Exploring Developer Patterns and Code Similarities: An In-Depth Study Using Code Stylometry, Developer Profiling, and Clone Detection in Software Projects

by

Apurba Afiat
23341096
Jumanah Suha Parisa
21201774
Sarker Md Talha
21201508
Sanzida Afrin
21201214
Taan Gazi Safowan Islam
21201384

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

Department of Computer Science and Engineering
Brac University
October 2025.

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Apurba Afiat
23341096

Taan Gazi Safowan Islam
21201384

Sanzida Afrin
21201214

Sarker Md Talha
21201508

Jumanah Suha Parisa
21201774

Approval

The thesis/project titled "Exploring Developer Patterns and Code Similarities: An In-Depth Study Using Code Stylometry, Developer Profiling, and Clone Detection in Software Projects" submitted by

1. Apurba Afiat (23341096)
2. Jumanah Suha Parisa (21201774)
3. Sarker Md Talha (21201508)
4. Sanzida Afrin (21201214)
5. Taan Gazi Safowan Islam (21201384)

of Summer, 2025 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science in October 16th, 2025.

Examining Committee:

Supervisor:
(Member)

Md Aquib Azmain

Lecturer
Department of Computer Science and Engineering
BRAC University

Co-Supervisor:
(Member)

Md Sabbir Ahmed

Lecturer
Department of Computer Science and Engineering
BRAC University

Thesis Coordinator:
(Member)

Dr. Golam Rabiul Alam

Professor
Department of Computer Science and Engineering
BRAC University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi

Associate Professor
Department of Computer Science and Engineering
BRAC University

Abstract

In recent years, code theft has become a significant threat to the software industry. Incidents of code being stolen and used anonymously are increasingly common, alongside other security challenges such as plagiarism, copyright disputes, software piracy, and unauthorized, anonymous contributions on platforms like GitHub. This paper presents a clear pipeline that uses code writing style, developer profiles, and a hybrid clone detector called CodeCloneX to find who wrote code and where code was reused. The dataset comprises source code samples from 10,000 prominent GitHub contributors, obtained through GitHub mining. We extracted simple file-level style features, then made per-developer profiles by averaging those features. We compared three clustering methods: KMeans, Agglomerative, and BIRCH and our clone detector can detect clones of about 90% of lines of code and 57% of files. These results show that combining style profiles with clone analysis helps detect likely authors and reused code in large codebases. By providing a framework for recognizing coding styles and optimizing team collaboration, the findings contribute to improved code readability, maintainability, and software project management.

Keywords: Code Stylometry, Clone Detection, Developer Profiling, Code Analysis, Code Metrics, Developer Behaviour

Acknowledgement

Our deepest thanks go to our supervisor and co-supervisor as well as our group mates who offered ongoing support along with guidance and encouragement during this research process. The essential input from our contributors served as the cornerstone for our study's success. We extend our special thanks to Allah because His infinite blessings and strength helped us address obstacles and finish this research. Our gratitude extends deepest to everyone who provided us help during this research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The fast growing software industry depends on development platforms such as GitHub where numerous developers can work together on both professional and open-source projects. Although it has been an amazing innovation which has made coding easier, it has also caused concerns regarding code authorship, security and privacy, and software integrity. It is now more important than ever to identify authorship and the origins of code due to the growing complexity of software programs and the demand for reliable and maintainable code. Code stylometry, developer profiling, and clone detection might offer possible answers to these issues. This study examines how a combined approach using these three can be used to address the major challenges in today's software engineering. Stylometry is typically used in forensic linguistics, yet it is only recently being used in the software field. It can be applied to study distinct coding patterns, allowing developer attribution based on their coding styles. Stylometric models using deep learning and Abstract Syntax Trees (ASTs) could determine authorship with as much as 98% accuracy [5]. However, as programmers change their coding style, this accuracy decreases, demanding the improvement of stylometric models to deal with intended mimicry and obfuscation. A recent research discussed the effects of code formatting and minimizing on stylometry, and results show that extensive code alteration can affect accuracy on detecting authors [24]. The Deep Learning-based Code Authorship Identification System (DL-CAIS) has demonstrated that stylistic evaluation functions across several programming frameworks can improve security and writer authentication processes [12] .

Developer profiling can analyse developer's work activity, coding patterns and contribution records in addition to standard authorship attribution methods. Organizations benefit from profiling by optimizing their team's structures while discovering individual team member strengths and maximizing productivity. In multi-author setups, where a single codebase may be modified by several developers, stylometry presents difficulties and hence developer profiling is a crucial supplement to stylometry. One such study proposed a fingerprinting technique which combines stylometry with machine learning to identify developer writing patterns[20]. Another model, the k-nearest neighbors (k-NN) classifier has a 71% accuracy rate in identifying developers[25]. These studies show that developer profiling can improve teamwork, identify irregularities, and prevent unethical changes in software projects.

Code cloning is a common practice but it also possesses some drawbacks. Code reuse can increase productivity, but unchecked duplication can result in maintenance problems, security vulnerabilities, and plagiarism issues. Gitor, a deep learning-based technique for finding Type-4 semantic clones, was recently introduced [21]. It focuses on logical similarities rather than lexical matching. Despite these developments, detecting clones in multiple language environments remains a challenge. It is also difficult to identify conceptual code similarities. This emphasizes the necessity for improved algorithms and therefore we used a combined approach with developer profiling, stylometry, and code clone detection in a single framework.

Hence, by addressing these critical research gaps, we intend to offer more effective tools which can contribute to software development. This topic has the potential to improve further and offer important insights for preserving software projects that remain sustainable and effective.

1.2 Motivation

This study aims to solve the growing issues in software development caused by complex teamwork and large-scale worldwide code production. In contemporary software development, initiatives typically include developers and scientists who employ different coding styles and have different levels of experience. Stylistic and clone detection algorithms become more important since different writers have different writing styles that affect code analysis for quality improvement as well as performance optimization and preventative security measures. One of the biggest challenges facing the industry is code authorship attribution, as IT professionals must locate the original developers or designers of code segments. Code authorship attribution satisfies critical business requirements since it helps developers keep track of their responsibilities and work assignments and alerts unauthorized users who jeopardize program security. Similar to how handwriting analysis identifies distinct patterns, stylometric analysis enables researchers to create unique developer profiles [5]. Collaboratively changing code becomes difficult. when a repository is being worked on by several developers. Another significant problem is code clones, which arise when developers utilize copy-paste methods or reuse old code structures, making it difficult to identify duplicates or extremely identical code blocks throughout a software project's infrastructure. While code reuse aids in development, unchecked duplication causes issues such as increased maintenance costs and software faults that propagate throughout the codebase, making it more difficult to integrate upgrades. Reducing technical debt in large software projects requires the ability to identify textual similarities as well as structural and semantic clones. Standard clone detection techniques are insufficient for detecting clones with comparable conceptual ideas but different syntactical patterns [17].

Combining stylometric research and software clone detection allows for creative solutions to important industry problems. Code profiling helps development teams find duplicate and incorrectly credited code segments by examining developer coding behavior. This improves developer accountability metrics and teamwork. Combining these techniques makes it possible to identify instances of plagiarism in academic or professional contexts and identify the origin of hostile cyberattack code. By giving software developers practical tools, this program seeks to fill in recognized

knowledge gaps. The research handles authorship attribution, clone detection, and collaborative development needs while concentrating on resolving real-world software difficulties. This allows enhancements to the requirements for software security and quality maintenance. The examination of data sets from the GitHub platform shows how stylometry and clone detection can be combined to support safe and efficient software development procedures.

1.3 Problem Statement

These days, software projects grow into complex structures where multiple development teams work together to produce large applications. Technical collaborations create difficulties in recognizing repetitive code and interpreting different coding standards in an attempt to enhance program quality. Research indicates that stylometry provides promising results for identifying certain code patterns through style analysis. Stylometry programming poses special challenges. It may be difficult to determine whether coding characteristics are the product of personal writing styles or actual programming methodologies. Modern clone detection tools mostly focus on syntax-based code similarities, ignoring crucial, meaningful relationship patterns found throughout the source. The use of existing code presents difficulties because it makes it hard to determine whether it represents purposeful code sharing or plagiarism. Monitoring and analyzing code reuse helps developers decrease code repetition and maintain software systems effectively. The evolving nature of developer collaborations together with coding practices makes tracking individual contributions increasingly difficult for large-scale and open-source projects. Various environmental situations challenge the capabilities of present-day assessment systems because they show limited success in finding solutions to these problems. To address these gaps, our study focuses on the following key questions:

RQ1: How can stylometry be extracted to identify unique developer styles and behaviors across different programming languages and environments?

RQ2: What are the best ways to detect code reuse and redundancy while ensuring code maintainability and preventing plagiarism?

RQ3: How can a combined approach using stylometry, developer profiling, and clone detection enhance collaboration, authorship attribution, and code quality in software projects?

These questions are what our study is built on. Our goal is to make better tools and methods that help us understand how developers work together and make sure that software projects have good code that can be kept up to date. These issues are being looked at so that the study can add something useful to the field of software engineering.

1.4 Objective

The aim of this research is to create innovative tools and approaches to address difficulties encountered in modern software development, where teams of engineers need

to collaborate on intricate, large-scale projects. It focuses on enhancing stylometry by combining stylometry, developer profiling, and clone detection to identify unique developer styles and find more effective ways to detect code reuse and redundancy to keep code clean while preventing plagiarism to improve teamwork, track contributions, and ensure high-quality code. Even though there are challenges in this study the main objective is to offer feasible solutions that enhance understanding of developer contributions, simplify coding procedures, and facilitate the production of exceptional, maintainable software in a variety of dynamic development environments. The objectives of this research encompass the following:

- To improve stylometry methods for recognizing distinct developer behaviors and styles across various programming languages and contexts and thus enabling more precise authorship attribution and developer profiling.
- To create efficient techniques for detecting and monitoring code redundancy and reuse while ensuring code maintainability, preventing plagiarism, and reducing unnecessary repetition in software projects.
- To develop an effective approach that enhances authorship attribution, code quality, and cooperation in software projects by utilizing developer profiling, clone detection, and stylometry. The goal is to understand the connection between these methods and how a combined approach can contribute to advancement in the software development field.

1.5 Methodology in Brief

Our study looks for clones by analyzing code similarities and development patterns in software projects through the use of code stylometry and developer profiling tools. The goal of this research is to create an accurate methodology for software code analysis that identifies authorship, detects code copying, and tackles resource optimization issues with software sustainability challenges. In Fig. 1.1, the proposed methodology is shown.

