScript	23	**3,084**
`scroll-utility`	Scroll utility for centering elements.	JavaScript	22	423

Table 4.2: Overview of GitHub repositories as of April 2025, selected for commit analysis, including a brief description, primary programming languages, popularity indicators (stars), and the total number of commits. These repositories were selected based on the criteria listed in Table 4.1. Notably, repositories such as `electron`, `uPortal-app`, and `uPortal-home` had significantly high commit volumes, making them valuable sources for analyzing contribution patterns and training classification models.

4.3 Justification for Sampling Subset of Commits

Although the selected repositories collectively contained approximately 39,146 commit messages, only a curated subset of 572 commits was used for model training and evaluation. Three key considerations primarily drove this decision.

- **Representativeness of the Subset:** The selected 572 commit messages were sampled from eight diverse open-source repositories in different domains. This subset captures a broad range of commit semantics and developer practices, making it a reasonable representation of typical commit behaviors. Prior research (Mitchell et al., 1997) has shown that carefully sampled smaller datasets (minimum 150 samples) can effectively reflect domain characteristics and support reliable model training, particularly for ML tasks.
- **Computational Constraints:** Multilabel classification tasks, particularly those involving ML models, are computationally intensive. Training on large-scale textual data would have significantly increased the training time and required greater computational resource requirements. By selecting a smaller, balanced subset, we ensured practical execution within reasonable resource limits while minimizing the risk of overfitting due to data redundancy.
- **Ethical and Reproducibility Considerations:** A smaller dataset facilitates transparent manual annotation and enhances the reproducibility of research. Sharing a manageable number of thoroughly reviewed samples enables other researchers to replicate or extend the study without being overwhelmed by its scale. This aligns with open science principles and supports the dissemination of clean, validated data.

4.4 Commit Extraction:

We retrieved raw commits from selected repositories and stored them in the directory `data/raw_data` during the data fetching stage. In the data extraction stage, we extracted commits from each repository's `.json` files and stored them in the `data/csv_data/raw_commit_messages.csv`. This step can be performed by executing the CLI command 3 (see appendix A for detail).

4.5 Splitting squashed commit messages and Labeling them

We noticed that many repositories contained squashed commits, where multiple commits were merged into a single commit message; therefore, we split these squashed commits into individual entries and treated each commit separately.

Most commit messages were written using conventional keywords and patterns associated with different software development activities. *Subsequently, based on the definitions of the DEMPE functions in Table 1.2 and the conventional commit keywords in Table 1.3, we mapped conventional commit tags to the corresponding DEMPE function, as shown in Table 4.3.* We then labeled the cleaned data for multi-label classification. The labeled commits are stored in `data/csv_data/labeled_commits.csv`. We can run the split squashed commit and label commit for both steps by executing the CLI command 4 (see Appendix A for details).

DEMPE Function	**Conventional Commit Tags**	**Class Label**
Development	`feat`	0
Enhancement	`BREAKING CHANGE`, `perf`	1
Maintenance	`fix`, `chore`, `docs`, `style`, `refactor`	2
Protection	`test`	3
Exploitation	`build`, `ci`	4

Table 4.3: Mapping of DEMPE functions to conventional commit tags and assigned class labels. This labeling bridges the semantics of commits with software functions to enable structured labeling for machine learning tasks.

Additionally, Figure 4.2 shows the co-occurrence of DEMPE class labeling, indicating that there is no overlap or ambiguity between classes based on the final labeling.

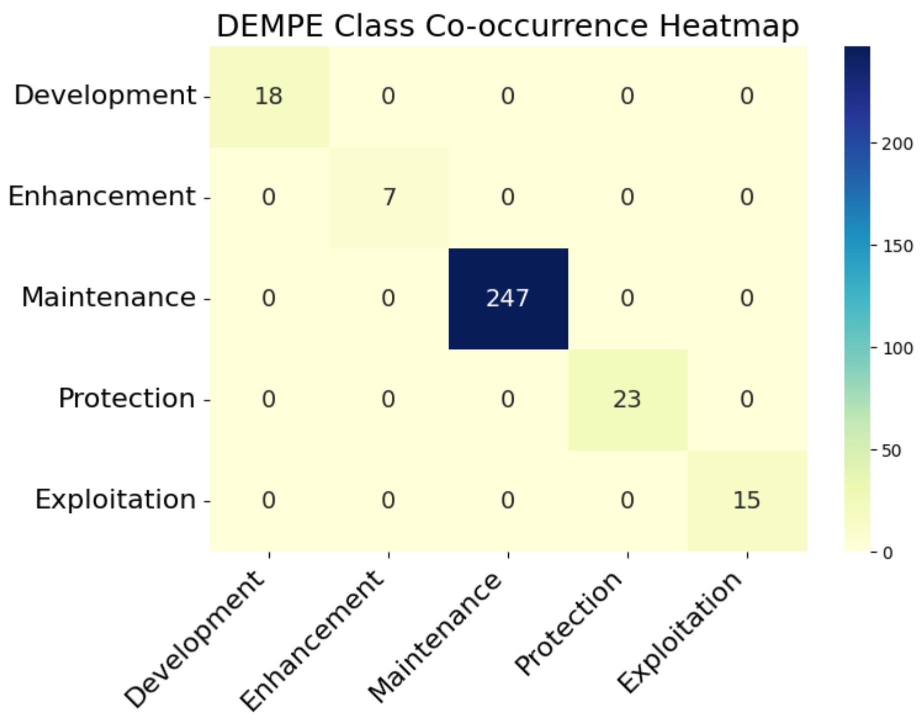


Figure 4.2: DEMPE class co-occurrence heatmap after labeling. The diagonal structure confirms that each commit message is uniquely labeled into one of the five DEMPE classes without any overlap. Class Maintenance dominates the dataset, followed by Protection, Development, Exploitation, and Enhancement.

4.6 Data Cleaning

We used a systematic text-cleaning procedure on the commit messages to ensure consistent and noise-free input for the model training. The following steps were performed sequentially:

Data Cleaning Steps

1. **Lowercasing:** Convert all characters to lowercase to eliminate case sensitivity, decrease vocabulary size, and improve generalization.
2. **Elimination of Commit Metadata:** Remove non-semantic elements from commit messages:
   - Commit hashes, such as `commit 4dae7a7,...`
   - Authors and email addresses
   - Date stamps
   - Co-authorship and signed-off-by attributions
   - Long lines of reasoning or reference lines
   - URLs
   - Numerical characters and punctuation
   - Whitespace normalization

This preprocessing ensured that the final text representation focused solely on the core semantic content of each commit message, thereby improving the quality of the input for classification. We can run this step by executing the CLI command 5 (see Appendix A for details).

4.7 Data Analysis

Following the commit data cleaning process, 310 adhered to the conventional commit format, and 263 were classified as non-conventional. For model training and evaluation, only conventional commits were utilized, as they provided structured and semantically rich information. Non-conventional commits were retained separately for potential manual inspection and comparative evaluations.

The percentage of cleaned conventional commit data distribution is shown in Figure 4.3. The figures clearly show that the majority of commit messages from the chosen repositories belong to the maintenance class. This is almost 80% in number.

The distribution is acceptable and is supported by many studies. Susan A. Sherer (1991), in her chapter "Software Maintenance" from the Software Failure Risk book, states that software maintenance consumes *60–80%* of most companies' software budgets, highlighting its significant impact on software costs (Sherer, 1991).

Islam and Katiyar, 2014, in their research on developing a software maintenance cost estimation model, they report that about 90% of software life cost is related to its maintenance phase, underscoring the growing significance of maintenance in the software lifecycle.

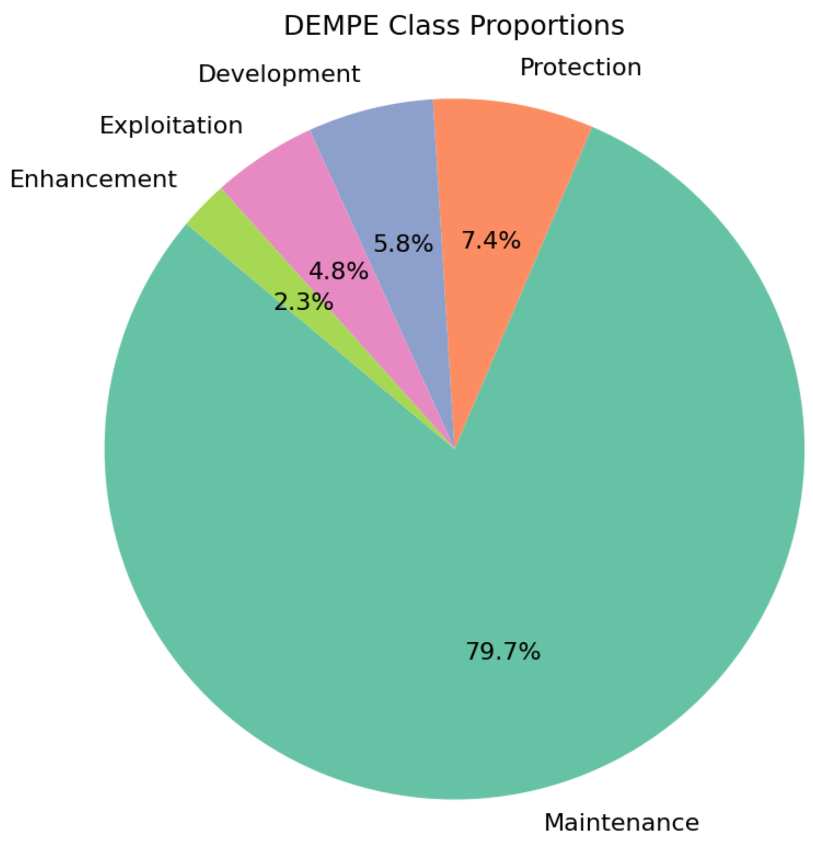


Figure 4.3: Pie chart displaying each class's distribution within the cleaned conventional commit dataset. *Maintenance* constitutes the majority with 79.7% of the total data, followed by *Enhancement* with 2.3%, *Development* with 4.8%, *Protection* with 7.4%, and *Exploitation* with 4.8%.

The figures 4.4, 4.5, 4.6, 4.7, and 4.8 illustrate the most frequently occurring terms in commit messages for each DEMPE functions. In the Development class, the keyword *feat* was dominant, which aligned with the expectations for commits that introduce new features or modules. High-frequency terms like *breaking change* and *perf* appear consistently in the Enhancement class, signifying standard performance improvement and code optimization efforts, respectively.

In the context of maintenance, the most frequently used terms are *chore*, *fix*, *refactor*, and *docs*, which stand for routine bug fixes, configuration modifications, and documentation updates, respectively. The Protection class is notably characterized by words such as *test*, which often appear in commits addressing security patches, access control, or compliance-related changes. Lastly, the most common word for exploitation was *build*, which indicates commits that contribute to packaging, releasing, and integrating software into production environments.

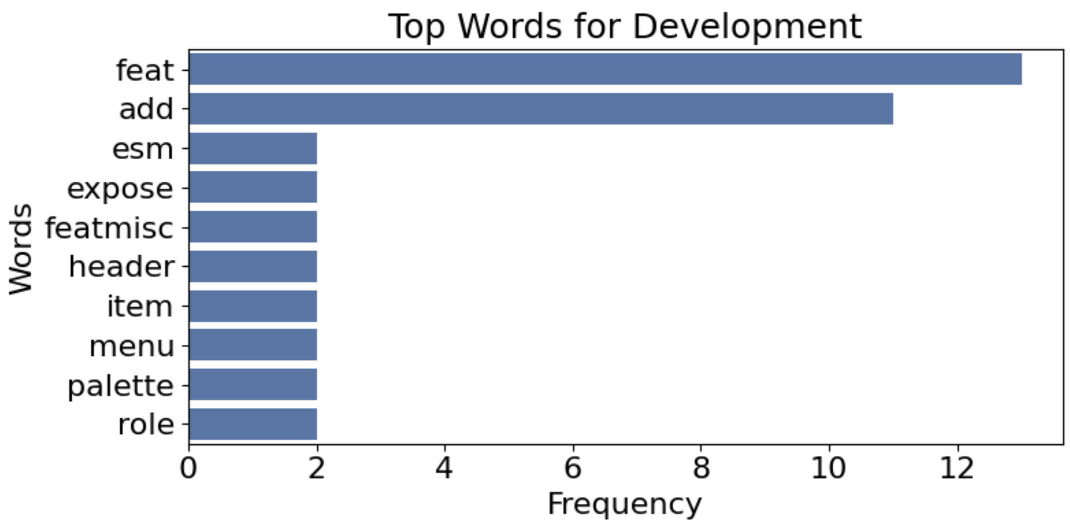


Figure 4.4: This bar chart illustrates the top 10 most frequently occurring words found in commit messages labeled under the Development function. The terms "add" and "feat" appear most often, indicating a strong association between the development class and actions related to adding new features and components. Other words, such as "esm", "menu", and "palette", reflect specific modules or UI-related additions. This frequency analysis helps identify common linguistic patterns in commits that are labeled as development. This gives us an idea of the semantic traits that determine model classification for this class.

