Learning Developer Style: A Stylometric Framework for Attribution, Detection, and Profiling

by

Mohammad Sayed Safi Sams
21301704
Sudipta Nandi
21301534
Nabil Al Hami
21301512

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:

Mohammad Sayed Safi Sams

21301704

Sudipta Nandi

21301534

Nabil Al Hami

21301512

Approval

The thesis titled "Code Stylometry" submitted by

1. Mohammad Sayed Safi Sams (21301704)
2. Sudipta Nandi (21301534)
3. Nabil Al Hami (21301512)

of Fall, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on October 10, 2025.

Examining Committee:

Supervisor:
(Member)

Mr. Md. Aquib Azmain
Lecturer
Computer Science and Engineering Department
BRAC University

Thesis Coordinator:
(Member)

Dr. Md. Golam Rabiul Alam
Professor
Computer Science and Engineering Department
BRAC University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD
Chairperson and Associate Professor
Computer Science and Engineering Department
BRAC University

Abstract

Code Stylometry applied to source code is the task of revealing the author or source of a source code file, based on style and structure. Authorship verification has grown to be of great significance with the emergence of the open-source code and issues of plagiarism. This paper gives us a composite research on code stylometry consisting of extensive experimentation on more than 3.2 million Python source code files. To make upstream features well-suited for style tasks, we build a pipeline which extracts both low-level (e.g. token-level, syntax) and high-level (e.g. structural, behavioral) styles. Putting the code snippets in groups according to their author, we will collect a set of more than 44,000 different author styles. We use and test a number of machine learning and deep learning models to predict metadata and notice how stylometry features correlate with metadata. Our pipeline will analyze the coding style of the author in every code and the outcome proves that authorial fingerprinting is attainable with immense precision concerning varying pieces of code. In addition, we evaluated the similarity of coding style among authors by using the stylometric features which were provided by our pipeline. Our model offers a modular system on which future tasks, like plagiarism detection, authorship attribution or code-generation/mimicking based on a style can be built upon. This contribution provides a solid methodology, a large stylometric datasets, and high-quality baselines, which will enable future line of research on in-secure-code forensics and author-independent intelligent code-generators.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The study of identifying, examining and comparing the coding patterns used in certain code files or even snippets of a code is called code stylometry. Code stylometry is used in various usages like finding out the author of a code or part of a code, learning style and pattern of a programmer or coder, finding plagiarism in a code i.e, author fingerprinting, detecting malicious codes or malwares for security reasons and finding out the identity of an organization or an individual behind the code. According to Oman & Cook (1989) [1], code authorship came from forensics applications and is important in tackling issues like deanonymization and is used in fields like forensics applications and plagiarism. There have been a few numbers of methods used in code stylometry like static analysis, natural language processing (NLP) and graph analysis, but for the most part, the most use method in this study is machine learning. There are a few challenges faced in code stylometry, namely on how it can be difficult to trace a coder in a collaborative work, or how in different IDEs, there are default patterns and not any style or pattern used by an individual coder or programmer, standardizing code patterns and thus making it difficult to trace the author.

1.2 Rational of the Study or Motivation

Code stylometry has become an essential forensic tool in software development because programming has become more complicated and widely adopted. Knowing the author of a programming segment enables authorities to handle problems that include intellectual property infringement and software vulnerabilities with academic honor violations. The techniques for determining authorship struggle because programmers use code camouflage techniques and work together in programming groups while programming styles constantly change. Recent developments in machine learning, along with deep learning have substantially improved both accuracy and operational speed in code stylometry. The research study aims to enhance code stylometry by evaluating datasets through machine learning model assessments and Large Language Model (LLM) investigations which focus on accuracy improvement. The research addresses current challenges to develop an accurate identification system that improves security measures while improving both legal and ethical functions

for software security.

1.3 Problem Statement

In code stylometry, it is necessary to use Large Language Models (LLMs) to scan through various codes in a dataset or a databank, as there are a large number of codes in these datasets. Finding similarity between or among codes can be as challenging as finding a needle in a haystack, so the machine learning model has to be quite advanced to find similarities in codes or literature in a quicker amount of time with remarkable accuracy. Then there is the challenge for optimization of the machine learning model. Machine learning models can face specific challenges or shortcoming in a specific stylometric tasks. Training LLMs or any machine learning models or combining techniques in a way to achieve accuracy in detecting an author while optimizing the performance can be quite challenging. There are datasets like Github, Google Jam, OJClone etc to find codes from. This study answers these questions:

RQ1: Which datasets are effective in finding similarities in codes, thus helping in detecting authors of a code or a code snippet?