Quantitative data collection from publicly available software repositories is our aim, with a focus on diverse codebases that offer detailed documentation of all code changes and detailed information on developer contributions. We also want to use developer surveys and interviews to get qualitative data regarding developers' coding methods and project management approaches. To identify developer code styles and patterns, the gathered data would be processed and analyzed using sophisticated machine learning algorithms in conjunction with stylometric techniques. After that, we intend to use clone detection algorithms to find and assess patterns of resemblance across code bases. Our work plan describes an ethical assessment technique that makes use of metrics derived from source code reuse patterns and developer behavior. In order to improve code quality and operational performance, the current research produces valuable information regarding developer profiling approaches and sustainable development practices for software.

1.6 Scopes and Challenges

The primary goals of this research are to better understand and analyze the coding styles and behaviors of developers across different programming languages and environments by focusing on stylometry, code reuse and redundancy detection, and a combined approach that uses clone detection and developer profiling. The goal is to develop tools that enhance collaboration, authorship attribution, and code quality in software projects. Due to its limitation to publicly available statistics, such as team-based development projects and GitHub's open-source software repositories, the scope might not be typical of all software development settings. It is limited by datasets and relies on publicly available projects and repositories, which may not fully capture the diversity and complexity of all software development environments. This study has a tool dependency because it uses pre-existing tools and algorithms, which can be biased or limited in some way, especially when working with complex or uncommon coding techniques. Finally, the study's breadth and depth may be limited by the time and computational resources available for the research.

The scope and accuracy of this research are impacted by a number of restrictions. One significant issue that arises and is beyond our control is when the source file's stated author differs from the actual contributor. Furthermore, it can be difficult to correctly assign specific patterns or styles because code fragments can reflect the work of numerous authors. We plan to broaden the study to include datasets with similar complications in order to address this problem, but for now, we are focusing on git commits from open-source projects to look into the feasibility of classifying code that has been co-authored. The results might be more relevant to particular programming environments or languages and this study might not

fully capture every variation across all programming environments since developers' behavior and coding methods can differ greatly between languages. Additionally, the coding styles of developers may change over time or between projects, which could make profile consistency problematic. The accuracy of the detection approach may be compromised since stylometry and clone detection are not always perfect and may produce false positives or negatives. It can be challenging to adapt and integrate tools in real-world software development settings, especially in agile or fast-paced development contexts. Furthermore, despite focusing on the technical aspects of code quality and cooperation, the study may not adequately address the larger social and interpersonal dynamics that influence teamwork in software projects.

Chapter 2

Literature Review

2.1 Preliminaries

The use of code to analyze programming styles Stylometry finds distinctive elements in source code that are personal to an author. Programming code stylometry has attracted significant interest because it serves multiple purposes including authorship detection, copyright infringement inspection and software examination linked to security needs. Code stylometry analysis utilizes AST, CST along with k-NN and CNN technology and other advanced machine learning methods to detect unique programming elements in source code. The code analysis tools Gitor and Code2Vec refine code examination through exploration of both local and global code elements including control flow structures and loop definitions. However, many researchers have deeply investigated code formatting issues and language-specific traits and code minimization effects on stylometric analytical capabilities in the scientific literature. Multiple research reports show that code minification and reformats decrease the ability to detect programming authorship. A combination of static and dynamic analysis procedures has been integrated into existing techniques to manage problems from obfuscation and syntax mismatches. Recent research shows progress but the analyzed studies recognize key shortfalls in current methods related to dataset range and multi-lingual capability and managing collaborative development contexts. Research opportunities exist to develop stronger code stylometry systems which work across diverse programming code applications.

2.2 Review of Existing Research

Ningfei Wang and colleagues in 2018 [13] developed an innovative code de-anonymization framework which combines static and dynamic stylometry analysis to surpass conventional methods that use static features alone. Traditional methods struggle to handle syntax differences and cannot analyze compiled binaries which demonstrates the requirement for a solution that identifies distinctive code execution patterns. A programmer's writing style can be better examined with the help of the new framework which efficiently gathers dynamic elements like CPU and memory usage patterns. This approach combines static and dynamic analysis, enabling improved accuracy and increased robustness for de-anonymization tasks. The proposed method opens up new directions for programming forensics research towards multi-contributor programming scenarios and bug tracing applications. By laying the groundwork for

future coding style analysis research, this study enhances the dependability of code de-anonymization applications.

Yang et al. (2023) created Asteria-Pro using their novel approach to identify firmware vulnerabilities and detect binary code similarity [22]. In order to address problems resulting from diverse IoT hardware code formats, Asteria-Pro enhances its predecessor Asteria by combining deep learning with domain expertise. The authors' findings demonstrate that standard detection methods suffer from inaccuracies and inefficiencies due to disparate coding procedures. In addition to a complete training model, Asteria-Pro employs a sophisticated computational framework to increase the identification accuracy for homologous susceptible functions. When compared to current technological solutions, the evaluation data demonstrates that the suggested changes produced fewer false positives while providing superior detection performance. Software security implications are examined in the study, which also emphasizes the need for sophisticated approaches to develop in tandem with rapidly evolving software. With the incorporation of contemporary computational techniques that satisfy present security requirements, Asteria-Pro offers revolutionary advancements in safe firmware systems.

The stylometric model for authorship attribution and identifying document changes in a multi-authored document based on style shifts between paragraphs was investigated in another study by Zamir et al [26]. Even when there are several authors, stylistic analysis uses syntax, punctuation, and vocabulary to determine authorship. The difficulties of recognizing distinct writing styles and author changes over time in a paragraph were discussed in the paper. They emphasized the application of this technique in a number of domains, including literary analysis, historical document verification, forensic linguistics, etc. This study advances the field of computational linguistics by examining authorship and style.

A study by Balla et al. (2024) explores the effects of code formatting and minification on code stylometry accuracy through machine learning-driven authorship detection methods built upon programming styles. The researchers implemented a code2vecbased classifier to analyze Python code from the Google Code Jam dataset through Concrete Syntax Tree (CST) and Abstract Syntax Tree (AST) representations from 59 different authors. The model registered a 68% success rate for original unformatted code yet it fell to 53% accuracy during the code formatting process and dropped further to 50% with code minification. The study examines software development simplification techniques in contrast to stylometric analysis. However it emphasizes the necessity of extending testing to other programming languages and settings in order to improve external validity [24].

To differentiate between various writing patterns, Iqbal et al.[20] suggested a fingerprinting technique that combines machine learning techniques with stylometric approaches. This approach mainly concentrated on gathering the distinctive characteristics of stylometry, which serve as indicators of an author's writing style, such as syntactic structures, word frequency distributions, and character n-grams. Many machine learning classifiers were used to identify authors from text samples and

then the accuracy of this model was compared to traditional ones. Stylometric methods combined with machine learning techniques showed better performance than traditional statistical models in identifying authors from a small collection of texts. The study shows how stylometric techniques and machine learning can help with author identification. It has practical uses, such as verifying documents in forensic cases. Even with complex datasets, this model can offer a solid solution for author attribution that uses both stylometric analysis and machine learning for more precise results. This suggests that the recommended identification methods can offer solutions for improving the accuracy and efficiency of author identification tasks.

A k-nearest neighbors (k-NN) classifier solution is shown in another work by Gurioli et al. (2024) to identify authors of open-source code, with a focus on high-level expert contributions to significant real-world projects. For a fresh data collection, their work used 114,400 code snippets authored by 104 contributors. Strong identification capabilities beyond the training writers were demonstrated by the system's peak accuracy rate of 71% when it recognized authors from out-of-distribution categories. The study's main drawback emerges from its restricted dataset diversity which can disregard possible coding style modifications across cooperative programmer environments thus affecting team-based software development projects [25].

Zakeri-Nasrabadi et al. (2023) analyzed source code similarity measurements through a systematic literature review which included 136 studies and 80 software tools focusing on.Clone detection as well as Plagiarism detection and Code recommendation methodologies. Their research found that Java tool support represented half of tool deployments while C/C++ tools made up 37% of the market yet similar tools for alternative programming languages were scarce. The review discovered limited access to source code datasets because twelve were available while only eight could be accessed publicly. The analysis found limited evidence of hybrid methods that utilize several similarity detection methods and insufficient support for multi-paradigm languages that require detection across different languages. This opens space for the creation of future tools that use machine learning models to provide dynamic code similarity identification capabilities that are reliable and flexible [23].

Another study by Sarnot et al. (2019) offers a novel method for code stylistic analysis that uses deep convolutional neural networks and transfer learning to analyze programming styles. SnapCode, which removes the need for manual feature extraction, converts code snippets into images and extracts structural patterns. This innovative approach analyzes the structural characteristics and distinctive writing style of each programmer's code to show its usefulness in tasks like author identification, cybersecurity investigations, and plagiarism detection. The results highlight the usefulness of SnapCode and its wider relevance to comprehending programming behavior. The paper does not address the computational efficiency of the VGG-CNN model, which is known to be resource-intensive. Lightweight models or methods for enhancing the resource efficiency of existing designs may be the subject of future research [14].

Another work by Rattan et al. (2015) investigated methods for finding conceptual

clones that represent overlapping domain concepts, including principal component analysis (PCA) and latent semantic indexing (LSI) [6]. Making tools to identify high-level similarities between source code and design models is a major addition to this discipline. By using Unified Modeling Language (UML) diagrams in addition to code, this method makes it easier to identify structural similarities at the design level. By using multi-version program analysis, these methods have proven to be effective and scalable for understanding and sustaining changing software systems. According to the study, high-level abstractions are essential for enhancing software quality, including extensibility, reusability, and maintainability. The suggested approach is compared to other high-level similarity detection methods in the paper, but it is not compared against the most advanced methods for software evolution analysis or clone detection. It's still unknown how novel and effective the methodology is in relation to more recent approaches.

Caliskan-Islam et al. (2015) investigates the potential for source code authorship attribution through machine learning methods applied to coding style. Using a novel Code Stylometry Feature Set derived from Abstract Syntax Trees (ASTs), the authors demonstrate the ability to de-anonymize programmers with remarkable accuracy (up to 98%) [5].Their datasets were C++ code sourced from the "Google Code Jam" competition. The research highlights that experienced programmers and solutions to complex tasks are more readily attributable. Despite achieving robustness and scalability across programming languages, there are still issues with obfuscation countermeasures. The study's difficulties include addressing code fragments from numerous authors, determining whether the source code belongs to the declared author, and the potential for imitation attacks brought on by the addition of dummy code; all of these require thorough verification [5].