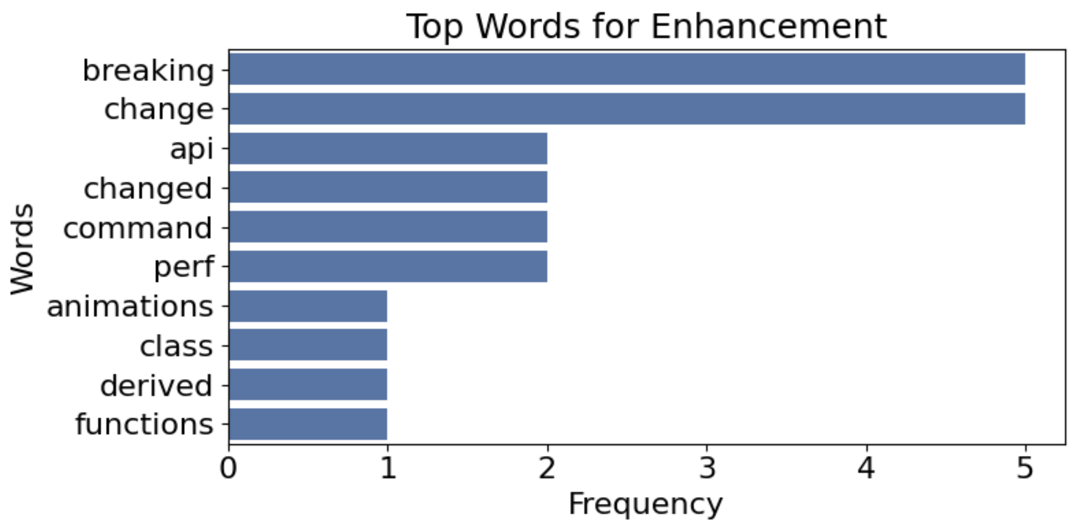


Figure 4.5: This bar chart shows the top 10 most frequent words found in commit messages categorized under the Enhancement function. The dominance of terms such as "breaking" and "change" indicates significant modifications or improvements made to existing codebases, often signaling backward-incompatible updates. Other common words, such as "perf", "api", and "changed", suggest performance optimization and interface adjustment. This frequency distribution helps illustrate the typical vocabulary associated with enhancement activities, reinforcing the semantic patterns leveraged during classification of this DEMPE category.

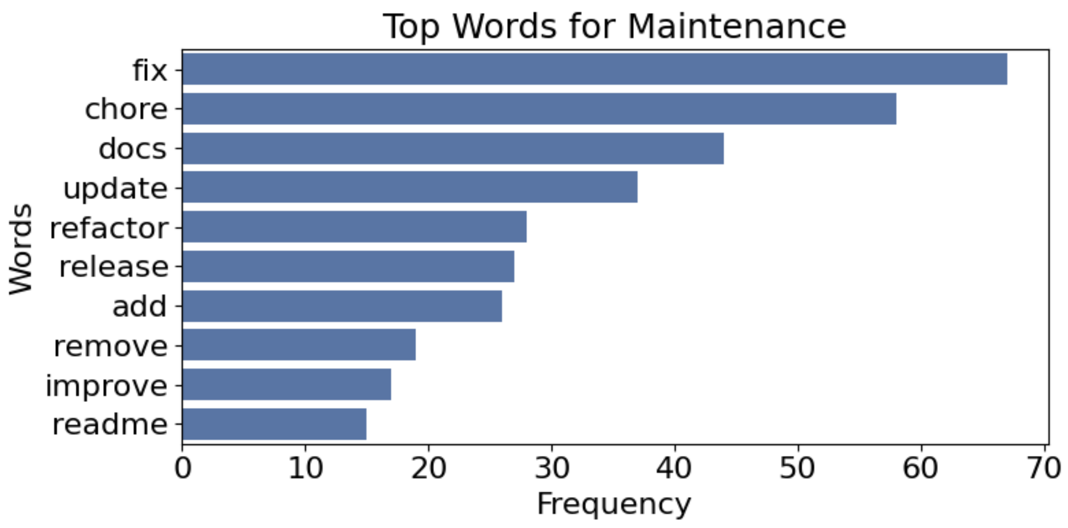


Figure 4.6: This chart visualizes the top 10 words that often show up in commit messages classified under the Maintenance function. High-frequency terms like "refactor", "fix", and "chore" emphasize routine codebase upkeep, bug fixes, and structural improvements without altering external functionality. The presence of "docs", "update", and "release" indicates attention to documentation, versioning, and software lifecycle tasks. These lexical patterns reflect common maintenance practices essential for preserving software stability, making them critical signals for automated classification in the DEMPE framework.

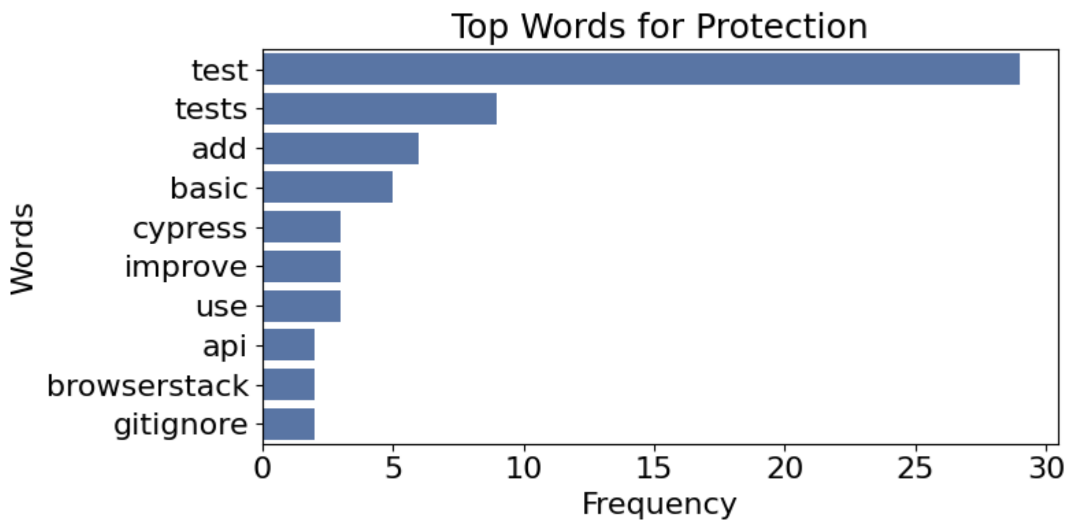


Figure 4.7: This bar chart presents the top 10 most frequently used words in commit messages classified under the Protection category. The dominant occurrence of terms such as "test" and "tests" reflects a strong focus on quality assurance and system validation. Other words, such as "cypress", "browserstack", and "gitignore" indicate efforts related to automated testing frameworks, cross-browser testing tools, and version control hygiene. These keywords collectively highlight the role of testing and configuration safeguards in preserving software integrity and enforcing intellectual property protection, core objectives of the Protection function in the DEMPE framework.

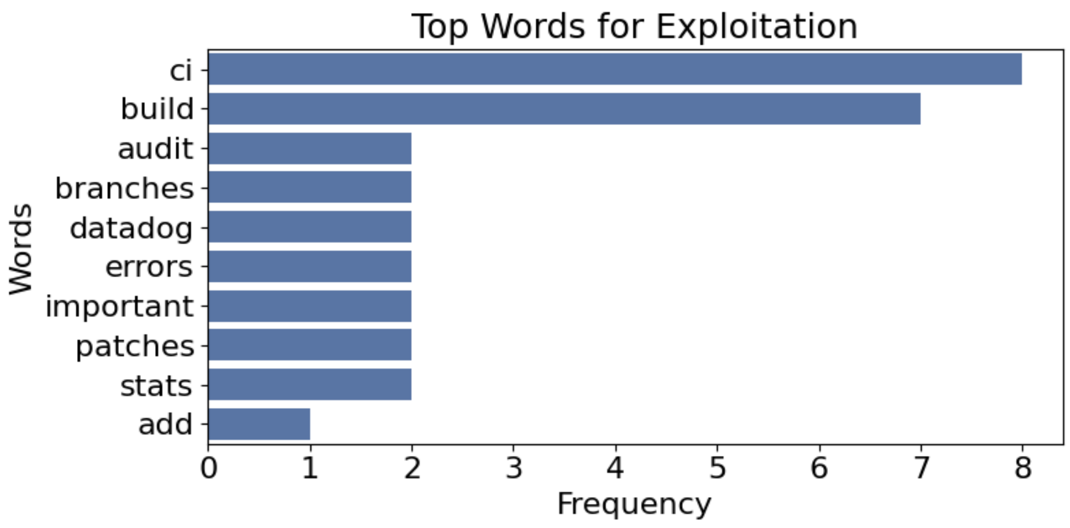


Figure 4.8: This figure illustrates the top 10 most frequently occurring words in commit messages categorized under the Exploitation class. Dominant keywords like "build", "bump", and "ci" indicate frequent activities related to build automation, version upgrades, and continuous integration pipelines. Words such as "builddepsdev", "output", and "matcher" further reflect the operational focus of this class, highlighting tasks essential for deploying and executing software in production environments. These terms align with the Exploitation function in the DEMPE framework, which emphasizes leveraging software artifacts to deliver business value, such as through deployment, monetization, or internal integration.

To generate the visualization plots, we executed the command 6 (see Appendix A for more details).

4.8 Sentence BERT (SBERT) encoding and applying MLSMOTE

To enhance the semantic understanding of commit messages and address the class imbalance in the dataset, two key preprocessing steps were applied: SBERT-based sentence embeddings and MLSMOTE-based data resampling.

4.8.1 Sentence Embedding using SBERT

Traditional text vectorization techniques, such as TF-IDF or Bag-of-Words, often fail to capture contextual meaning, particularly in brief texts such as commit messages. To overcome this limitation, we used Sentence-BERT (SBERT), a modification of the BERT architecture specifically designed for generating semantically meaningful sentence embeddings.

In this study, we used the pre-trained `all-MiniLM-L6-v2` model from the SBERT library (Reimers and Gurevych, 2019). This model is lightweight and efficient while still achieving strong performance on sentence-level semantic similarity tasks. Each cleaned commit message was passed through the SBERT to produce a vector representation. These embeddings capture both syntactic and semantic nuances, making them well-suited for downstream classification.

4.8.2 Handling Class Imbalance with MLSMOTE

To address class imbalance and enhance the classifier's generalization abilities, we applied the Multi-Label Synthetic Minority Oversampling Technique (MLSMOTE)(Charte et al., 2015).

Given a multi-label dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, where $x_i \in \mathbb{R}^d$ denotes the input feature vector and $y_i \in \{0,1\}^L$ denotes the label vector for L labels, The following steps are carried out by the MLSMOTE algorithm::

1. **Identify Minority Samples:** Label frequency thresholds are used to identify instances that belong to rare labelsets.
2. **Nearest Neighbors Search:** For each minority instance, k nearest neighbors are found using Euclidean distance in the feature space.
3. **Synthetic Sample Generation:** By interpolating between a minority instance and one of its neighbors, new synthetic feature vectors are produced. To preserve label dependencies, the corresponding label vector is calculated using label union or weighted averaging techniques.

This process continues until a balanced label distribution is achieved for each class. In our experiments, we set the number of nearest neighbors $k = 5$, following recommendations from the prior literature (Charte et al., 2015).

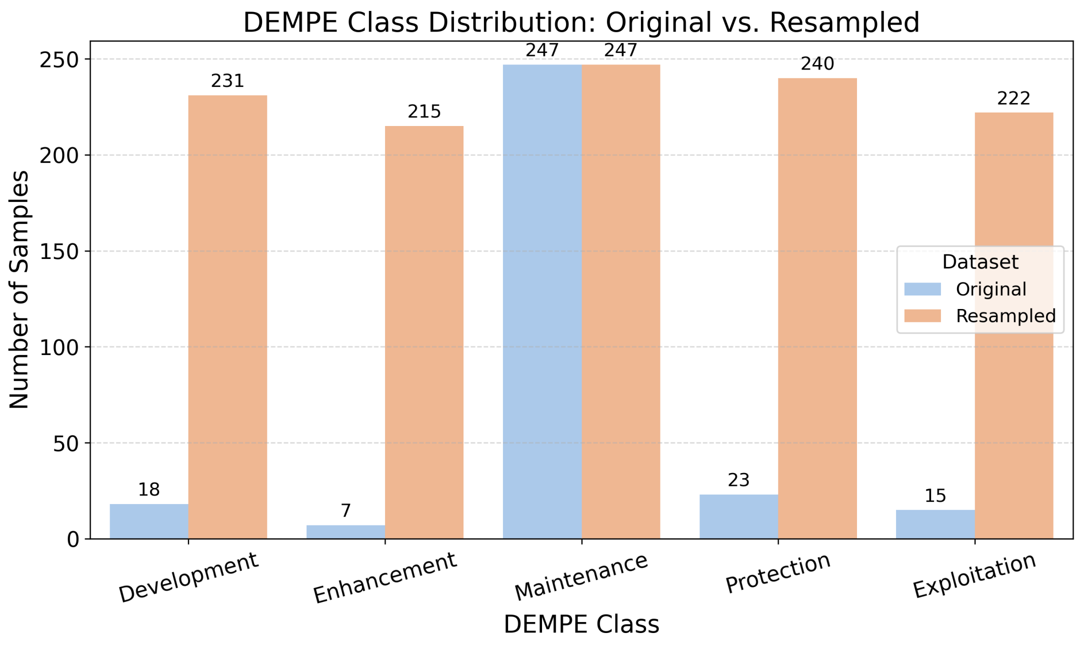


Figure 4.9: Comparison of class distribution between the original and resampled datasets for DEMPE labels. The original dataset exhibited severe class imbalance, particularly for classes such as **Enhancement** (7 samples), **Development** (18 samples), and **Exploitation** (15 samples), whereas the **Maintenance** class was dominant with 247 samples. To mitigate this imbalance and improve the classification performance, the MLSMOTE technique was applied, resulting in a balanced dataset in which each class contained approximately the same number of samples. This resampling ensures fair representation across classes and prevents the classifier from being biased toward the majority class during training.

4.9 Train Test Split

The dataset was then split into 20% test set and 80% training set, as it is a common ratio used in the machine learning domain. To perform the split, we ran the command 8.

4.10 Machine learning models

In this study, we evaluated a range of supervised classification models varying in complexity, including *Logistic Regression, Random Forest, XGBoost, Feedforward Neural Networks, and Classifier Chains.* Our approach to model selection was intentionally progressive, moving from simple to complex models to achieve a balance between interpretability and predictive performance.

We began with simpler models, such as Logistic Regression, because of their transparency and ease of interpretation. These models do not behave as black-box systems and provide clear insights into how features contribute to predictions. As we progressed to more complex models, such as XGBoost and Neural Networks, we gained increased representational capacity and potentially better performance, although at the cost of reduced interpretability. In other words, we followed the explainable Machine Learning (XML) approach (Roscher et al., 2020).

An essential characteristic of our task is that it involves multi-label classification: a single commit message may reflect multiple DEMPE business functions simultaneously (such as Development and Maintenance). This requires a classification strategy capable of assigning multiple labels to a single input, rather than forcing a mutually exclusive choice.

To support this requirement, we employ modeling techniques that natively support or are adapted to multilabel learning, such as *One-vs-Rest strategies and Classifier Chains.*

The remainder of this section is organized as follows:

We begin by introducing the *One-vs-Rest* (OvR) classification strategy in Section 4.10.1, and Section 4.10.2 explains the *Logistic Regression (OvR)* model, detailing its configuration and training strategy. Section 4.10.3 discusses the *Random Forest (One-vs-Rest)* classifier. Section 4.10.4 covers the use of *XGBoost*, a gradient-boosting technique known for its high performance with structured data. Section 4.10.5 presents the design and training of the *Feed forward Neural Network* model using Keras Tuner for hyperparameter optimization. Section 4.10.6 introduces the *Classifier Chain* approach and illustrates how label dependencies were leveraged for improved prediction accuracy.

4.10.1 One-vs-Rest (OvR) Classification Strategy

One-vs-Rest (OvR), also known as One-vs-All (OvA), is a popular strategy for handling classification problems in which an instance can belong to one or more classes, such as in multi-label classifications.

In the OvR approach, for a classification task with N classes, N independent binary classifiers are trained. Each classifier is responsible for distinguishing whether a given sample belongs to its respective class.

In the context of our DEMPE classification problem (where each commit message may belong to multiple classes), the OvR strategy is naturally suitable. This allows each class to be predicted independently using a binary classifier trained for that specific label. Figure 4.10 illustrates the OvR decision strategy.

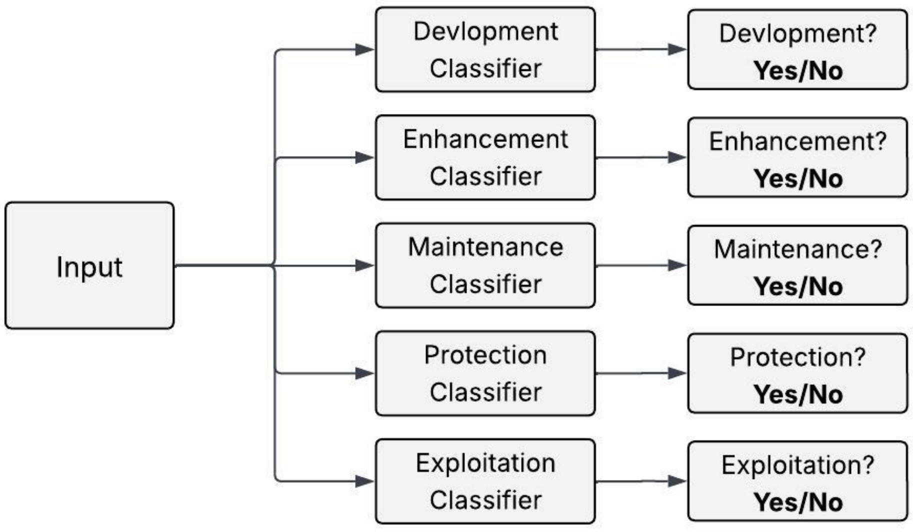


Figure 4.10: Visualization of the One-vs-Rest (OvR) classification approach used for DEMPE multi-label classification. In this setup, a single input (e.g., a commit message) is passed to five independent binary classifiers, each specialized in detecting one specific DEMPE function: Development, Enhancement, Maintenance, Protection, or Exploitation. Each classifier outputs a binary decision (`Yes/No`) indicating the presence or absence of its respective class. Because the classifications are independent, a commit can be assigned to multiple classes simultaneously. This structure makes OvR especially suitable for multi-label problems like the DEMPE framework, where overlapping functional roles are common.

4.10.2 Logistic Regression (One-vs-Rest)

Logistic Regression is a widely used linear classification algorithm that models the probability of a binary outcome using the logistic (sigmoid) function. In the context of multi-label classification, we employed the **One-vs-Rest (OvR)** strategy, in which a separate binary classifier was trained for each DEMPE class. This means that five independent logistic regression classifiers were trained, one for each of the following categories: Development, Enhancement, Maintenance, Protection, and Exploitation.

For this study, we used `scikit-learn`'s `OneVsRestClassifier` wrapper with a base estimator of `LogisticRegression`. Each classifier was trained to distinguish whether a commit belonged to the corresponding DEMPE class. The logistic regression model was configured using the `liblinear` solver, which is well suited for small datasets and supports both L1 and L2 regularization.

We implemented a **hyperparameter tuning** process using `GridSearchCV` with

3-fold cross-validation. The following hyperparameters were evaluated:

- Regularization strength: $C \in \{0.01, 0.1, 1, 10\}$
- Penalty type: L1 and L2 regularization

The best model was selected using the `f1_micro` scoring metric, resulting in:

- **Best regularization strength:** $C = 1$
- **Best penalty type:** L2 regularization
- **Best cross-validated F1 score:** 0.907

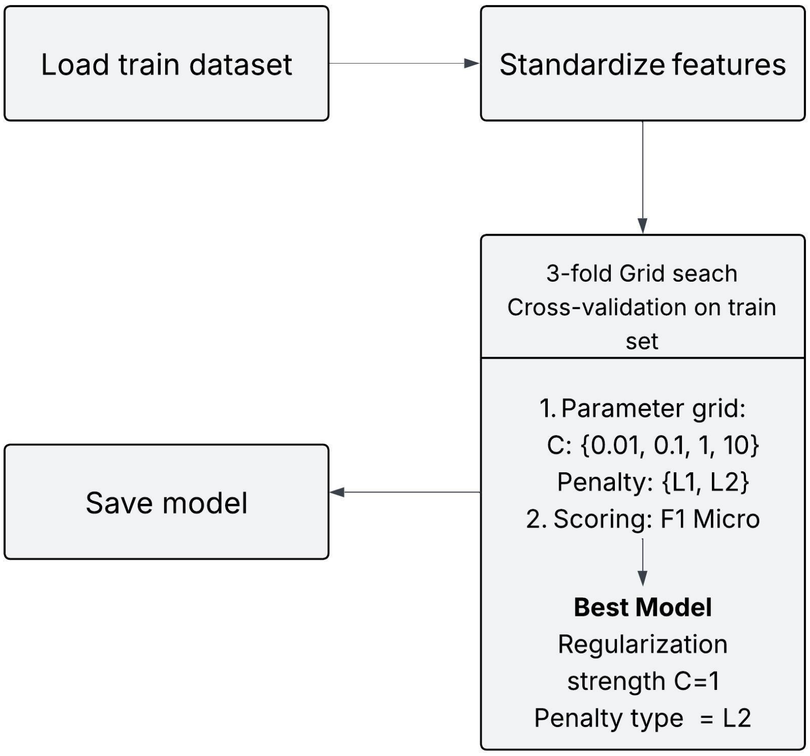


Figure 4.11: Visual representation of the training pipeline for the Logistic Regression (One-vs-Rest) model. The workflow begins by loading the training dataset and applying feature standardization. A 3-fold cross-validation was performed using GridSearchCV to optimize the hyperparameters, namely, the regularization strength (C) and penalty type (L1, L2). The best-performing model, identified using F1-micro scoring, is finally saved for evaluation and inference.

Once the optimal hyperparameters were identified, the final model was trained on the complete training dataset and evaluated using a held-out test set. The predictions are analyzed in Chapter 6. The trained model and parameters were serialized using `joblib` and saved to the disk for reproducibility and future use.

4.10.3 Random Forest (One-vs-Rest)

Random Forest is an ensemble learning method that constructs a multitude of decision trees during training and outputs the class that is the mode of the classes predicted by individual trees. It is robust against overfitting and performs well even in high-dimensional spaces, making it suitable for classifying sentence-encoded commit messages.

For this study, we used a **One-vs-Rest (OvR)** strategy to adapt the Random Forest for multi-label classification. Each DEMPE class was treated as a separate binary classification task, resulting in five parallel Random Forest classifiers. Feature standardization was performed using `StandardScaler`.

We implemented a hyperparameter tuning pipeline using `GridSearchCV` with 3-fold cross-validation to select the best-performing model configuration. The grid search explored the following hyperparameter space:

- Number of estimators (trees): `n_estimators` $\in \{100, 200\}$
- Maximum tree depth: `max_depth` $\in \{\text{None}, 10, 20\}$
- Minimum number of samples required to split an internal node: `min_samples_split` $\in \{2, 5\}$

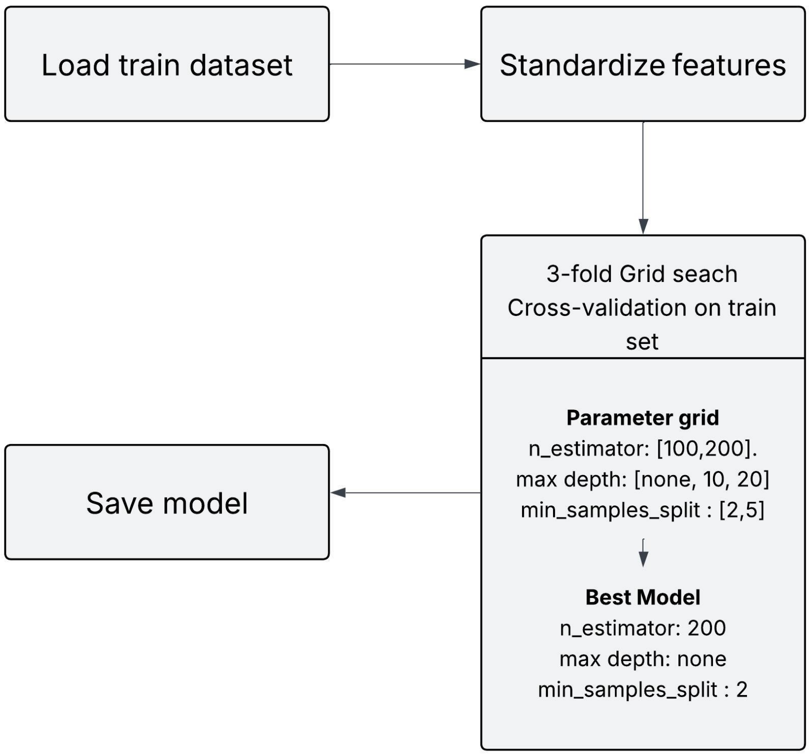


Figure 4.12: Visual representation of the training pipeline for the Random Forest (One-vs-Rest) model. The process begins with loading the training dataset and standardizing its features. A 3-fold cross-validation was conducted using GridSearchCV to explore combinations of hyperparameters, including the number of trees (`n_estimators`), maximum tree depth (`max_depth`), and minimum samples required to split a node (`min_samples_split`). The best-performing configuration was selected based on the F1-micro score. Finally, the optimized model is serialized and saved for downstream evaluation and prediction.