Tereszkowski-Kaminski* et al.'s study from 2022 used code stylometry to solve the difficulties presented by unstructured and incomplete datasets, like those seen in underground forums [19]. The noise and volatility of such data can pose challenges for traditional approaches. A methodology that combines a forum-specific learning pipeline with conformal prediction was put forth to address this. This approach improves author identification reliability by utilizing sophisticated preprocessing techniques such as function-level analysis and clone detection. The method greatly improves attribution accuracy even when open-world assumptions are used, which makes it a useful tool for cybersecurity applications like evaluating illegal code-sharing activities and de-anonymizing harmful actors. The research requires substantial text lengths for maximum accuracy and encounters challenges when working with a large number of authors. In some circumstances, these restrictions may render the tool less helpful.

This research done by Junjie Shan et al. (2024) [21] utilizes the Gitor system to detect Type-4 semantic clones while also acknowledging code variations in style between identification. Gitor employs Word2vec/Code2vec along with global code graphs and loop counts and control logic features for precise clone detection. Testing on 100 million lines of code, established Gitor surpasses existing BigCloneBench detectors in both accuracy and speed. The system Gitor is relevant to my thesis as

it utilizes developer patterns alongside code stylometry techniques to analyze code through local and global file elements. This technique provides no specification for developer coding conventions and supports exclusive Limited language ranges which hinders general application. Even though Gitor's approach does not fully handle multi-language code or a variety of programming methods, its scalable nature yields important results.

2.3 Summary of Key Findings

The study of code stylometry offers both solutions and challenges for machine learning-based author identification and code similarity detection. The inefficiencies caused by formatting code and minifying techniques reduce the efficacy of author recognition systems. When analyzing unformatted source code, analysis begins more easily since it preserves helpful programming stylistic markers, but code formatting and minification make it more difficult for models to recognize these styles. The current limitations of STEM research are related to the data sets and computer language they use. Although Java and C/C++ are the focus of the majority of code author identification efforts, there are still few technologies available for doing analysis over the wider coding range. Most research faces data restrictions because researchers analyze small or specialized datasets which decreases their capability to adapt to various coding environments combined with three or more programming teams. The research shows that combining static code analysis with dynamic analysis can enhance accuracy while assessing coding behavior. The evaluation approach requires analyzing both the written code as well as its operational behavior including memory attributes and CPU usage impact. Deeper understanding of programmers' tactics during cooperative projects is possible with hybrid analytical approaches. The study also looks into how stylometry might improve cybersecurity operations by tracking illegal activity on virtual discussion platforms and identifying dangerous code in cyberspace. In order to handle unresolved dataset issues, new techniques were developed, which increased the accuracy of these tools when analyzing challenging data sets. Because it necessitates analyzing large code samples and handling author identification complexity, the methodology continues to face difficulties. While code stylometry is still making progress, further study is required to handle language extension, dataset restrictions, and detecting techniques used by authors to conceal their identities. The performance and accuracy of these analytical tools in field applications will be improved by more research into various settings and integrated methodologies.

Chapter 3

Requirements, Impacts and Constraints

3.1 Economic Analysis

The economic analysis includes the system's overall financial feasibility and long-term sustainability. As the project is software-based, the main costs are on computational works and research time. No manufacturing or deployment was required for this. This section talks about whether the system can be managed at reasonable cost while still providing the intended benefits of clone detection and developer profiling.

Cost Components: The development and testing phases were conducted on a regular desktop computer with 16 GB of RAM which costs around BDT 120000. No specialized GPUs or high-end hardwares were needed for this. All frameworks and libraries that were used including Python, scikit-learn, pandas, and machine learning modules, were publicly available and free. Data was collected from GitHub and using these open-source resources helped us to avoid expensive licensing fees which would usually be required if we wanted to implement deep-learning methods. It also has low maintenance costs as maintenance mainly involves regular updates to Python dependencies and refreshing the pre-trained models when newer versions become available.

Benefit Analysis : In software development, code clones represent a hidden but significant cost. Managing duplicated codes requires a good amount of developer labour cost. When issues appear in the system, they must be fixed multiple times which increases both the time and cost of maintenance. Our system can identify code clones early which allows teams to identify unwanted duplications. Suppose a development team uses 20% of their maintenance budget on clone-related issues now if this cost can be minimised by even 15% then significant savings can be done. This can potentially save many work hours annually which can be used for other development works rather than on repetitive bug fixes. Additionally, our tool improves plagiarism detection and ensures fair evaluations which can be beneficial for educational institutions. It reduces code redundancy and the possibilities of copyright issues which can be useful in legal settings. Hence the benefit outweighs the costs.

In short, the system is fairly inexpensive to develop, easy to maintain, and useful in both academic and corporate fields, particularly within the Bangladeshi research

and software engineering community. Thus implementation of our framework makes long term economic sense.

Chapter 4

Proposed Methodology

4.1 Design Process

The system has two goals: profile developers from their coding style and detect code clones for plagiarism, reuse, or redundancy. All steps are kept simple, repeatable, and aligned with these goals.

Inputs: We start from a fixed sample of 10,000 GitHub users (from the Kaggle "GitHub Users" dataset) and collect public Python files from their repositories. We keep a Developer table (cleaned profile fields) and a Code table (Python files and features).

Preparation: Data is cleaned and standardized. We remove non-personal accounts, generated or vendored code, unreadable or tiny files, and read all files with a consistent encoding. Comments and whitespace are kept because they carry stylistic features. User text fields are treated as structure (presence and length), and numeric fields are checked and then scaled with StandardScaler.

Feature building: From metadata, we create stable signals such as account age, activity per year, and follower ratios. From code, we extract stylometry features: average line length, comment ratio, whitespace and indentation, identifier patterns, loop and conditional rates, import diversity, and light branch density. For clone work we tokenize, normalize identifiers, build k-gram fingerprints, and compute lightweight embeddings for cosine checks. Features are pruned to remove near-constant or duplicate variables; PCA/t-SNE are used only for plots.

Models: For developer profiling, unsupervised clustering is used. K-Means, Agglomerative and BIRCH are tested then we select Birch as the final model with k = 2 for the global step. We used a hybrid method: token-based hashing, local alignment algorithms, and semantic embeddings are combined to make the system which can detect Type-1, Type-2, Type-3 and Type-4 clones. The plagiarism threshold for clones is 0.80.

Evaluation: The Silhouette Score, Davies-Bouldin Index, and Calinski-Harabasz Score are used to find the quality of clustering with the use of PCA/t-SNE visualizations. The degree of duplication is indicated and clone identification is scored using cosine similarity and basic coverage metrics (%FILE and %LOC).

Output: The pipeline creates a per-file clone report with similarity scores. It matched similarities with a normalized per-developer feature matrix to cluster on. It is known

as cluster profiles which characterize common styles. Similar procedures can be used to handle new Python files. It can map them to an existing cluster or verify them while cloning.

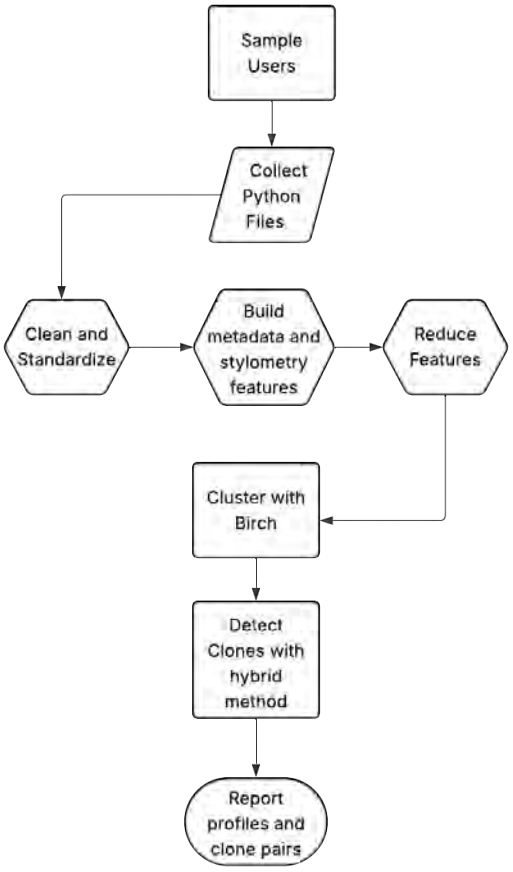


Figure 4.1: Design Process Summary

4.2 Model Specification

We identify and evaluate the models that will be used for code clone detection and developer profiling in this phase. The objective is to use stylometric analysis to find patterns in developer behaviour and to find duplicated or redundant code fragments that might be signs of plagiarism, inefficiency, or reusability. In order to accomplish this, we looked at a number of different approaches and compared them for verification. The following sections talk about it exactly.

4.2.1 Developer Profiling and Code Stylometry Analysis

In order to capture the innate structure of the data, cluster analysis is an effective method that separates the data into meaningful groupings, or clusters. The natural links or patterns found in the data should be reflected in these clusters. Hence, due to the nature of our problem, where it was important to find developer coding behavioral patterns, clustering was used. Only the Python files were used from the dataset to extract a collection of stylometric features including average line length,

comment density, identifier name patterns, indentation depth, and whitespace ratio. To maintain a fair comparison, all the models were initialised with the same number of clusters (k=2). Clustering was used to group source code based on these inherent stylistic similarities, and for this, three clustering approaches were explored:

1. **KMeans Clustering:** One study found that using K-means clustering, developers were divided into behavior groups for better practice and team balance [16]. Since it is effective in high-dimensional spaces, it was used with a fixed number of clusters (k=2).

2. **Agglomerative Clustering:** Its operation is done by recursively combining the nearest pair of clusters and creating a hierarchy of clusters. Even in sparse and high-dimensional spaces it was seen that this model achieved high silhouette score which indicates that it is effective in finding stylometric features similarities

3. **Birch:** With threshold=0.5 and k=2, Birch was able to construct a compact clustering tree gradually before employing a global clustering step to fine-tune the clustering. To preserve the entire stylometric feature space, the model's default branching factor of 50 was employed, and no dimensionality reduction was done. After thorough testing it was proven that Birch consistently produced the most compact and well-separated clusters across all evaluation measures and hence it was chosen.

Model	**Silhouette**	**CH**	**DB**
Birch	0.967	3002.820	0.243
KMeans	0.874	3747.823	1.148
Agglomerative	0.849	3476.434	1.174

Table 4.1: Comparison of clustering models using Silhouette, Calinski-Harabasz (CH), and Davies-Bouldin (DB) scores.

4.2.2 Clone detection framework

Before talking about our approach, it is important to understand the code clone classifications since different approaches require different detection methods. Clone classifications influence how tools are built and what reengineering procedures can be implemented. Gautam and Saini (2016) researches that there are three ways to study the composition of clone instances within the program[7]:

1. Similarity between two code segments: some clones have the same text, while others have the same functionality but a different syntax

2. Possibilities for refactoring: Some clones are better suited for code cleanup and reengineering and for that, it is important to identify redundant patterns

3. Location of clone instances: The position and function of clones within the program have an impact on refactoring and maintainability.

Software clones are often divided into four main categories:

- Type-1: Code fragments that are identical with the exception of differences in comments, layout and whitespace.
- Type-2: Identical code segments but differs in variables, types, or identifiers along with Type-1 clones differences.
- Type 3: Code that is structurally or syntactically identical with or without further changes; statements were added, altered, or deleted
- Type 4: Semantic clones, which differ syntactically yet carry out the same computation. They have similar functionalities

Mayrand et al. underlined that function names, code structure, lexical information, and control flow may all be used to evaluate how similar two clones are[1]. Various clone detection tools have been built based on these clone types, and we have explored three such tools:

1. **SoucererCC:** It is a widely used token-based clone detection tool that is designed to detect accurate and near-miss clones. It uses a quick and easy bag-of-tokens method that is resistant to Type-3 modifications to compare code blocks. For the first three clone kinds, it was discovered that this method offers good recall and precision[8]. It can scale up to 250MLOC on a single system. Its sensitivity to deeper semantic or structural alterations is constrained as it mostly depends on token overlap and thus it is particularly ineffective for Type-3 and Type-4 clones.

2. **CCLearner:** It is the first token-based clone detection method that uses deep learning. Deep learning enables CCLearner to efficiently detect clones in testing data by capturing the token usage patterns of clones in training data[11]. When compared to non-deep learning approaches, CCLearner demonstrated competitive accuracy in studies conducted over BigCloneBench. However, the design adds complexity in model training, relies on representative labeled data, and makes matching clones less interpretable.

3. **CodeCloneX:**Our tool uses a hybrid approach that is optimised to detect all clone types (Type1 to Type-4). It relies on tokenisation and k-gram hashing for syntactic fingerprinting and to detect Type-1 and Type-2 clones. It incorporates the Smith-Waterman alignment algorithm to efficiently detect near-miss clones i.e Type-3 clones. For Type-4 detection Sentence-BERT embeddings are implemented with cosine similarity. It can be successfully used in real-world applications for plagiarism detection and redundancy analysis. The design integrates both syntactic accuracy and lightweight semantic scoring to improve reliability while avoiding the computational cost of full deep learning methods. It can address near-miss clones since it uses a combined embedding-based similarity scoring with token-level hashing. These results led to the selection of our hybrid clone detection method combined with lexical stylometry profiling as the final CodeCloneX model specification.

Below we have shown the comparison between three clone detector tools in depth-

Table 4.2: Comparison of Clone Detectors

Aspect	**SourcererCC**	**CCLearner**	**CodeCloneX**
Detection Type	Tokenbased (frequency overlap)	Tokencategorybased (AST + token categories)	Token + Semantic (SentenceBERT embeddings)
Similarity Computation	Overlap ratio between token sets	Average of 8 categorywise similarities	Cosine similarity (SentenceBERT embeddings) + kgram hash overlap
Clone Pair Threshold	Token overlap $\geq 0.5 \times \max(\text{sizeA}, \text{sizeB})$	Mean similarity ≥ 0.5	Semantic similarity ≥ 0.80 (plagiarism threshold)
Output Metrics	sourcerercc_clones.csv, sourcerercc_metrics.csv	cclearner_clones.csv, cclearner_metrics.csv	clone_pairs.csv, coverage_by_file.csv
Performance Scope	Fast — uses index pruning	Moderate — full pairwise comparisons	Moderate — embedding generation per file
External Dependencies	None beyond `pandas`	`numpy`, `pandas`, `ast`	`sentence-transformers`, `scikitlearn`, `pandas`
Minimum Code Size Filter	10 tokens per block	10 LOC per method	8 tokens per file (for kgrams)
Strengths	Efficient, scalable for large datasets	Structurally aware (semantic categories)	Captures deep semantic similarity and plagiarism
Weaknesses	Misses semantic clones	Slower on large codebases, categoryweight bias	Computationally expensive, modeldependent
Intended Use	Largescale syntactic clone detection	Structural clone learning & analysis	Plagiarism and semantic code similarity detection

4.3 Dataset Overview

This dataset was sourced from Kaggle which consists of GitHub user profiles. It has around 150,000 records where each row represents one user account. The data was

narrowed by only including users who had at least 69 followers. Thus the dataset concentrates on active and notable developers rather than newly created or inactive accounts. We employ two interconnected components. A fixed snapshot of 10,000 GitHub user accounts from the "GitHub Users" dataset on Kaggle were used first. Each entry provides some user information, such as their GitHub username, number of followers, number of followings, and the number of public repositories and gists they have made. Some profile details are also given like location, company, bio or connected accounts (if it was provided by the individual), login, account age (based on timestamps). To enable anyone to recreate the identical sample, we stored the snapshot date and the precise list of user IDs

We gathered publicly available Python files from these persons' GitHub repositories. Only files with OSI-compatible licenses were retained during collection and went through data cleaning, transformation and integration phases which ensured the integrity and reliability of our final cleaned dataset.

The dataset is useful in studying developer patterns and their contributions. It can also be used to create recommendation systems that offer collaboration prospects and can help identify popular developers. However as the dataset only includes users with at least 69 followers, the dataset is skewed toward popular accounts and does not accurately represent most of the small developers. It also reflects a point in time which means that follower counts and repository numbers are not changing dynamically. Many fields may be incomplete due to many users not offering optional profile information such as a bio or location.

4.3.1 Data Collection

We used the Kaggle "GitHub Users" dataset as our starting list of developers and selected a fixed sample of 10,000 user accounts. We saved the snapshot date and the exact user IDs so anyone can rebuild the same sample later. From these users, we collected public Python files from their GitHub repositories while following repository licenses and GitHub's terms. During collection, we removed organization and bot accounts, skipped generated or vendored code (such as build folders or third-party libraries), and dropped unreadable or very small files. We kept comments and whitespace because they carry the writing style. Every file was read using the same line endings and encoding. Two aligned assets are produced as a result: a Code table that contains several Python files per user along with their extracted features, and a Developer table that contains cleaned profile fields (such as account age and repository counts). These logs and IDs are unambiguous. These are prepared for clone detection, grouping, and stylometry.

4.3.2 Data Cleaning

The Python code and Kaggle user metadata were the two databases we grabbed. Eliminating noise and non-conforming content was the idea. Additionally, keep features important for clone analysis and style.

Missing Values :

The profile text sections including location, company, and bio were frequently

inconsistent or blank. Meaning was not inferred by us. The neutral features, such as the field's presence and the character's length inside it, were kept. In the case of code-based functionalities, unreadable files were eliminated. Additionally, user-level summaries were calculated for files that passed initial testing.

Outlier management:

The number of repositories followed and followers varied from one individual to another. We made sure that very high values were real numbers and then normalized numerical properties so that large values did not dominate clustering. It stored strange and just cases and model compositions.

Screening and licensing:

We accepted Python files under the OSI-compatible licenses only. Generated or vendored content (migrations, compiled outputs, third-party trees, and so on) was written out using path patterns and header cues. This reduced corpus to that which expresses personal fashion and does not cross normative boundaries.

Resolution of inconsistency:

Usernames have been normalised in terms of case and encoding to avoid multiple identities. Organization accounts as well as clear bot accounts were being filtered in order to keep the focus to individual developers. The identified highly auto-formatted projects were found to be considered about sensitivity at a later stage.

Reproducibility Record:

We recorded the snapshot date to reproduce the clean sample. The verbatim list of 10,000 user IDs and recorded all the exclusion causes to allow the re-creation and audit of the clean sample.

This cleaning made the data ready for clustering, keeping coding and habit insights, though sparse features were a small problem.

4.3.3 Data Transformation

We thought of simple conservative modifications. We were behaving as true authors, avoiding the free-text speculation of semantics. We created robust features with the ability to cluster and detect clones.

Metadata Features: Account level variables were transformed to comparable signals such as account age in days, repositories/year (activity), follower-to-repository and follower-to-following ratios with simple treatment of zeros, and simple profile-completeness measures. Free-text scales were normalized to neutral counts (length of character, text contains a URL or email-like token). All numeric features were standardized using StandardScaler so that there was no dominant scale in the models.

Stylometry Characteristics: Each Python file has been encoded as a numeric vector with readily interpretable characteristics: line length average, comment count, whitespace ratio, indentation count, identifier casing regularities, loop rates, conditional rates, import variety, and a slim branch-density per 100 lines of code.

The developer lumped the file-level vectors into a single style profile that was stable, and then made them comparable on an equal footing.

Clone-Detection Preparation: Identifiers, numbers, strings, and other entities in the source files are replaced with generic placeholders that tokenized and normalized the files. We generated k-gram fingerprints to establish fast identification of Type-1 and Type-2 clones with the normalized tokens. And for type-3 clone detection we implemented Smith-waterman algorithm We also retrieved lightweight embeddings and used cosine similarity for Type-4 clones. The analysis, but not the transformation, uses a similarity cut-off of 0.80.

Quality Checks and Outputs: We ensured that there were no variables that narrowed to a constant and that the feature scale range was constant. This phase produced two products: a table of clean metadata characteristics per developer and a table of stylometry and fingerprints per file. They're both prepared to integrate.

4.3.4 Data Integration

Integration linked the cleaned Kaggle user records to their public Python code and produced the matrices used by clustering and clone detection.

- **Linking Strategy:** GitHub login was used to match repositories to users based on case-insensitive rules and best-effort management of visible account renames. A Developer table with cleansed metadata and a Code table with several Python files per user that had stylometry and fingerprint features were the two aligned assets that resulted from this.
- **Aggregation to Developer Level:** A consistent per-developer style profile was produced by aggregating file-level vectors using robust averages and medians. These profiles were then joined to the Developer table to form the matrix used for unsupervised profiling. Artifacts per file were stored to the clone detector and also to audit trails to allow inspection of individual comparisons.

This step generates a standard feature matrix (per-developer) to cluster (as used by the BIRCH model) and a set (per-file) to the hybrid clone detector.

4.3.5 Data Reduction

Data reduced the feature space, making it less volatile, small, and easy to analyze and no signal useful to style or clone identification was lost.