The model was evaluated using the `f1_micro` scoring metric, which is appropriate for multi-label classification tasks in which label imbalance may occur. The training process used the `RandomForestClassifier` from `scikit-learn`, wrapped in a `OneVsRestClassifier` to support the independent label prediction.

Best Configuration: The optimal model identified by the grid search had the following configuration:

- `n_estimators`: 200
- `max_depth`: None (nodes are expanded until all leaves are pure)
- `min_samples_split`: 2

This configuration yielded a cross-validated micro F1-score of `0.9099`. The final trained model was serialized using `joblib`, and detailed performance metrics were generated and discussed in Chapter 6.

4.10.4 XGBoost (One-vs-Rest)

XGBoost (Extreme Gradient Boosting) is a highly optimized implementation of the gradient boosting framework that excels in handling structured data and offers efficient regularization and scalability. For this study, we adapted XGBoost for multi-label classification using the **One-vs-Rest (OvR)** strategy, where a separate binary classifier was trained for each DEMPE label independently.

We used the `XGBClassifier` from the `xgboost` library as the base estimator inside `scikit-learn`'s `OneVsRestClassifier` wrapper. To optimize the model, we applied a 3-fold `GridSearchCV` using the following hyperparameter grid:

- Number of estimators: $n\_estimators \in \{100, 200\}$
- Maximum depth of trees: $max\_depth \in \{4, 8, -1\}$
- Learning rate: $learning\_rate \in \{0.05, 0.1\}$

All input features were standardized using `StandardScaler` before model training. The evaluation metric for model selection was `F1-micro`, which is suitable for imbalanced multilabel settings because it aggregates the contributions from all classes.

After the grid search, the best configuration was:

- $n\_estimators = 200$
- $max\_depth = 4$
- $learning\_rate = 0.1$

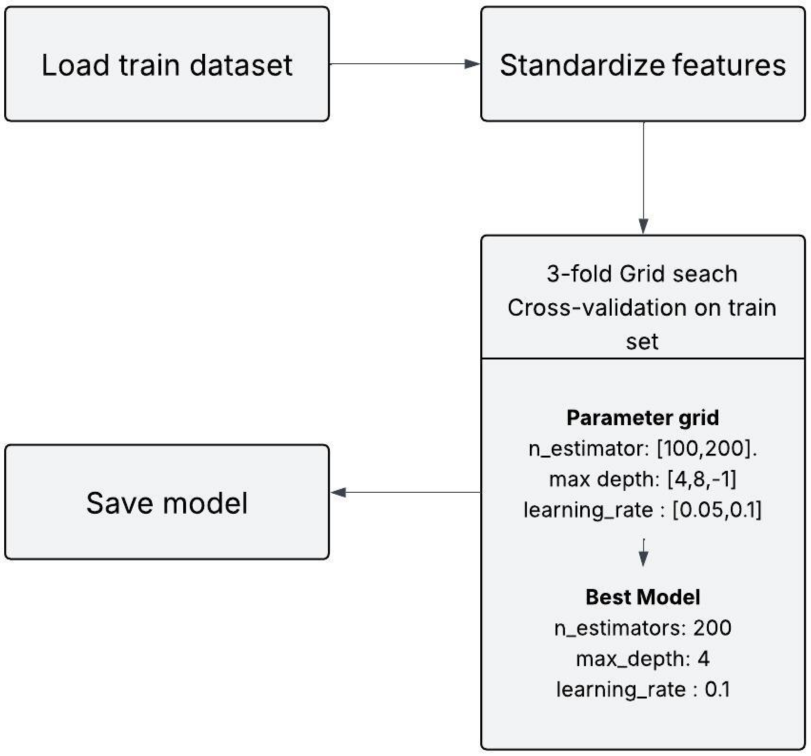


Figure 4.13: Training pipeline for the XGBoost (One-vs-Rest) model used in DEMPE classification. The process begins with loading and standardizing the training dataset, followed by a hyperparameter optimization phase using 3-fold cross-validation with GridSearchCV. The grid search explores combinations of key hyperparameters, the number of estimators (`n_estimators`), tree depth (`max_depth`), and learning rate. The best model configuration is selected based on F1-micro scoring and subsequently saved for evaluation and inference.

This configuration achieved the best cross-validated F1-micro score (0.904). The final model was persisted using `joblib` and used for evaluation on the test set. The prediction results and confusion matrices are further analyzed in Chapter 6.

4.10.5 Neural Network (Feedforward MLP)

To evaluate the applicability of deep learning for DEMPE classification, we implemented a feedforward Neural Network (Multilayer Perceptron, MLP) capable of handling multi-label outputs. Neural Networks are highly flexible models that can learn complex patterns in data; however, their performance and generalization capability depend heavily on proper architectural and hyperparameter choices. To ensure an optimal configuration, we employed the Keras Tuner library to automate hyperparameter search.

Model Architecture. The input to the network consisted of Sentence-BERT feature embeddings, which were standardized to improve convergence. The output layer contains five sigmoid-activated neurons corresponding to the five DE-MPE functions, enabling independent binary classification of each class. Binary cross-entropy was used as the loss function.

Tuning Strategy. We used `RandomSearch` from the Keras Tuner library with the following hyperparameter ranges:

- Number of units in the first hidden layer: 128, 192, 256, 320, 384, 448, 512
- Dropout rate after first hidden layer: 0.2, 0.3, 0.4, 0.5
- Number of units in the second hidden layer: 64, 128, 192, 256
- Dropout rate after second hidden layer: 0.2, 0.3, 0.4, 0.5
- Learning rate (Adam optimizer): 0.001, 0.0005

Best Model Configuration. After 10 trials with early stopping based on validation loss (patience=10), the best model had the following configuration:

- Hidden Layer 1: 256 units, ReLU activation
- Dropout after Layer 1: 0.4
- Hidden Layer 2: 128 units, ReLU activation
- Dropout after Layer 2: 0.2
- Optimizer: Adam with learning rate 0.0005

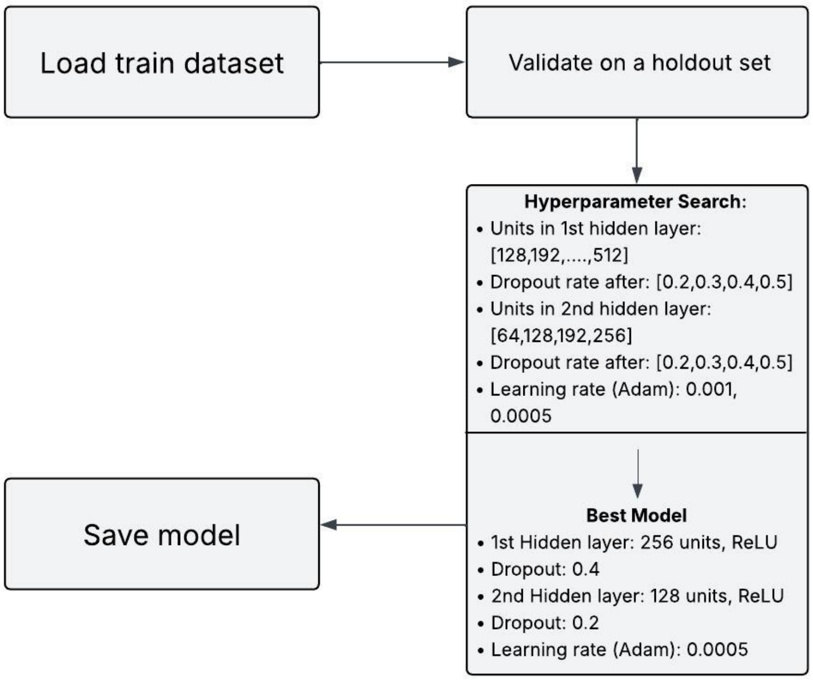

Load train dataset
Validate on a holdout set
Hyperparameter Search:
• Units in 1st hidden layer: [128,192,....,512]
• Dropout rate after: [0.2,0.3,0.4,0.5]
• Units in 2nd hidden layer: [64,128,192,256]
• Dropout rate after: [0.2,0.3,0.4,0.5]
• Learning rate (Adam): 0.001, 0.0005
Best Model
• 1st Hidden layer: 256 units, ReLU
• Dropout: 0.4
• 2nd Hidden layer: 128 units, ReLU
• Dropout: 0.2
• Learning rate (Adam): 0.0005
Save model

Figure 4.14: Training pipeline for the Feedforward Neural Network used in DEMPE classification. The workflow starts with loading the training dataset, followed by hyperparameter optimization using `Keras Tuner`'s random-search strategy. The tuning process explored combinations of hidden layer sizes, dropout rates, and learning rates, using a holdout validation set. The best-performing configuration, based on validation accuracy, was selected, comprising two hidden layers with ReLU activation, dropout regularization, and an Adam optimizer with a learning rate of 0.0005. The final model is saved and later used for evaluation on the test set.

The final model was evaluated on the test set, and its predictions were converted into binary values using a threshold of 0.5. The classification metrics are stored and analyzed in Chapter 6. The trained model was saved in the `HDF5` format for reproducibility, along with the selected hyperparameters in the JSON format.

This approach allowed us to explore the representation capacity of neural networks for multi-label classification while balancing overfitting through regularization and by early stopping.

4.10.6 Classifier Chain (Logistic Regression)

Classifier Chain (CC) is a multilabel classification strategy that extends binary classification by modeling label dependencies. In contrast to the independent classifiers used in One-vs-Rest (OvR), CC constructs a sequence of classifiers, where each model predicts one label and receives all previous label predictions as additional features. This setup enables the model to capture label correlations, which is especially useful in real-world multi-label tasks, such as DEMPE classification.

For this study, we used `ClassifierChain` from the `scikit-learn` library with `LogisticRegression` as the base estimator. The input features were first standardized using `StandardScaler`, and the label columns were selected based on the `DEMPE_Class_` prefix.

To optimize the regularization parameters of the logistic regression model, we conducted a 3-fold `GridSearchCV` over the following hyperparameter grid:

- Regularization strength: $C \in \{0.01, 0.1, 1, 10\}$
- Penalty: $\mathit{penalty} \in \{\mathrm{l1}, \mathrm{l2}\}$

The evaluation metric for model selection was `F1-micro`, which was chosen for its robustness in multilabel settings by aggregating the performance across all labels.