- **Pruning of features:** We dropped features which have next to zero variance and also dropped apparent collinearity among features with nearly exactly the same value. None of the basic families, or the whitespace and indent, naming patterns, density of control-flow, comments, or diversity of imports were altered in order to preserve the meaning.
- **De-correlation and Stability:** The winnowing performed on pairs that were extensively correlated so one and two had a new feature with distinct information. We retained those two variables which contain the same story but were easier to read and were more consistent across batches.

- **Dimensionality techniques:** PCA and t-SNE were used to plot clusters and outliers in 2Ds only. They were not using their outputs to clustering as inputs. Clustering models were applied using the standardized and pruned original features to ensure that results would be interpretab

- **Run-time and Host Memory Controls:** Standardisation and pruning enabled meaningfulness of distances and minimised the compute time. Down-sampling was applied in visualisation. Modelling was performed with the complete cleaned data.

- **Clone Pipeline Compression:** To detect a clone, raw token streams were substituted with compact k-gram fingerprints and stored in memory. This prevented repeated comparisons without sacrificing high sensitivity to exact and near-exact matches. To check a fast cosine, lightweight embeddings were stored in a fixed size. Local alignment algorithm was implemented as well.

- **Quality Checks:** Once we reduced, we ensured that every retained feature varied among users and that internal clustering scores showed improvement or did not change. The smaller space was consistent with the pruned-out structure.

At this level, a standardized, pruned, per-developer matrix of clustering and a compact per-file representation of the clone detector are produced, with well-understood rules and parameters such that the same inputs can be reconstructed.

4.3.6 Summary of Preprocessed Data

The resulting preprocessed data is a fixed 10,000-user sample of the Kaggle Users file plus a filtered Python-only code corpus of the users in their public repositories under the OSI license. Some standardized features we constructed based on the metadata included account-age, activity-density, follower-ratio, and profile-complete flags. Our generated per-file stylometry vectors (line length, comment density, whitespace and indentation, identifier patterns, control-flow rates, import diversity, light complexity, and the rest) were then aggregated into per-developer profiles and made everything fair in comparison. To do clone analysis, each file contains a small k-gram fingerprint for token-level matching, optimized with Smith-Waterman alignment for structural similarity, and paired with a light fingerprint embedding that is used to carry out cosine screening, and the decision threshold of 0.80 is used during the analysis. Reduction removed near-constant and redundant variables, kept interpretable feature families, and left PCA/t-SNE for plots only. All steps are recorded with a snapshot date, the exact 10,000 user IDs, and exclusion logs. The outputs are two aligned, analysis-ready artifacts: a standardized per-developer feature matrix for unsupervised profiling (used by BIRCH) and a compressed per-file representation for the hybrid clone detector, with audit trails and parameters to rerun the pipeline.

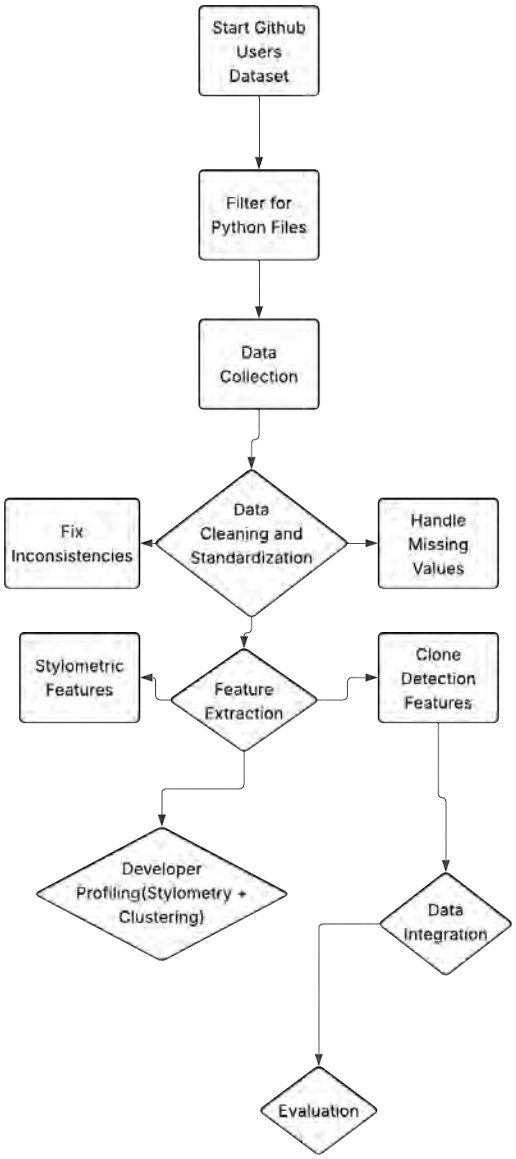


Figure 4.2: Preprocessing Summary

4.4 Implementation of Selected Design

Our final system of developer profiling adopts a completely unsupervised, clustering-based solution with token-based clone detection, thereby revealing any underlying stylistic patterns in programming behavior. The cleaned Kaggle dataset with developer metadata and repository details is loaded by the system first. Through GitHub login, each repository is associated with its respective developer by utilizing case-insensitive matching and best-effort account rename processing. After filtering the Python files, two aligned datasets are created: a Code table and a Developer table. Python's scikit-learn package was used to implement all models, with extra scripting added for feature parsing and batch evaluation.

For each Python code file from our dataset, a broad range of features, including total lines, average comments, loop rate, CPU/memory usage, variable counts, indentation

behavior, and error data, were extracted. In order to create stable profiles and preserve individual file-level fingerprints for verification and clone detection, file-level features are combined into per-developer vectors using reliable averages and medians.

StandardScaler was applied to these features in order to ensure that each dimension is scaled consistently to contribute equally to the clustering process. Although accuracy or separation measures are frequently used for theoretical evaluation, a model's real usability is dependent on how well it can handle high-dimensional data in large-scale systems. The selected design in our case was Birch, and it was initialised with a threshold of 0.5, which maintained the sub-clusters' radius while underfitting and overfitting of the model were balanced. The number of final global clusters (k=2) was adjusted following the construction of the first CF (clustering feature) tree. After a thorough empirical analysis, this configuration was selected as such that threshold=0.5 produced the optimal balance between inter-cluster distance and intra-cluster compactness. Birch constructed a compact tree structure of clustering features by incrementally scanning the dataset. The CF tree was constructed, and the leaf entries were subjected to a global clustering algorithm, which resulted in the final 2 groupings (since k=2). A cluster label was given to each occurrence in the dataset by converting numbering groups to readable formats. Relevant logical interpretations of the groupings were then obtained by comparing these labels to the averaged feature values.

Prediction in real-time can also be performed by entering a new code sample into the system with extracted features, then mapping a pre-trained cluster, and subsequently obtaining a style profile. To demonstrate the results of clustering, using PCA and t-SNE, the data was packaged so that groupings of developers in two dimensions can be visualised as well as determine instances of overlapping developer behavior or outliers.

Our clone detection framework was implemented in Python which focuses on making a hybrid system that can detect all 4 clone types. The first thing the system does is read input source files from a designated directory. It uses k-grams hashing for lexical fingerprinting. Source files are first tokenized and normalized by substituting generic symbols (such as id, num, and str) for identifiers, integers, and strings. Line by line, each file is tokenized, separating IDs, keywords, and symbols into unique tokens. This guarantees that structural similarities are preserved even when variables or function names differ, which minimizes noise brought on by formatting variations (such as whitespace, indentation, or comments). After tokenizing files, the program matches files based on lines. For every comparison, normalization of tokens is done to accommodate small syntactic variations. Compact fingerprints of the code are created by using k-gram hashing (SHA-1) to these normalized sequences. It is used to prevent unnecessary computations and to speed up repeated similarity checks. This is done to detect Type-1 and Type-2 clones. It uses the Smith-Waterman local alignment approach to identify Type-3 clones. While allowing for gaps, insertions, and substitutions, this technique aligns token sequences. Invalid matches are filtered away and a normalized similarity score is calculated to detect partial overlap using alignment statistics. The system employs Sentence-BERT embeddings (all-MiniLM-L6-v2) for Type-4 clones. Cosine similarity is calculated for each file pair contained in a semantic vector space.

The flagging of two code fragments as clones is based on a plagiarism threshold which

is set at 0.80 and it can be interpreted such that:

- A high ratio (greater than 0.95) corresponds to Type-1 clones with whitespace and comments being the only differences.
- A range over the threshold but less than 0.95 (0.85 to 0.95) corresponds to a Type-2 clone (same structure but with a different identifier).
- A moderate ratio (from 0.70 to 0.85) corresponds to Type-3 clones. Strong Alignment of Smith-Waterman also means Type-3 clones.
- Score greater than or equals to 0.60 (0.60) corresponds to Type-4 clones (semantic clones)

File pairs that have already been compared are skipped to prevent unnecessary computation. Additionally, large collections of Python files can be analyzed due to this design's low runtime, and it can maintain traceability to the original developer, which is useful for audit purposes.

Chapter 5

Result Analysis

5.1 Performance Evaluation

For this study, we have only chosen Python codes, so these patterns studied can be language-specific. Cross-validation was applied to assess model generalizability, as it is a common technique used in machine learning. The metrics were chosen to provide a balanced and fair analysis of the models used. The three models' performance was assessed using evaluation metrics, which include:

- Silhouette Score
- Davies-Bouldin Index (DBI)
- Calinski-Harabasz Score.

Silhouette Score was used as an internal evaluation measure, which evaluates cluster separation and cohesion without the need of ground-truth labels. It ranges from -1 to 1, where a higher score indicates the presence of better-defined clusters. It is particularly helpful in the early stages of model selection, as it can determine which model generated the most compact and distinct groupings. However, it may not perform well with clusters that have varying shapes and densities. A score of about 1 means that a point is distant to other groups and fits very well in its group i.e. cluster. A score near zero means that the point is on the boundary between two clusters. The point may be in the incorrect cluster if the score is near -1 and a score closer to 1 is preferable.

Davies-Bouldin Index (DBI) can be used to assess a dataset's clustering quality. It examines the compactness (tightness) and separation (distance) between each cluster. A lower DBI means better, more distinct clusters, whereas a higher DBI means overlapping, disorganized clusters. A lower score is preferable.

Calinski-Harabasz Index accounts for the quality of a dataset's clusters. It looks at the points and measures how close the points inside each cluster are. What is the distance between the clusters? A higher score is better as it indicates the clusters are tight and far apart. It helps in figuring out the optimal number of clusters.

The ability of a clone detection tool to reliably and effectively identify clone pairs across various codebases is a sign of its effectiveness. For this, we considered some evaluation metrics, which include:

- Similarity Scores
- Clone Coverage

Cosine similarity computes the cosine of the angle between two non-zero vectors to determine how similar they are. It is frequently used in data analysis and machine learning, particularly in recommendation systems, text analysis, document comparison and search queries. It can be used to compare two sentences in Python as the Cosine Similarity function allows us to calculate how similar two words are to one another. As a result, the framework can identify clones that have the same functionality but different syntactical structures. Higher similarity indicates smaller distance.

- Strong semantic similarity is indicated by a value near 1.0
- Zero Semantic similarity is indicated by a value around 0.0.

To improve clone classification detection a fixed plagiarism threshold of 0.80 was set which minimizes false positives while still ensuring high sensitivity. With the help of the threshold, the program can convert raw similarity scores into labels for grouping clones. Clone pairs that the system correctly recognizes are known as true positive clones. False Positives are pairs that are mistakenly identified as clones. Lastly, missed clone pairs that should have already been identified are known as false negatives. The system's accuracy is maintained by ensuring that all these False positives and negatives are examined along with true clone pairs.

Two indicators based on coverage were taken into consideration:

- %FILE: The percentage of source files that contain at least one identified clone segment.
- %LOC: The percentage of total lines of code that are part of at least one clone instance.

A low %LOC but high %FILE means that many files contain small duplicated sections.

A high %LOC and high %FILE imply significant redundancy, which often calls for refactoring.

A high %LOC but low %FILE indicates that few files contain significant duplicated code blocks.

These metrics help in evaluating the overall effect of cloning on software maintenance and plagiarism detection, as well as code redundancy level.

5.2 Analysis of Design Solutions

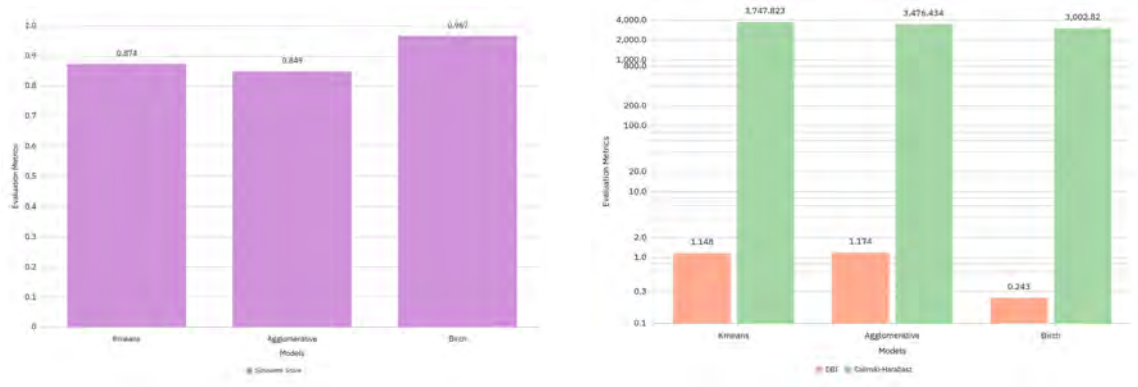
In our research, we investigated unsupervised learning following the processing of our data. The design deals with stylometric feature extraction and developer profiling, and token-based clone detection. The three algorithms- KMeans, Agglomerative Clustering and Birch were evaluated using the evaluation metrics with a view to

identifying the developer-specific coding patterns on the basis of stylometric data. This section discusses the strengths and weaknesses of each approach as well as other factors such as accuracy, efficiency, feasibility and overall performance.

K-Means: It is easy to understand and apply, and since it is widely used in high-dimensional data applications and is very scalable it is often chosen in modeling complex developer behaviors. After running this algorithm it was seen that the silhouette score came high 87.40% (0.874) which means that it is similar to its own cluster compared to the neighbouring groups. The Calinski-Harabasz Score (3747.823) and Davies-Bouldin Score (1.148) shows that K-means formed well-defined clusters but not as distinct and compact as Birch. K means can be taken in consideration if computational speed is prioritised. However, it makes the assumption that clusters are uniformly sized and spherical, and it requires predetermining the number of clusters which may not always align with finding developer styles.

Agglomerative Clustering: It is able to describe complex relationships by creating a hierarchy of clusters based on pairwise distance and it does so without making any assumptions about the cluster structure or shape beforehand. Although it has scalability issues with very large datasets it can be effective in presenting subtle stylistic changes in codes. The majority of agglomerative algorithms are a liability in terms of computational and storage needs[9].It performed well with a silhouette score of 84.90% (0.849) and Calinski-Harabasz Score of 3476.434 and a low DBI of 1.174 which indicates that they created clusters that were mostly not compact and overlapping despite its hierarchical structure.

Birch (Balanced Iterative Reducing and Clustering using Hierarchies): It can scale well with high-dimensional data and is especially effective with huge datasets. It can form compact, well-separated clusters with little overlap due to its hierarchical method. The Birch tackles two major clustering analysis problems: it can scale when handling large amounts of data, and its robustness to noise and outliers[2]. It obtained the best Davies-Bouldin Score (0.243), the greatest Silhouette Score of 96.70%(0.967), and a high Calinski-Harabasz Score (3002.820). Thus, the clusters are internally cohesive, compact and very well-separated which shows that the stylometric features of developers do not overlap and form separate groupings. Even though the Calinski-Harabasz Score (3002.820) is lower than Agglomerative Clustering and K-means Calinski-Harabasz Score, overall Birch's performance is better. Hence it was chosen to be the best approach.



(a) Comparison of Silhouette Score

(b) Calinski–Harabasz vs DBI

Figure 5.1: All models clustering metrics.

While exploring the clone detection tools, it was found that:

SourcererCC: Although token-based indexing makes SourcererCC efficient, it does not perform well on very big datasets. It is scalable and achieves high precision for Type-1 and Type-2 clones but it shows poor performance with Type-3 or Type-4 clones. It was specifically designed for Type-1 clones. It is also limited to syntactic similarity and could result in false positives when identifiers are slightly different. SourcererCC shows low LOC% (52%) but high File% (approx 86%). This means that it identifies a large number of files containing clones, but the percentage of cloned lines in those files is quite small. Thus it achieves high file coverage but limited detection.

CCLearner: It demonstrates better performance with Type-3 and Type-4 clones by utilizing a learning framework to extract semantic similarities. Compared to other clone detection tools it has the highest time cost i.e very poor runtime [8]. It is not productive with large scale repositories and since it requires a large size of training data it has a high computational cost. In the case of semantic clones, CCLearner performs well in recall, but noise affects precision. CClearner has a very high LOC% of 153% (approx) which suggests strong semantic detection that can identify overlapping or several clone dependencies per line which may result in insufficient coverage. Its File% is fairly low (57%), indicating that it finds fewer files containing clones than SourcererCC but it provides better coverage of those files. This shows CClearner's strong ability in semantic (Type-4) detection, however it may have the potential of over-coverage or false positives.

CodeCloneX: It is a hybrid tool that combines semantic embedding, lexical fingerprinting and alignment algorithm to effectively detect all clone types. This tool has an 80 percent similarity threshold to ensure that it has a good accuracy rate and fewer false positives compared to other token-based methods. One characteristic that sets it apart from other clone detectors is the usage of Smith-Waterman algorithm which makes the tool resistant to changes in the code, including insertions, deletions, and rearranged statements. This is especially useful in detecting Type-3 clones. Since this approach uses Smith-Waterman alignment and semantic embeddings it has an increased runtime in comparison to solely token-based techniques like SourcererCC. For huge codebases with millions of files, this might restrict scalability. Semantic detection i.e Type-4 detection is made possible through the usage of Sentence-BERT

embeddings. This is very easy to deploy since this model does not rely on training on labeled data.

CodeCloneX performs well with a high LOC% (approximately 90%) and a moderate File% (almost 57%). So it can detect a good fraction of duplicated lines and a moderate more than half of the files include clones. This shows that it is good at detecting line-specific redundancy while maintaining complete file coverage.

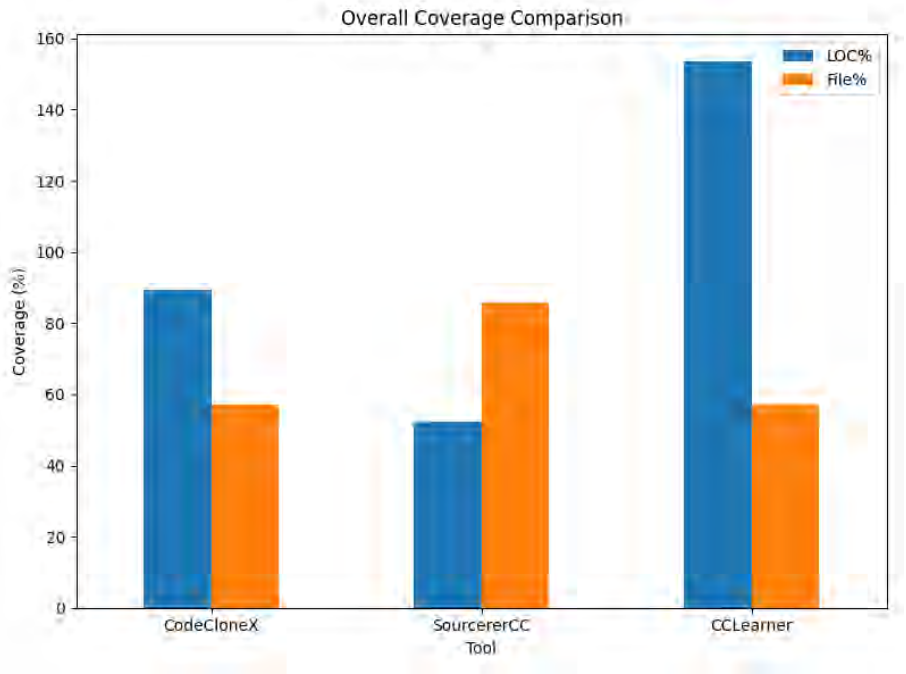


Figure 5.2: LOC% Coverage and File% Coverage of the tools

5.3 Statistical Analysis

In order to have reliability and consistency of our results, it was decided to conduct a serious statistical analysis of all our clustering models and assessment measures. It is essential to make sure that differences in the model performance were not caused by random errors or an imbalanced sample. In this way we can check the validity of our approaches by statistical analysis. A variety of statistical analysis methods already discussed also helped to verify the dataset. Principal Component Analysis (PCA) was used to maintain dimensionality consistency and outlier detection was done after finding the z-score (Standard Deviation Method) and points greater than 3σ from the mean were removed. The efficacy of three clustering models—KMeans, Agglomerative Clustering, and Birch was compared using the Friedman test followed by the Nemenyi post-hoc. Friedman test is a non-parametric test where the data might not be regularly distributed. For every model, metric values (Silhouette score, DBI and Calinski-Harabasz Score) were gathered. Every model was ranked according to a metric. The Python scipy.stats library was used to calculate the Friedman test statistic and p-value. A $p - value < 0.05$ was obtained from the test, which suggests that at least one model performs substantially differently from the others.