After hyperparameter tuning, the best configuration was:

- $C = 10$
- $\mathit{penalty} = \mathrm{l1}$

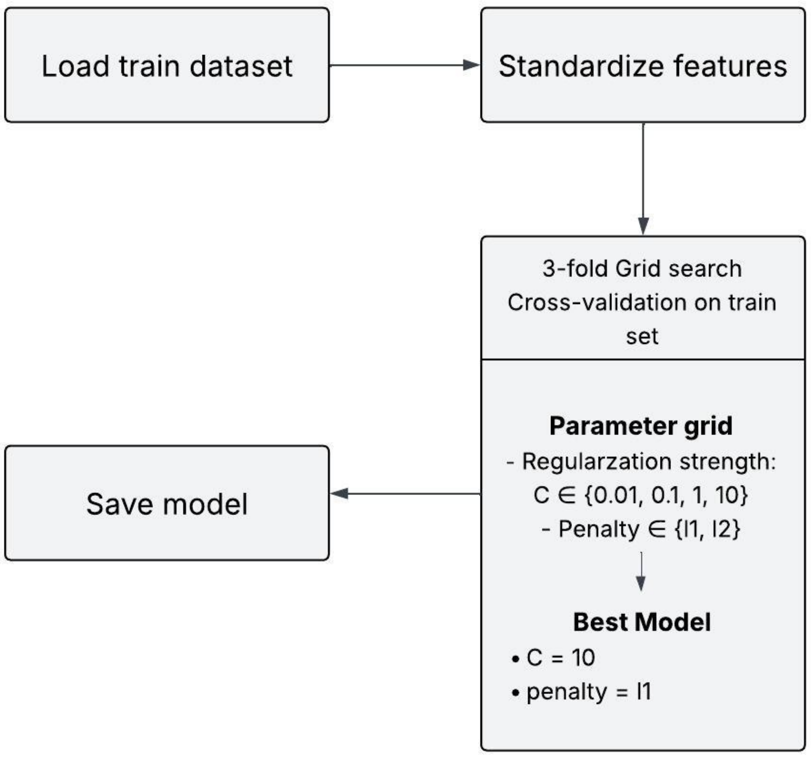
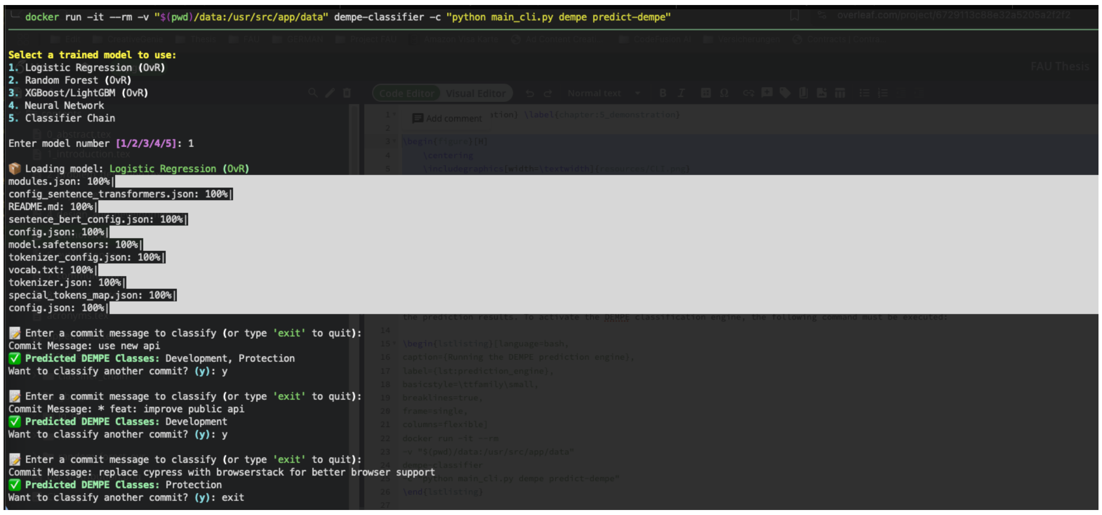


Figure 4.15: Training pipeline for the Classifier Chain model using Logistic Regression. The pipeline begins with loading and standardizing the training dataset, followed by a 3-fold hyperparameter optimization using GridSearchCV. Each classifier in the chain receives the predictions of the preceding classifiers as additional inputs, allowing the label dependency to be modeled. The best-performing configuration is selected based on the F1-micro score and saved for evaluation.

This configuration achieved the best cross-validated F1-micro score of 0.9118. The final model was serialized using `joblib` and evaluated on the test dataset. Further analysis of the prediction results is presented in Section 6.

5 Demonstration

Based on the evaluation results presented in Chapter 6, the Logistic Regression (One-vs-Rest) classifier emerged as the most effective model for the DEMPE classification task. In this chapter, we demonstrate the practical utility of our developed CLI tool by showcasing how it can be used to classify both conventional and non-conventional commit messages into the corresponding DEMPE functions.



```
docker run -it --rm -v "$(pwd)/data:/usr/src/app/data" dempe-classifier -c "python main_cli.py dempe predict-dempe"

Select a trained model to use:
1. Logistic Regression (OvR)
2. Random Forest (OvR)
3. XGBoost/LightGBM (OvR)
4. Neural Network
5. Classifier Chain

Enter model number [1/2/3/4/5]: 1

📦 Loading model: Logistic Regression (OvR)
modules.json: 100%|
config_sentence_transformers.json: 100%|
README.md: 100%|
sentence_bert_config.json: 100%|
config.json: 100%|
model.safetensors: 100%|
tokenizer_config.json: 100%|
vocab.txt: 100%|
tokenizer.json: 100%|
special_tokens_map.json: 100%|
config.json: 100%|

📝 Enter a commit message to classify (or type 'exit' to quit):
Commit Message: use new api
✅ Predicted DEMPE Classes: Development, Protection
Want to classify another commit? (y): y

📝 Enter a commit message to classify (or type 'exit' to quit):
Commit Message: * feat: improve public api
✅ Predicted DEMPE Classes: Development
Want to classify another commit? (y): y

📝 Enter a commit message to classify (or type 'exit' to quit):
Commit Message: replace cypress with browserstack for better browser support
✅ Predicted DEMPE Classes: Protection
Want to classify another commit? (y): exit
```

Figure 5.1: Interactive usage of the DEMPE classification CLI tool. The terminal interface demonstrates how users can load a trained model (such as Logistic Regression) and input either conventional or non-conventional commit messages for classification. The predicted DEMPE functions (for example, Development and Protection) are returned in real-time, highlighting the tool's applicability for automated analysis of software contributions.

While complete documentation of the CLI tool can be found in Appendix A, this section focuses solely on demonstrating the classification engine's functionality. All classifiers were pretrained and stored in the `data/models/` directory. To launch the DEMPE classification engine and interactively classify commit messages, command 15 can be executed.

For demonstration purposes, we selected 10 random entries from two sources: (i) conventional commit messages found in `data/csv_data/cleaned_commits.csv`, and (ii) non-conventional commit messages from `data/csv_data/non_conventional_commits.csv`. These commits were then classified using the trained Logistic Regression model, and the predicted DEMPE labels are reported in Tables 5.1 and 5.2, respectively.

No.	Conventional Commit Message	Predicted DEMPE Class
1	`fix: improve detection of external scroll`	Maintenance
2	`chore: package update dependencie`	Maintenance
3	`BREAKING CHANGE: api changed`	Enhancement
4	`test: improve tests`	Protection
5	`build: harden release-please.yml permissions`	Exploitation
6	`feat: improve public api`	Development
7	`docs: update readme`	Maintenance
8	`build: node require node`	Exploitation
9	`fix: browser fix shims so that yargs continues working in browser context`	Maintenance
10	`breaking change: singleton usage of yargs yargsfoo yargsargv has been removed`	Enhancement

Table 5.1: Examples of conventional commit messages and their corresponding DEMPE classification predicted by the Logistic Regression model. Each message follows a standard conventional commit format such as `feat`, `fix`, `test`.

No.	Non-Conventional Commit Message	Predicted DEMPE Class
1	Add missing `}` in widget	Development
2	Removes white border style when hovering over "Add more to home" on compact layout	Maintenance
3	Improved performance of the analytics dashboard	Enhancement
4	Adds a test case covering login edge cases	Protection
5	Build on latest SNAPSHOT of uPortal-app-framework	Exploitation
6	Replace Cypress with BrowserStack for better browser support	Protection
7	Update `@types/node` to the latest version (#45)	Enhancement
8	Returns `true` if element is visible from its container #14	Maintenance, Protection
9	Activate Feedback Web Component in header	Maintenance
10	Merge pull request #1042 from `nogalpaulina/widget-action-focus-style`	Development, Maintenance

Table 5.2: Predicted DEMPE classifications for a set of non-conventional commit messages using the Logistic Regression (One-vs-Rest) classifier. Each message was taken from real-world repositories and lacked standardized commit syntax (for example, no prefixes like `feat:`, `fix:`). The results demonstrate the classifier's ability to infer business-related functions (such as Development, Maintenance, and Protection) even from informal or irregular commit styles, highlighting its robustness in practical development environments.

These demonstrations validate the system's ability to generalize across both conventional and nonconventional commit formats, showcasing its robustness in real-world scenarios. The consistent alignment between the predicted DEMPE classes and the semantic intent of the input messages reinforces the effectiveness of the classification pipeline. Overall, the outputs provide strong evidence of the practical relevance and technical reliability of the system for semantic commit analysis in software engineering contexts.

6 Evaluation

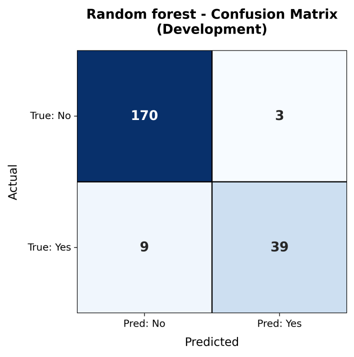
In this section, we present the evaluation performance of our classifiers and discuss the corresponding results obtained. Figure 6.1 illustrates the confusion matrices for each class using the Random Forest (one-vs-rest) model. Subsequently, similar confusion matrix visualizations for the remaining classifiers are shown in Figure 6.2, 6.3, 6.4, and 6.5 .

Confusion matrices offer valuable insights into the classification behavior of each model, including the true positives, false positives, false negatives, and true negatives. Based on these matrices, we calculated the key evaluation metrics: Accuracy, Precision, Recall, and F1-Score. Table 6.1 shows the attributes of the confusion matrices, their meanings, and the formulas used to evaluate the models.

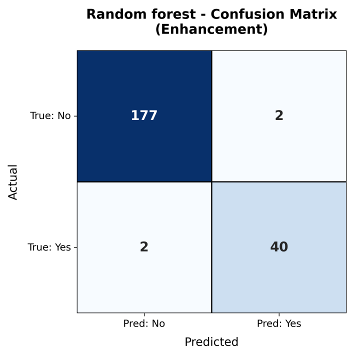
The final performance scores for all classifiers across the DEMPE classes are summarized in Table 6.2. The results on the test sets demonstrated consistently high classification performance across most models and classes.

Attribute / Metric	**Meaning**	**Formula**
True Positive (TP)	Instances correctly predicted as positive.	–
True Negative (TN)	Instances correctly predicted as negative.	–
False Positive (FP)	Negative instances incorrectly predicted as positive (Type I error).	–
False Negative (FN)	Positive instances incorrectly predicted as negative (Type II error).	–
Accuracy	Overall correctness of the model across all predictions.	$\frac{TP+TN}{TP+TN+FP+FN}$
Precision	Proportion of positive predictions that were actually correct.	$\frac{TP}{TP+FP}$
Recall	Proportion of actual positives that were correctly identified.	$\frac{TP}{TP+FN}$
F1-Score	Harmonic mean of Precision and Recall, balances the two.	$\frac{2\cdot\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}$

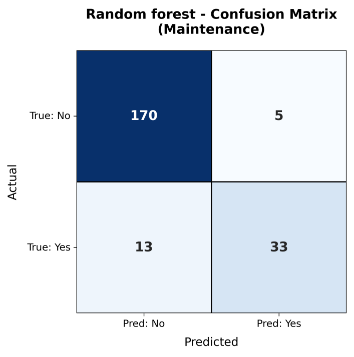
Table 6.1: Definitions and formulas for confusion matrix attributes and classification performance metrics. These metrics are fundamental to evaluating the effectiveness of classification models.



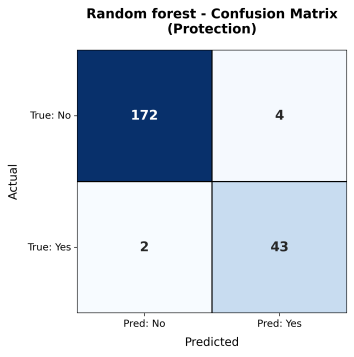
(a) Development



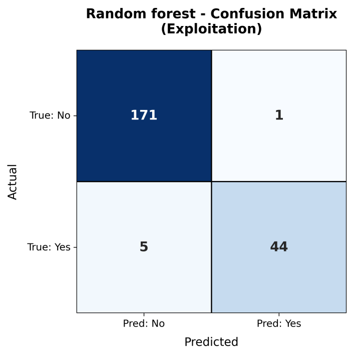
(b) Enhancement



(c) Maintenance

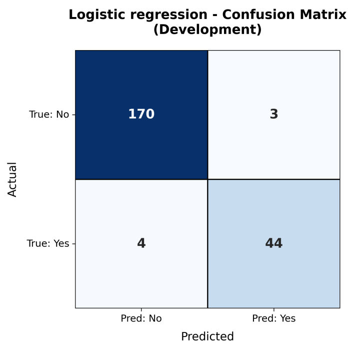


(d) Protection

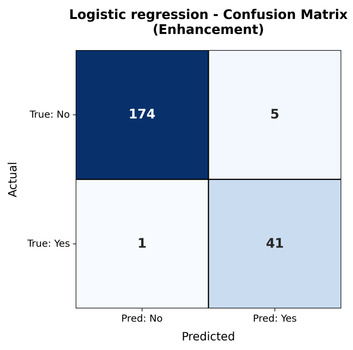


(e) Exploitation

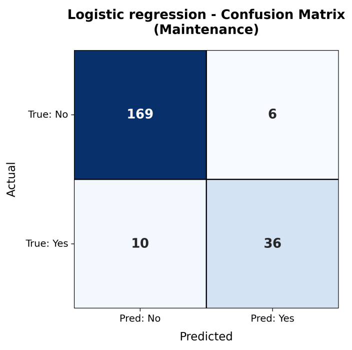
Figure 6.1: Confusion matrices for the **Random Forest (One-vs-Rest) classifier** across the five DEMPE classes. Each subfigure corresponds to a binary classification result for one class, where the positive class is defined by the presence of the respective DEMPE label. The matrices display the number of true positives, false positives, true negatives, and false negatives, offering insights into the class-wise performance of the model. These visualizations serve as the basis for the evaluation metrics reported in Table 6.2.



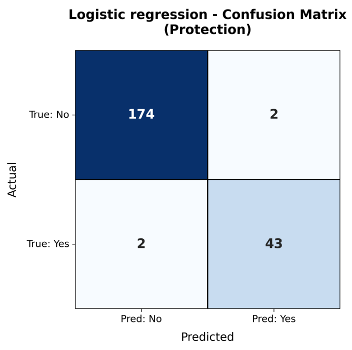
(a) Development



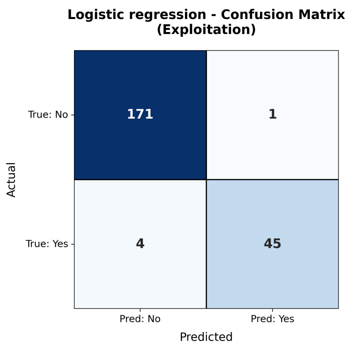
(b) Enhancement



(c) Maintenance

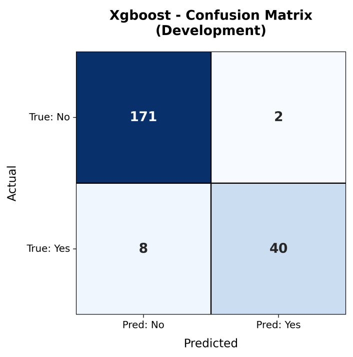


(d) Protection

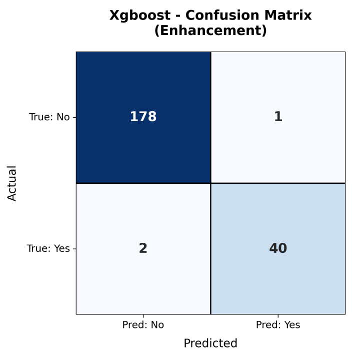


(e) Exploitation

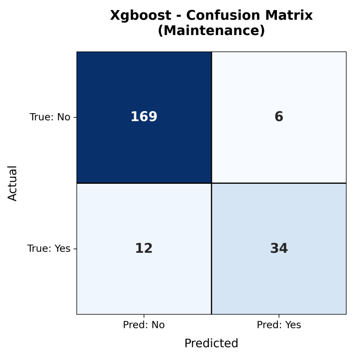
Figure 6.2: Confusion matrices for the **Logistic Regression (One-vs-Rest) classifier** across the five DEMPE classes. Each sub-figure corresponds to a binary classification result for one class, where the positive class is defined by the presence of the respective DEMPE label. The matrices display the number of true positives, false positives, true negatives, and false negatives, offering insight into the model's class-wise performance. These visualizations serve as the basis for the evaluation metrics reported in Table 6.2.



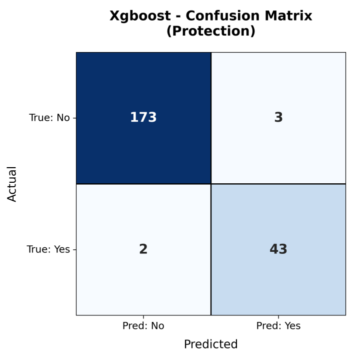
(a) Development



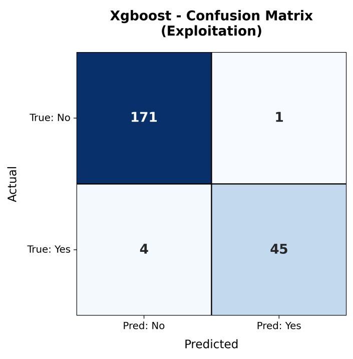
(b) Enhancement



(c) Maintenance

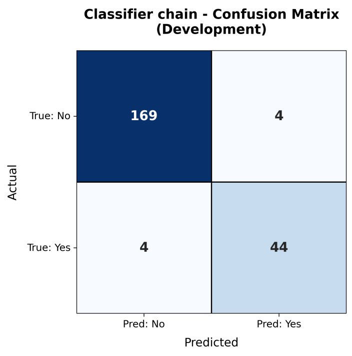


(d) Protection

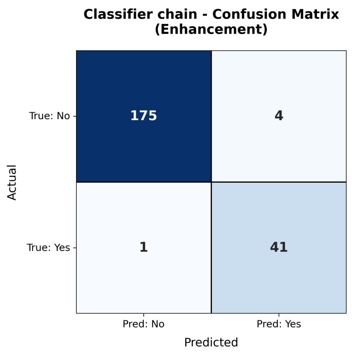


(e) Exploitation

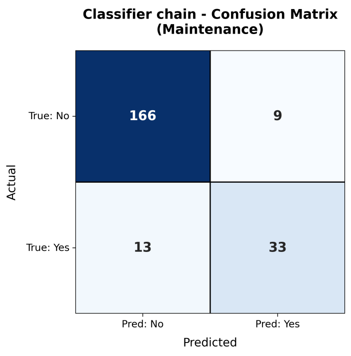
Figure 6.3: Confusion matrices for the **XGBoost classifier** across the five DEMPE classes. Each sub-figure corresponds to a binary classification result for one class, where the positive class is defined by the presence of the respective DEMPE label. The matrices display the number of true positives, false positives, true negatives, and false negatives, offering insight into the model's class-wise performance. These visualizations serve as the basis for the evaluation metrics reported in Table 6.2.



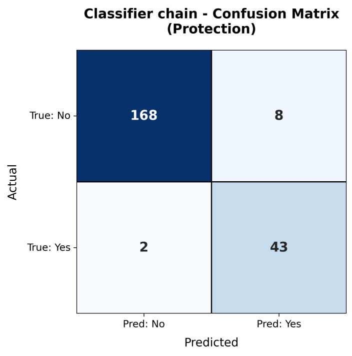
(a) Development



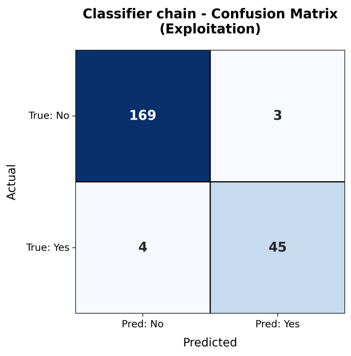
(b) Enhancement



(c) Maintenance

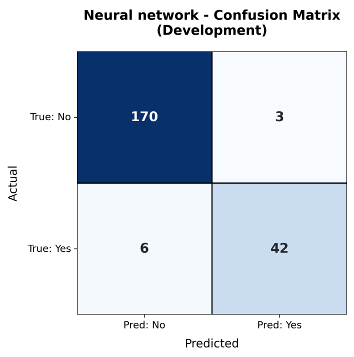


(d) Protection

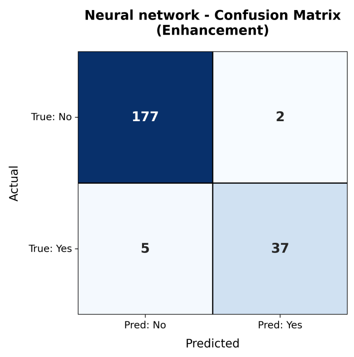


(e) Exploitation

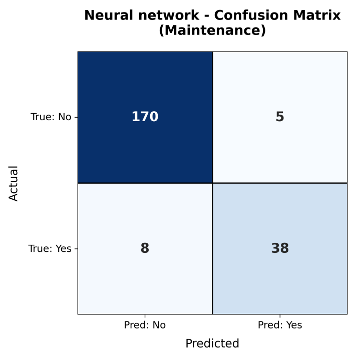
Figure 6.4: Confusion matrices for the **Classifier Chain** across the five DEMPE classes. Each sub-figure corresponds to a binary classification result for one class, where the positive class is defined by the presence of the respective DEMPE label. The matrices display the number of true positives, false positives, true negatives, and false negatives, offering insight into the model's class-wise performance. These visualizations serve as the basis for the evaluation metrics reported in Table 6.2.



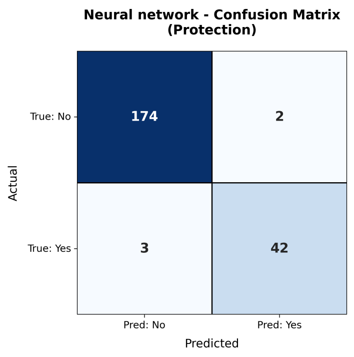
(a) Development



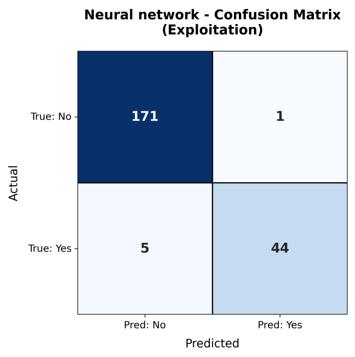
(b) Enhancement



(c) Maintenance



(d) Protection



(e) Exploitation

Figure 6.5: Confusion matrices for the **Feed forward Neural Network** across the five DEMPE classes. Each sub-figure corresponds to a binary classification result for one class, where the positive class is defined by the presence of the respective DEMPE label. The matrices display the number of true positives, false positives, true negatives, and false negatives, offering insight into the model's class-wise performance. These visualizations serve as the basis for the evaluation metrics reported in Table 6.2.

Model	Metric	Devel-opment	Enhance-ment	Mainte-nance	Protec-tion	Exploi-tation	Avg.
Random Forest	Accuracy	0.946	0.981	0.919	0.972	0.973	0.958
	Precision	0.927	0.952	0.868	0.915	0.978	0.928
	Recall	0.813	0.952	0.717	0.956	0.898	0.867
	F1-Score	0.867	0.952	0.786	0.934	0.936	0.895
Classifier Chain	Accuracy	0.963	0.977	0.900	0.955	0.968	0.953
	Precision	0.917	0.911	0.786	0.843	0.938	0.879
	Recall	0.917	0.976	0.717	0.956	0.918	0.897
	F1-Score	0.917	0.943	0.750	0.896	0.928	0.887
Neural Network	Accuracy	0.960	0.973	0.932	0.982	0.977	0.965
	Precision	0.953	0.950	0.860	0.977	0.978	0.944
	Recall	0.854	0.905	0.804	0.933	0.918	0.883
	F1-Score	0.901	0.927	**0.831**	0.954	**0.947**	0.912
XGBoost	Accuracy	0.955	0.986	0.919	0.977	0.977	0.963
	Precision	0.952	0.976	0.850	0.935	0.978	0.938
	Recall	0.833	0.952	0.739	0.956	0.918	0.880
	F1-Score	0.889	**0.963**	0.790	0.945	**0.947**	0.907
Logistic Regression	Accuracy	0.097	0.973	0.928	0.982	0.978	0.966
	Precision	0.936	0.891	0.857	0.956	0.978	0.923
	Recall	0.917	0.976	0.783	0.956	0.918	0.910
	F1-Score	**0.926**	0.932	0.818	**0.956**	**0.947**	**0.916**

Table 6.2: Per-class classification performance of five different models on the commit messages, evaluated using Accuracy, Precision, Recall, and F1-score for each DEMPE class. The best F1-score per class is highlighted in bold. The final column shows the average metrics across all classes, enabling a comparison of the overall model. *Logistic Regression achieved the highest average F1-score (0.916)*, followed closely by Neural Network and XGBoost. Since F1-score balances both false positives and false negatives, it is used as the primary metric for determining the best-performing model.