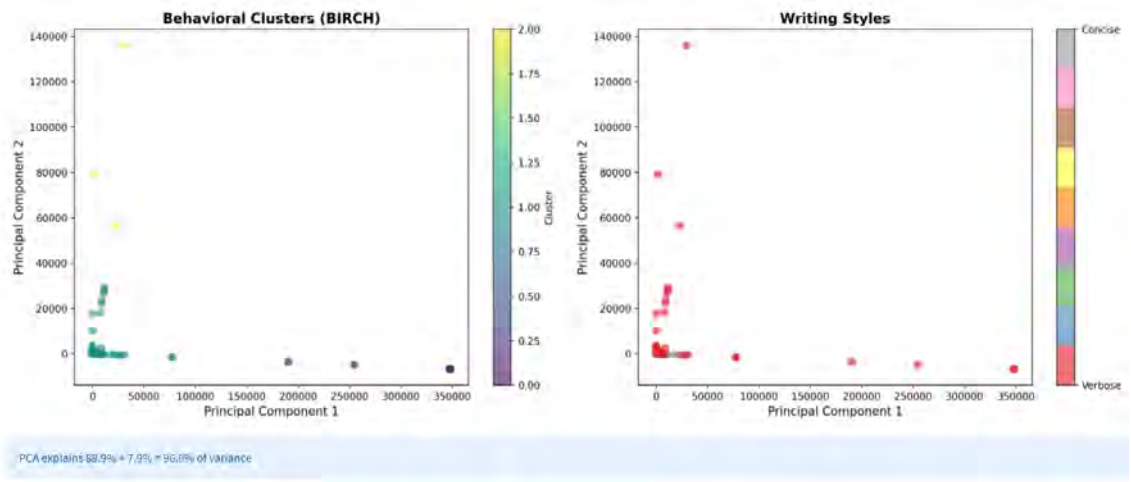


Figure 5.3: Principal Component Analysis

Thus, the null hypothesis that all models function similarly was thereby disproved. Now to identify which particular models were significantly different, the Nemenyi post-hoc test was used. This test assesses model differences pairwise and calculates a critical difference (CD) value to determine whether the performance difference between any two models surpassed the threshold for statistical significance. In terms of all measures (Silhouette score, DBI and Calinski-Harabasz Score), Birch performed noticeably better than Agglomerative Clustering and K-means.

For the clone detection part, comparative and descriptive statistical studies were performed on the similarity scores produced from Smith–Waterman algorithm and cosine similarity. The accuracy and consistency of CodeCloneX in detecting Type-1 and Type-2 code clones were assessed using continuous Token similarity ratios from k-gram hashing.

The alignment statistics from Smith-Waterman algorithm records: aligned length, matches, mismatches, gaps, gap ratio and match ratio A normalized similarity score is calculated by adding all these values together. Alignment statistics are a better option rather than simple overlap measurements for identifying near-miss clones in which pieces are slightly altered or rearranged since they can capture partial structural similarities.

In order to confirm the threshold (0.80) is correct for our design the mean, median, and standard deviation of similarity scores between known clone and non-clone pairs were computed. While non-clone pairs frequently had average low scores typically below 0.50 across all metrics, Type-1 and Type-2 clone pairs regularly had higher average scores (usually above 0.85). Alignment statistics scores for Type-3 clones averaged between 0.70 to 0.80 and Cosine similarity score for Type-4 clones was usually above 0.65. Thus the reliability of the 0.80 threshold for clone pair classification was validated. Using z-score normalization, similarity values were adjusted to reduce bias brought on by variations in file size or token count, which ensured fair comparisons across varying file sizes and complexities.

Table 5.1: Summary of Statistical Parameters Used

Statistical Method	**Application**	**Parameter Used**	**Threshold / Criteria**	**Purpose**
Principal Component Analysis (PCA)	Dimensionality reduction and consistency maintenance	Principal components	Based on explained variance	To maintain dataset consistency and reduce dimensionality
Z-score (Standard Deviation Method)	Outlier detection	Standard deviation (σ)	Points greater than 3σ from mean removed	To eliminate extreme outliers from dataset
Friedman Test	Model performance comparison	Test statistic and p-value	$p < 0.05$ indicates significant difference	To verify if models perform differently
Nemenyi Post-hoc Test	Pairwise comparison between models	Critical Difference (CD) value	CD threshold exceeded	To identify which model pairs differ significantly
Similarity Threshold	Clone pair classification	Similarity score	0.80	To classify code pairs as clone or non-clone
Z-score Normalization	Adjust similarity values	Mean, SD	Normalized across datasets	To reduce bias caused by varying file sizes or token counts

5.4 Comparative Analysis

This section talks about our approach and provides a comparative discussion with previous existing works. There were many studies conducted on developer profiling, stylometry, and clone detection but separately. There were not many works found that used a combined approach like ours.

Although previous stylometry research has investigated both supervised and unsupervised methods, the clustering strategy used in this work was determined by the characteristics of our data. Previous studies have shown that supervised classification may be highly accurate. For instance, a study done to investigate the impact of developer profiles on predicting vulnerabilities in software used Random Forest and other models, such as SVM, in their supervised learning approach, which showed noteworthy success [18]. These techniques, however, rely significantly on labeled data, which isn't always scalable or accessible in widely available open-source datasets. Even though this problem can be solved using classification, clustering was a better strategy because our objective was to create a scalable system for developer profiling. During preprocessing, the Kaggle GitHub dataset was cleaned and normalized.

Important stylometric characteristics were extracted, including not only syntactic properties, i.e., use of loops, conditionals, and a variable naming convention, but also computer performance characteristics like CPU time and memory usage. The quality of this preprocessing was statistically tested and ensured that no stylistic cues were lost due to noise or scale imbalance. Then we ran our three clustering methods: KMeans, Agglomerative Clustering, and Birch. Among the models Birch continuously outperformed the other examined models with the highest silhouette score. K-means was partially successful and worked well enough. A similar research based on developer profiles was done using clustering to group developer coding patterns and they showed successful results [16].

Our goal was to learn how various users code, and our cluster-naming system does precisely that. After forming a cluster, we determine the average feature-values of each cluster and then generate descriptive, interpretable labels that can describe the key characteristics of the cluster. An example is a cluster named concise writers, which characterizes those who produce small and simpler codes with little to no comments. Such archetypal labels are particularly helpful to educators, reviewers or team leaders who would like to learn about developer behavior in a glance. The naming task fills the gap between the rough numerical outcome of clustering and qualitative descriptions that can be used in making decisions. Moreover, our system allows continuous profiling of developers due to the ability to map incoming codes of interest to the previously generated clusters without retraining. This allows a real-time analysis of continuous data as soon as new data is available. Our solution aligns closely with the study of Hassan et al. (2017), which showed that stylometric clustering, using such characteristics as lexical and syntactic features, has the potential to attribute authorship and reveal the behavioral patterns without any prior classifications. Their study supports the efficiency in the combination of structural analysis and unsupervised models of behavior profiling[10].

Clone detection approaches were studied by Fu et al. (2024), and their own approach was compared. It showed that existing tools for clone detection, like SourcererCC, uses only tokenisation, which makes it less efficient than new hybrid methods similar to ours[15]. However, the study states that deep learning methods are most useful, but due to the computational cost, that was not implemented. One study on plagiarism detection methods done for large repositories by Burrows et al (2004) has been proven to be very effective[4]. But our thesis not just focuses on detecting plagiarism but describes developer style through clustering, providing a more behavioral viewpoint. The stylometry provides details about the similarities and differences between the code and does not only tell us if it is plagiarized. A study by Franrzeskou et al (2006) is done to identify source code authors using a method called SCAP [3]. Our stylometric features capture more macro/structural signals (comments, loops, indentation, function complexity), while SCAP features more micro-level stylistic signals, which are byte-level, n-gram style. It may be possible to achieve even better results by combining the two types (both for profiling and categorization).

5.5 Discussions

Some assumptions about the data are made by the clustering technique that was chosen. Identical groups of developers that share some common characteristics but

differ greatly from other groups were formed. The algorithm chosen was Birch, which does not assume equal variances among features or spherical clusters like the K-means. Rather, it allows for flexible, irregular cluster sizes, which makes it suitable for high-dimensional, sparse datasets, such as those produced from stylometric characteristics. After running the algorithms, two separate groups were generated by the model using stylometric and performance-related features- Cluster 0 and Cluster 1. The resulting clusters are statistically valid and could be interpreted semantically.

Cluster 0 corresponds to "concise writers," which features codes with shorter lines, fewer comments, compact functions, and fewer keywords. This cluster's developers often write simpler, less complex, and more functional code with fewer comments often with minimal documentation Cluster 1 corresponds to "verbose writers" that prioritized readability and documentation and included codes with lengthier functions, larger comment-to-code ratios, and a wider range of tokens. This represents elaborate coding techniques that use a large amount of conditional logic, loops, multiple variables, and greater line lengths.

When a demo code is given, the model can successfully identify the demo code's assignment to its Cluster and t-SNE is used to visualise this. For instance, you have provided two code snippets to our tool- Code A and Code B. The tool then checks if the code snippets are clones and if they are a clone pair the most likely developer or coder is associated with it by combining their stylometric features. If they are not clones, then each code snippet is profiled separately and mapped to potential developers. As a result, the program can effectively assign unknown codes to probable authors (within the dataset).