6.1 Summary

Table 6.2 presents the per-class classification performance of all five models using Accurecy, Precision, Recall, and F1-Score metrics for each DEMPE category. Among the models, *Logistic Regression achieved the highest average F1-Score of 0.92 across all classes*, indicating a robust and consistent performance. It demonstrated particularly strong performance in the *Development*, *Protection*, and *Exploitation* classes, achieving the best F1-Scores in those categories.

XGBoost and the Neural Network also performed competitively, each achieving an average F1-Score of 0.91, with XGBoost securing the best F1 in the *Enhancement, Exploitation* class and the Neural Network in the *Maintenance, Exploitation* class.

Overall, Logistic Regression demonstrated the most balanced and reliable classification performance across all DEMPE functions, making it the best-performing model in this multi-label classification task.

6.1.1 Discussion

Why Logistic Regression Worked Best: Out of all the classifiers, Logistic Regression had the highest average F1-score. Its robustness, simplicity, and strong generalization ability in high-dimensional semantic spaces derived from Sentence-BERT embeddings are responsible for this superior performance. These embeddings capture contextual meanings that go beyond word counts, and Logistic Regression's linear decision boundaries help avoid overfitting, which makes it ideal for input representations that are both dense and interpretable.

Failure Analysis: Most models consistently struggled with the *Maintenance* class, as evidenced by its relatively lower recall and F1-scores across all classifiers. This indicates a higher rate of false negatives, potentially due to the lexical and contextual similarity of maintenance related commit messages with other DEMPE functions. Such an overlap introduces ambiguity that even semantically rich embeddings may not fully resolve.

Implications of the Findings: These findings imply that multi-label classification tasks involving brief technical texts can be successfully completed by straightforward, interpretable models such as Logistic Regression. Their impressive performance demonstrates the viability of using such models in real-world settings where interpretability and dependability are crucial, like software quality pipelines or automated commit tagging in version control systems. However, to better differentiate between semantically similar classes and increase classification granularity in ambiguous cases, further advancements are required.

7 Conclusion

7.1 Summary of the Research

This thesis explored the application of machine learning techniques to classify software development activities under the DEMPE functions using commit messages from real software repositories. The primary objective was to develop a systematic and automated approach to infer accounting-related functions from textual commit data, thereby supporting classification in the context of transfer pricing.

To achieve this, the study employed the **Design Science Research Methodology (DSRM)** as its guiding framework. The DSRM facilitated a structured progression from problem identification to artifact design, implementation, and evaluation. The core artifact developed was demonstrated by a proof-of-concept: **command-line interface (CLI)** tool that automates data preprocessing, multi-label classification, and evaluation of commit messages using machine learning models.

7.2 Key Findings

The developed artifact was evaluated using multiple classifiers, including Logistic Regression, Random Forest, and XGBoost with One-vs-Rest (OvR), as well as Neural Networks and Classifier Chain strategies. The evaluation results indicated that the models, particularly Logistic Regression (OvR), performed effectively on conventional commit messages, closely followed by neural network and XGBoost.

The following are the major contributions of this study:

Key Contributions

- **Practical Implementation:** Developed a machine learning based system to classify software development activities under the DEMPE functions using real-world commit logs.
- **Dataset Creation Pipeline:** Designed a pipeline that leverages conventional commit syntax and manual annotation to generate a multi-label training dataset.
- **Modular CLI Tool:** Built a reproducible, containerized CLI application that enables seamless experimentation and deployment of the classification pipeline.

These findings contribute both methodologically by offering a reusable and extensible classification framework and practically by enabling tax and audit professionals to gain actionable insights from the software repositories.

7.3 Limitations and Challenges

Throughout the course of this research, several limitations and challenges emerged that shaped the study's scope and outcomes. Understanding these challenges is critical for contextualizing the results and identifying areas for future improvements.

7.3.1 1. Inconsistent Commit Message Practices

One of the primary challenges encountered during dataset construction was the inconsistent use of structured or meaningful commit messages across repositories. Despite selecting well-maintained and widely used open-source projects that were presumed to follow conventional commit guidelines (prefixes such as `feat:`, `fix:`, `test:`), a significant portion of the collected data deviated from these conventions.

Out of the 572 collected commit messages, 263 were categorized as nonconventional. These messages lacked the expected structures. It highlights the broader issue in real-world software development where commit hygiene is not uniformly maintained, *posing a barrier to just tag-based commit classification.*

7.3.2 2. Domain and Language-Specific Dataset Bias

Another notable limitation stems from the composition of the dataset. The commits were sourced from open-source projects, with the majority written in JavaScript or closely related frameworks. While this choice was intentional to maintain consistency in the domain and development practices, it potentially limits the generalizability of the trained models.

Commit messages may vary significantly in structure, vocabulary, and semantic style across different programming communities (such as Python, C++, and Java) or software domains (for example, data science, embedded systems, and web development). Therefore, a model trained predominantly on JavaScript-based projects might exhibit reduced performance when applied to repositories from other ecosystems. Future studies should consider expanding the dataset to include more diverse programming languages and domains to enhance robustness and cross-domain applicability.

7.4 Future Work

Several promising directions exist for future research.

7.4.1 1. Model Specialization per Class

The classification performance results in Table 6.2 reveal that the different models exhibit varying strengths across specific DEMPE classes. For instance, the Neural Network and XGBoost models show superior performance in identifying the *Maintenance*, *Enhancement*, and *Exploitation* classes, while Logistic Regression performs best on *Development*, *Exploitation*, and *Protection.* These insights pave the way for the natural extension of this work to develop a hybrid classification framework, where the best-performing model is assigned to predict its corresponding high-performing class. For example:

- Use Logistic Regression for **Development Exploitation**, and **Protection**
- Use Neural Network for **Maintenance** and/or **Exploitation**
- Use XGBoost for **Enhancement** and/or **Exploitation**

This ensemble-by-class strategy could lead to an overall improvement in multi-label classification performance, leveraging the strengths of each model without retraining monolithic architectures.

7.4.2 2. Expert-in-the-Loop Evaluation of Non-Conventional Commits

In this study, non-conventional commits were deliberately excluded from the training because of their inconsistent syntax and unstructured formats. However, these commits represent a significant portion of real-world software-development activities.

A valuable future work direction would be as follows:

- Apply the trained DEMPE classifiers on the set of non-conventional commit messages.
- Then engage domain experts or project maintainers to manually validate the predicted DEMPE classes.

This "expert-in-the-loop" validation serves two purposes:

- Understand how well the models trained on conventional commits perform on noisier, informal data.
- Expert-reviewed predictions can be added as new labeled data, allowing for semi-supervised learning or active learning in future iterations.

Such a feedback mechanism can lead to better model robustness, real-world applicability, and possibly fine-tuned classifiers for informal commit styles in the future.

7.4.3 4. Generalization to Larger and Diverse Datasets

Finally, as more annotated datasets become available, testing these hybrid or ensemble strategies on broader repositories and languages (such as Python, Java, and C++) would further validate their robustness and applicability in real-world scenarios.

7.5 Final Thoughts

This study bridges the gap between software engineering data and business-relevant classification by leveraging machine learning within a structured design science framework. The developed artifact demonstrates that commit messages, although often short and informal, can carry sufficient signals to support a meaningful classification aligned with business functions.

Ultimately, this study offers a foundation for further academic exploration and industrial adoption, opening new opportunities for data-driven decision-making in tax, audit, and software project governance.

Appendices

A Command-Line Interface (CLI) for DEMPE Classification

This appendix documents the command-line interface (CLI) tool developed for the research project titled *"Deterministic Classification of Accounting Functions in Code Contributions"*. The CLI facilitates preprocessing, vectorization, multilabel resampling (using the approximated MLSMOTE), and model training. It supports a reproducible machine learning pipeline built on top of Python's Click framework and is available at:

Repository URL: https://github.com/islam15-8789/Deterministic-Classification-of-Accounting-Functions-in-Code-Contributions

A.1 Tool Capabilities

The developed CLI tool enables a complete pipeline for the deterministic classification of accounting-related business functions in the commit messages. The functionalities are containerized and can be executed using Docker. Below is a breakdown of its high-level capabilities:

Overview

- **Data Acquisition:** Fetches commits from configured repositories.
- **Preprocessing:** Extracts, cleans, and labels commit messages.
- **Visualization:** Plots label distribution and other data summaries.
- **Resampling:** Applies approximated multi-label SMOTE (MLSMOTE) for class balancing.
- **Modeling:** Trains various machine learning models and generates performance reports.
- **Encapsulation:** All steps are encapsulated as CLI commands runnable via Docker.

A.2 Build Instructions

Before running the CLI tool, the Docker image must be built locally from the project's repository. Ensure that Docker is installed on your system.

Step 1: Build the Docker Image

```
docker build -t dempe-classifier . --no-cache
```

Note: Building the Docker image may take some time. On a MacBook Air M2 (8GB RAM) setup, the initial build took approximately 10 min (609 s).

A.3 Key Docker Commands

The CLI tool is designed as a modular pipeline. To ensure successful execution, users should follow the commands **in the given sequence**, as each step is based on the outputs of the previous step.

Step 0: Prepare `repos.json`

Create a `repos.json` file in the project's root directory. This file defines which GitHub repositories the tool should fetch commits from.

Format:

```
[
  {
    "repo_name": "https://github.com/OWNER/REPO",
    "owner": "OWNER",
    "token": "ghp_yourGitHubTokenHere"
  }
]
```

Note: Replace `ghp_yourGitHubTokenHere` with a valid GitHub Personal Access Token. This token must have `repo` or `public_repo` access, depending on the repository visibility.

Once this file is ready, proceed with the following Docker-based pipeline.

Step 1: Build Docker Image

Command 1: This command builds a fresh Docker image named `dempe-classifier` using the Dockerfile in the current directory. The `-no-cache` option ensures a clean build by ignoring cached layers, which is useful for incorporating recent changes in dependencies or the source code. This image serves as the runtime environment for executing the DEMPE classification CLI tool.

```
docker build -t dempe-classifier . --no-cache
```

Step 2: Fetch Commits

Command 2: This command uses Docker to execute the textttfetch-commits pipeline stage of the DEMPE classification CLI tool. It mounts the current directory's texttdata folder and texttrepos.json file into the container and runs a Python script that fetches the commit data from the specified repositories listed in texttrepos.json. The retrieved raw commit messages are saved in the *data/raw_data* for further processing.

```
    docker run --rm \
      -v "$(pwd)/data:/usr/src/app/data" \
      -v "$(pwd)/repos.json:/usr/src/app/repos.json" \
      dempe-classifier \
      -c "python main_cli.py data fetch-commits --input-file
          repos.json --output-folder data/raw_data"
```

Step 3: Extract Raw Commit Messages

Command 3: This command executes the `extract-raw-commit-messages` step of the DEMPE classification CLI tool within a Docker container. It mounts the `data` folder from the local machine into the container and processes raw Git commit logs from the previously fetched repositories. The extracted commit messages were saved in a structured CSV file named *data/csv_data/raw_commit_messages.csv*. This step converts raw Git logs into analyzable text data for further cleaning and labeling.

```
docker run --rm \
  -v "$(pwd)/data:/usr/src/app/data" \
  dempe-classifier \
  -c "python main_cli.py data extract-raw-commit-messages --input-folder
     data/raw_data --output-file data/csv_data/raw_commit_messages.csv"
```

Step 4: Label Commit Messages

Command 4: This command runs the `label-commits` stage of the DEMPE classification CLI tool inside a Docker container. It mounts the local `data` directory and then executes a Python script that applies rule-based labeling to commit messages stored in *raw_commit_messages.csv*. Each commit is labeled as one of the DEMPE functional categories using predefined rules based on conventional commit tags. The labeled output is saved as *data/csv_data/labeled_commits.csv*, preparing the dataset for further preprocessing and model training.

```
docker run --rm \
  -v "$(pwd)/data:/usr/src/app/data" \
  dempe-classifier \
  -c "python main_cli.py data label-commits --input-file data/csv_data/
      raw_commit_messages.csv --output-file data/csv_data/labeled_commits
      .csv"
```

Step 5: Clean Labeled Commit Messages

Command 5: This command executes the `clean-commits` stage of the DEMPE classification CLI pipeline within a Docker container. It mounts the local `data` directory and invokes a Python script that cleans and preprocesses the labeled commit messages stored in *labeled_commits.csv*. The cleaning process involves text normalization (e.g., lower casing, removing metadata, punctuation, and irrelevant patterns) to enhance model readiness. The cleaned output is saved as *cleaned_commits.csv*, while commit messages that do not follow the conventional commit structure are filtered out and stored separately in *non_conventional_commits.csv* for further review or manual labeling.

```
docker run --rm \
  -v "$(pwd)/data:/usr/src/app/data" \
  dempe-classifier \
  -c "python main_cli.py data clean-commits --input-file data/csv_data/
      labeled_commits.csv --output-file data/csv_data/cleaned_commits.csv
       --nonconv-output data/csv_data/non_conventional_commits.csv"
```