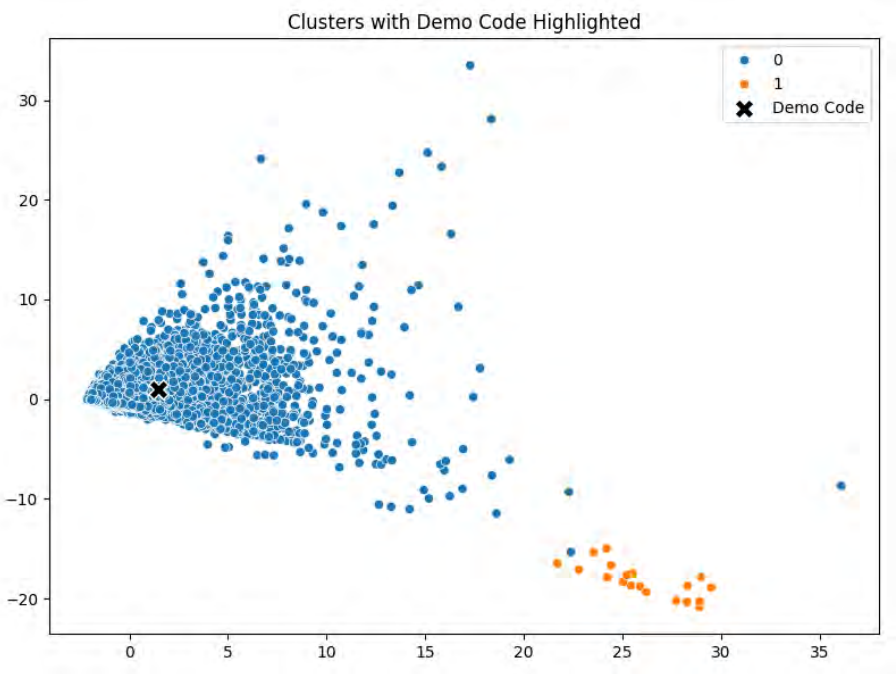


Figure 5.4: tsne

That being said, there are some limitations to our approach. These include dependence on feature quality since the quality and expressiveness of the collected features have a significant impact on the clustering results. A Sentence-BERT model that

has been pre-trained is used for semantic detection of Type-4 clones and while it is useful for general-purpose embeddings, it might miss very domain-specific coding semantics.

Also, very short or highly encrypted code snippets may lessen the accuracy of clone identification and developer attribution. Although it was suitable for preliminary study, using a fixed number of clusters, k=2 may have oversimplified the variety of developer styles.

Chapter 6

Conclusion

6.1 Summary of Findings

This paper intends to investigate the profiling of developers based on a stylometry approach to the code research problem with the help of unsupervised machine learning. After going through the preprocessing procedure, namely cleaning, feature engineering, and normalization based on the above-mentioned data reduction methods, the resulting filtered dataset can be characterized as stylometrically equivalent. A total of 19 attributes were kept, which successfully reflected structural and performance-based properties of code (e.g. number of lines, comment density, the frequency of loops, CPU/memory usage). The study thus shows that unsupervised learning may be used to profile developer actions and that unique coding styles can be meaningfully represented by stylometric patterns obtained from code.

Three clustering algorithms, KMeans, Agglomerative Clustering and Birch were used with their comparison on the basis of the internal validation measures, Silhouette Score, Calinski-Harabasz Index and Davies-Bouldin Score. Out of the above, Birch achieved the best silhouette of about 0.967, thus being the best method of clustering due to its visualization of a high silhouette score and the lowest Davies-Bouldin score (almost close to 0) in the model comparison graph.

Clusters were also semantically labeled (e.g. Concise and Verbose writers), by reading the mean feature of every cluster. This makes clustering more interpretable by practitioners. Furthermore, the system allows real-time profiling of the new code samples through mapping the code samples to predefined clusters. PCA and t-SNE provided two dimensional visualizations that gave an understanding about developer behavior and cluster separability.

Moreover, for the clone detection part a combined approach was adopted to identify the clones. K-gram analysis (SHA-1) with lexical fingerprinting was used to identify approximate matches. This simplified the process of determining Type-1 and Type-2 clones. It can also detect Type-3 clones by using the Smith-Waterman alignment algorithm. For detecting Type-4 clones semantic embedding with cosine similarity was implemented using Sentence-BERT embeddings.

Although there are some challenges, the main aim is to provide practical solutions that will enhance understanding of developer contributions, improve coding processes and contribute to the development of software engineering by producing impressive, maintainable software in the varied agile development settings. The results indicate

how developer profiling, stylometry, and clone detection may open up the possibilities of developing better tools and processes that highlights the importance of integrating contextual knowledge with analytical abilities.

6.2 Contributions to the Field

This study aims to create new tools and methods to address challenges in modern software development where in this field, groups of engineers often need to collaborate on complex, large-scale projects. The main focus is on improving stylometry to identify developers' styles and finding better ways to discover code reuse and redundancy. This approach helps keep the code clean while avoiding plagiarism.

In this paper, we present a code stylometry-based approach to developer profiling and clone detection. This method is significant for software engineering as the entire pipeline serves as a consistent guide for behavior code clustering. It starts with processing raw information and continues through interpreting the clusters. This approach can be adapted for applications like team profiling, plagiarism detection, anomaly detection, and onboarding assessment. Team leaders, human resource managers and collaborative developers can find it useful.

We compared various clustering models, including KMeans, Agglomerative, and Birch. The effectiveness of different algorithms for analyzing developers on a broad scale was studied. Code clone detection tools were discussed and the most accurate and practical solution was chosen i.e our tool called CodeCloneX. The aim of this study is to explore the relationship between these methods and how a combined approach can contribute to the advancements in software development. Our tool is not just a mere plagiarism detector as it also focuses on analyzing the stylometric features and behavior patterns of developers.

In summary, these contributions can improve software engineering by focusing on developer behavior, style, and individuality instead of just code accuracy. This creates new research and application areas in software forensics, education, developer analytics, and collaborative tools.

6.3 Recommendations for Future Work

A few recommendations for further research can help expand on the framework established by this study. Examining developer coding patterns in private repositories and a wider range of programming languages would provide a broader perspective. Applying machine learning, deep learning techniques, and artificial intelligence may lead to better results in improving the accuracy of profile and clone detection methods, especially in complex or multicoding environments. Additionally, investigating how to integrate these tools into real development processes and measuring their impact on team dynamics and project outcomes could offer valuable verification of their effectiveness. The current methodology uses a fixed number of clusters; in the future, we might have access to dynamic clustering methods that can determine the optimal number of groups. Finally, using stylometry to detect code reuse or plagiarism along with code clone detection in academic and commercial settings could reveal important insights. Further study in this area could help overcome current limitations and

create improved algorithms, ultimately enhancing software engineering and making the results more accurate and relevant.

Bibliography

[1] Mayrand, Leblanc, and Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *1996 Proceedings of International Conference on Software Maintenance*, IEEE, 1996, pp. 244–253.

[2] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," *ACM sigmod record*, vol. 25, no. 2, pp. 103–114, 1996.

[3] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, "Effective identification of source code authors using byte-level information," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 893–896.

[4] S. Burrows, S. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software: Practice and Experience*, vol. 37, no. 2, pp. 151–175, 2007.

[5] A. Caliskan-Islam et al., "De-anonymizing programmers via code stylometry," in *24th USENIX security symposium (USENIX Security 15)*, 2015, pp. 255–270.

[6] D. Rattan, R. Bhatia, and M. Singh, "Detecting high level similarities in source code and beyond," *International Journal of Energy, Information and Communications*, vol. 6, no. 2, pp. 1–16, 2015.

[7] P. Gautam and H. Saini, "Various code clone detection techniques and tools: A comprehensive survey," in *International Conference on Smart Trends for Information Technology and Computer Communications*, Springer, 2016, pp. 655–667.

[8] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 1157–1168.

[9] K. Bindra and A. Mishra, "A detailed study of clustering algorithms," in *2017 6th international conference on reliability, infocom technologies and optimization (trends and future directions)(ICRITO)*, IEEE, 2017, pp. 371–376.

[10] S.-U. Hassan, M. Imran, T. Iftikhar, I. Safder, and M. Shabbir, "Deep stylometry and lexical & syntactic features based author attribution on plos digital repository," in *International conference on Asian digital libraries*, Springer, 2017, pp. 119–127.

[11] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*, IEEE, 2017, pp. 249–260.

[12] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 101–114.

[13] N. Wang, S. Ji, and T. Wang, "Integration of static and dynamic code stylometry analysis for programmer de-anonymization," in *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, 2018, pp. 74–84.

[14] S. A. P. Sarnot, S. Rinke, R. Raimalwalla, R. Joshi, R. Khengare, and P. Goel, "Snapcode-a snapshot based approach to code stylometry," in *2019 International Conference on Information Technology (ICIT)*, IEEE, 2019, pp. 337–341.

[15] M. Fu, G. Luo, X. Zheng, T. Zhang, D. Yu, and M. Kim, "An ensemble learning approach for software semantic clone detection," *arXiv preprint arXiv:2010.04336*, 2020.

[16] C. A. González, L. A. Zumel, J. A. Acero, V. Lenarduzzi, S. Martínez-Fernández, and S. R. Rodríguez, "A preliminary investigation of developer profiles based on their activities and code quality: Who does what?" In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, IEEE, 2021, pp. 938–945.

[17] H. Zhang and K. Sakurai, "A survey of software clone detection from security perspective," *IEEE Access*, vol. 9, pp. 48 157–48 173, 2021.

[18] T. Coskun, R. Halepmollasi, K. Hanifi, R. F. Fouladi, P. C. De Cnudde, and A. Tosun, "Profiling developers to predict vulnerable code changes," in *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 32–41.

[19] M. Tereszkowski-Kaminski, S. Pastrana, J. Blasco, G. Suarez-Tangil, et al., "Towards improving code stylometry analysis in underground forums," in *Proceedings on Privacy Enhancing Technologies (PETS)*, 2022.

[20] M. M. Iqbal, A. Raza, M. M. Aslam, M. Farhan, and S. Yaseen, "A stylometric fingerprinting method for author identification using machine learning," *Technical Journal*, vol. 28, no. 01, pp. 28–35, 2023.

[21] J. Shan, S. Dou, Y. Wu, H. Wu, and Y. Liu, "Gitor: Scalable code clone detection by building global sample graph," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 784–795.

[22] S. Yang et al., "Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–40, 2023.

[23] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," *Journal of Systems and Software*, p. 111 796, 2023.

[24] S. Balla, M. Gabbrielli, and S. Zacchiroli, "Code stylometry vs formatting and minification," *PeerJ Computer Science*, vol. 10, e2142, 2024.

[25] A. Gurioli, M. Gabbrielli, and S. Zacchiroli, "Stylometry for real-world expert coders: A zero-shot approach," *PeerJ Computer Science*, vol. 10, e2429, 2024.

[26] M. T. Zamir, M. A. Ayub, A. Gul, N. Ahmad, and K. Ahmad, "Stylometry analysis of multi-authored documents for authorship and author style change detection," *arXiv preprint arXiv:2401.06752*, 2024.