Step 6: Visualize Cleaned Data

Command 6: This command runs the `visualize-cleaned-commits` phase of the DEMPE classification CLI pipeline within a Docker container. It takes as input the cleaned and preprocessed commit messages from *cleaned_commits.csv* and generates the visual analytics. These visualizations include the commit distribution by category, tag frequency, and class balance. The resulting plots are saved in the `data/plots` directory for further exploration or inclusion in reports.

```
docker run --rm \
  -v "$(pwd)/data:/usr/src/app/data" \
  dempe-classifier \
  -c "python main_cli.py data visualize-cleaned-commits --input-file data
      /csv_data/cleaned_commits.csv --output-dir data/plots"
```

Step 7: Apply MLSMOTE

Command 7: This command executes the `apply-mlsmote` stage of the DEMPE classification pipeline inside a Docker container. It takes the cleaned commit messages as input from *cleaned_commits.csv* and applies Multi-Label Synthetic Minority Oversampling Technique (MLSMOTE) to balance the dataset across multiple DEMPE classes. The resulting resampled dataset, which includes synthetic samples to address class imbalance, is saved as *resampled_mlsmote.csv* for subsequent model training.

```
docker run --rm \
  -v "$(pwd)/data:/usr/src/app/data" \
  dempe-classifier \
  -c "python main_cli.py data apply-mlsmote --input-file data/csv_data/
      cleaned_commits.csv --output-file data/csv_data/resampled_mlsmote.
      csv"
```

Step 8: Split Train/Test Dataset

Command 8: This command runs the `split-dataset` stage of the DEMPE classification pipeline using Docker. It reads the multi-label resampled commit dataset *resampled_mlsmote.csv* and splits it into training and testing subsets for machine-learning model development. The resulting datasets are saved as *train_re_sampled_mlsmote.csv* and *test_re_sampled_mlsmote.csv*, enabling proper model training and performance evaluation.

```
    docker run --rm \
    -v "$(pwd)/data:/usr/src/app/data" \
    dempe-classifier \
    -c "python main_cli.py data split-dataset --input-file data/csv_data/
        resampled_mlsmote.csv --train-output data/csv_data/
        train_re_sampled_mlsmote.csv --test-output data/csv_data/
        test_re_sampled_mlsmote.csv"
```

Step 9: Visualize Resampled Comparison

Command 9: This command triggers the `visualize-comparison-distribution` stage in the DEMPE classification pipeline using Docker. It compares the class distributions before and after applying the MLSMOTE algorithm by reading the cleaned dataset (*cleaned_commits.csv*) and the resampled dataset (*resampled_mlsmote.csv*). The resulting visualization is saved as *data/plots/class_distribution_comparison.png*, allowing visual verification of how resampling balanced the class distribution.

```
docker run --rm \
-v "$(pwd)/data:/usr/src/app/data" \
dempe-classifier \
-c "python main_cli.py data visualize-comparison-distribution
    --cleaned-file data/csv_data/cleaned_commits.csv --resampled-file
     data/csv_data/resampled_mlsmote.csv --output-image data/plots/
    class_distribution_comparison.png"
```

A.4 Model Training Pipeline

Once the dataset has been cleaned, labeled, and resampled, you can train all models using the following Docker commands. Each command executes a specific training routine inside the containerized environment and saves the corresponding model and evaluation results.

Prerequisite: Ensure the training and testing datasets are located at:
`data/csv_data/train_re_sampled_mlsmote.csv` and
`data/csv_data/test_re_sampled_mlsmote.csv`

Output: Trained models and evaluation reports will be saved in the `data/models` and `data/reports` directories.

Train Logistic regression (One-vs-Rest)

Command 10: This command launches the `train-one-vs-rest-lg` stage using Docker to train a Logistic Regression classifier in a One-vs-Rest configuration. The training process uses a resampled dataset stored in the mounted `data` directory. Logistic Regression is a linear model that assigns probabilities to multiple classes independently, making it well-suited for multilabel classification tasks.

```
docker run --rm \
-v "$(pwd)/data:/usr/src/app/data" \
dempe-classifier \
-c "python main_cli.py train train-one-vs-rest-ovr"
```

Train Random Forest (One-vs-Rest)

Command 11: This command runs the `train-random-forest-ovr` stage of the DEMPE classification pipeline using Docker. It initiates the training process of a Random Forest classifier in a One-vs-Rest (OvR) configuration on the preprocessed commit message dataset. The training logic is defined in the CLI script, and it reads from the default training dataset path mounted inside the container.

```
docker run --rm \
-v "$(pwd)/data:/usr/src/app/data" \
dempe-classifier \
-c "python main_cli.py train train-random-forest-ovr"
```

Train Gradient Boosting Model (e.g., XGBoost)

Command 12: This command runs the `train-gbm-ovr` pipeline using Docker to train a Gradient Boosting model (e.g., XGBoost) in a One-vs-Rest setup. The script uses the preprocessed and resampled dataset stored in the `data` directory. XGBoost is known for its ability to handle complex non-linear relationships with strong regularization and tree-based boosting, making it a powerful choice for multilabel classification.

```
docker run --rm \
-v "$(pwd)/data:/usr/src/app/data" \
dempe-classifier \
-c "python main_cli.py train train-gbm-ovr"
```

Train Neural Network (with Keras Tuner)

Command 13: This command triggers the `train-nn` pipeline to train a Neural Network model using Docker. The model was built using Keras, and its architecture was optimized using the Keras Tuner. This allows for hyperparameter tuning over layers, units, and optimizers. The model is trained on the resampled commit dataset for DEMPE classification.

```
    docker run --rm \
    -v "$(pwd)/data:/usr/src/app/data" \
    dempe-classifier \
    -c "python main_cli.py train train-nn"
```

Train Classifier Chain Model (with Logistic Regression Base)

Command 14: This command executes the `train-classifier-chain` training pipeline using Docker. It builds a Classifier Chain model with Logistic Regression as the base classifier. This technique models label dependencies by chaining binary classifiers, where the output of one classifier is passed as input to the next, improving performance on multilabel classification tasks such as DEMPE function prediction.

```
    docker run --rm \
    -v "$(pwd)/data:/usr/src/app/data" \
    dempe-classifier \
    -c "python main_cli.py train train-classifier-chain"
```

A.5 Run trained models

Command 15: Running the DEMPE prediction engine using Docker. The prediction engine loads the trained classification models and predicts the appropriate DEMPE categories (Development, Enhancement, Maintenance, Protection, Exploitation) for each commit message. The output contains predicted labels.

```
docker run -it --rm \
  -v "$(pwd)/data:/usr/src/app/data" \
  dempe-classifier \
  -c "python main_cli.py dempe predict-dempe"
```

A.6 Explore CLI Commands

You can explore all available CLI commands and their usage by appending –help to the main entry point. These commands provide a summary of the arguments and subcommands available in the tool.

List all top-level commands:

```
docker run --rm
dempe-classifier
-c "python main_cli.py" --help
```

Inspect commands under the `data` namespace:

```
docker run --rm
dempe-classifier
-c "python main_cli.py data" --help
```

Inspect commands under the `train` namespace:

```
docker run --rm
dempe-classifier
-c "python main_cli.py train" --help
```

Inspect commands under the `dempe` namespace:

```
docker run --rm
dempe-classifier
-c "python main_cli.py dempe" --help
```

These commands help users understand the structure and full capabilities of the CLI, making it easier to navigate, debug, and extend for future use.

References

Capraro, M., Dorner, M., & Riehle, D. (2018). The patch-flow method for measuring inner source collaboration, 515–525. https://doi.org/10.1145/3196398.3196417

Capraro, M., & Riehle, D. (2016). Inner source definition, benefits, and challenges. *ACM Computing Surveys*, *49*, 1–36. https://doi.org/10.1145/2856821

Charte, F., Rivera Rivas, A., Del Jesus, M. J., & Herrera, F. (2015). Addressing imbalance in multilabel classification: Measures and random resampling algorithms. *Neurocomputing*, *163*. https://doi.org/10.1016/j.neucom.2014.08.091

Christel, J. (2024). Legal and tax concerns of transferring IP rights within a group. https://www.dirkzwager.nl/en/insights/articles/legal-and-tax-issues-of-transfer-of-rights-within-a-group

Commons, I. (2021). *What is innersource?* [Accessed: 2025-01-12]. https://innersourcecommons.org

Constantino, K., Souza, M., Zhou, S., Figueiredo, E., & Kästner, C. (2021). Perceptions of open-source software developers on collaborations: An interview and survey study. *Journal of Software: Evolution and Process*, *35*. https://doi.org/10.1002/smr.2393

Conventional Commits Community. (2019). *Conventional commits 1.0.0*.

Dorner, e. a., M. (2024). Sustaining arm's length cost allocations for highly integrated development functions: An explorative case study of transfer pricing for innersource communities.

dos Santos, G. E., & Figueiredo, E. (2020). Commit classification using natural language processing: Experiments over labeled datasets. *Conferencia Iberoamericana de Software Engineering*. https://api.semanticscholar.org/CorpusID:229545423

Edison, H., Carroll, N., Morgan, L., & Conboy, K. (2020). Inner source software development: Current thinking and an agenda for future research. *Journal of Systems and Software*, *163*, 110520. https://doi.org/10.1016/j.jss.2020.110520

et al., B. H. (2021). *Intangibles in the world of transfer pricing*. Springer.

et al., O. T. (2022). Sustaining arm's length cost allocations for highly integrated development functions – an explorative case study of transfer pricing for innersource communities [https://mnetax.com/sustaining-arms-length-cost-allocations-for-highly-integrated-development-functions-an-explorative-case-study-of-transfer-pricing-for-innersource-communities-47288].

Exactera. (2021). Dempe for dummies [https://exactera.com/resources/dempe-for-dummies/].

Hevner, A., R, A., March, S., T, S., Park, Park, J., Ram & Sudha. (2004). Design science in information systems research. *Management Information Systems Quarterly*, *28*, 75–.

Hindle, A., Germán, D., & Holt, R. (2008). What do large commits tell us? a taxonomical study of large commits, 99–108. https://doi.org/10.1145/1370750.1370773

InnerSource Commons Community. (2020). Semantic release and its impact on software versioning [https://innersourcecommons.org]. *InnerSource Commons Blog*. Blog Post.

Islam, M., & Katiyar, V. (2014). Development of a software maintenance cost estimation model: 4 th gl perspective. *Volume 2*, 65–68.

Levin, S., & Yehudai, A. (2017). Boosting automatic commit classification into maintenance activities by utilizing source code changes, 97–106. https://doi.org/10.1145/3127005.3127016

Mitchell, T. M., et al. (1997). Machine learning.

OECD. (2014). *Guidance on transfer pricing aspects of intangibles*. https://doi.org/https://doi.org/https://doi.org/10.1787/9789264219212-en

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*, 45–77.

Reimers, N., & Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. https://arxiv.org/abs/1908.10084

Roscher, R., Bohn, B., Duarte, M. F., & Garcke, J. (2020). Explainable machine learning for scientific insights and discoveries. *IEEE Access*, *8*, 42200–42216. https://doi.org/10.1109/ACCESS.2020.2976199

RoyaltyRange. (2018). Dempe explained [https://www.royaltyrange.com/home/blog/dempe-explained].

Rudzika, K. (2018). Dempe explained. *RoyaltyRange Blog*. https://www.royaltyrange.com/home/blog/dempe-explained

Sabetta, A., & Bezzi, M. (2018). A practical approach to the automatic classification of security-relevant commits. *Published in the Proc. of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME) 2018.*

Semantic Versioning Community. (2013). *Semantic versioning 2.0.0* [https://semver.org/].

Sherer, S. A. (1991). *Software failure risk,* https://www.buecher.de/artikel/buch/software-failure-risk/37476197/.

Van Lessen, e. a. (2019). Collaboration strategies in innersource software development. *Journal of Open Software Practices*, *5*, 120–128.

Wan, Z., Xia, X., Zhang, Y., Lo, D., Zhou, D., Chen, Q., & Hassan, A. (2022, November). What motivates software practitioners to contribute to inner source? [Publisher Copyright: © 2022 ACM.; 30th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022 ; Conference date: 14-11-2022 Through 18-11-2022]. In A. Roychoudhury, C. Cadar & M. Kim (Eds.), *Esec/fse 2022 - proceedings of the 30th acm joint meeting european software engineering conference and symposium on the foundations of software engineering* (pp. 132–144). Association for Computing Machinery, Inc. https://doi.org/10.1145/3540250.3549148