RQ2: Which technique (a single Machine Learning Model or combination of various models) would be the most effective in code stylometry?

RQ3: Which are the shortcomings for the given techniques that are going to be explored when it comes to code stylometry?

1.4 Objective

The main purpose of this research focuses on developing and studying code stylometry methods used for author identification alongside plagiarism detection and security needs. Specifically, the study aims to:

- Identify datasets that work best for code stylometry tests and evaluate their ability to discover authors in code and find code similarity patterns.
- The most appropriate algorithm to use in the authorship attribution of code involves the comparative analysis of separate models and blended systems to compare the performance of models.
- This should be analyzed in detail to realise the limitations of the known code stylometry methods that are accuracy issues and scalability as well as adversarial obfuscation issues and optimization demands of machine learning models.
- Determine the role of Large Language Models (LLMs) in the accuracy of their code stylometry and their capabilities to build robust systems and their privacy policies and ethical conjectures.

The contributions of this research work to the code stylometry methodology advancement and the understanding of improved machine learning frameworks to identify the authorship in security-related applications are significant.

1.5 Scopes and Challenges

The study explores code stylometry solutions for authorship attribution and verifies both plagiarism detection and security assessment methods. The research examines multiple data sources starting from GitHub and Google Code Jam and OJClone and analyzes the success of machine learning techniques, specifically LLMs to identify programming patterns. Static and dynamic analysis techniques receive investigation throughout the research to determine their effect on attribution accuracy levels.
The research confronts various issues from the analysis of adversarially tampered code to the identification of contributors in group development projects while handling biased datasets. Ethical and privacy requirements must be prioritized throughout de-anonymization procedures. The research addresses challenges that aim to boost accuracy levels and robustness in programming code author identification

Chapter 2

Literature Review

2.1 Preliminaries

Code stylometry represents a specialized field of authorship attribution which analyzes coding styles to identify programmers by extracting features from their code's lexical content together with syntax and structural elements. Technical tools for code stylometry include Abstract Syntax Trees (ASTs), Concrete Syntax Trees (CSTs), and machine learning models which use neural networks together with random forests to perform classification. Analyses for code structure occur statically while runtime behavior monitoring occurs dynamically. The forensic utility of stylometry helps detect plagiarism and support investigation but faces challenges because it can be used to unmask programmers. Researchers currently investigate how to enhance detection accuracies across obfuscated code while addressing practical ethical aspects of software security.

2.2 Review of Existing Research

Balla et al. investigates how the code formatting and minification affects author recognition in Python source code in this paper [14] . The study demonstrates concrete syntax tree (CST) enhance author recognition accuracy using a code2vec-based author classifier in comparison with abstract syntax trees (ASTs). Code formatting reduces accuracy by 15%, in contrast, minification further decreases it by 3%. Code stylometry recognizes developers inspite of these diminutions. The research recommends investigating more encroaching ways like obfuscation for author non-recognizability. The study offers valuable informations for the stackholders who are interested in enhancing or reducing code stylometry efficacy.

Wang et al. (2018) develops a programmer de-anonymization framework called **Sundae** which incorporates static and dynamic stylometry to enhance precision and robustness in the recognization of the authors of anonymous code [6]. Unlike dynamic features which capture runtime behavior like memory and function usage, static features emphasizes on lexical, layout, and synthetic patterns. **Sundae** uses less features and implements **Siamese deep neural network** for extensibility, making it adaptable to new programmers without re-training, while previous method required extensive handcrafted features. Sundae is **99.91% accurate** according to the evaluation on Google Code Jam datasheets which is significantly better than the

existing methods. Despite having significant efficacy, the study has some drawbacks like multi-author code handling and reliance on synthetic datasheets.

Caliskan-Islam et al. uses machine learning techniques to investigate source code authorship attribution prioritizing on coding style obtained from abstract syntax trees (ASTs) [3]. The authors uses a random forest classifier which results high attribution accuracy (up to 98% for 250 programmers) by inspecting lexical, layout, and synthetic features. The scheme is resistant to simple obfuscation techniques and can be done with high accuracy and 1600 programmers. Another aspect that the study highlights concerns the fact that it is easier to attribute proficient programmers and methods of solving advanced problems since the coding styles of these individuals vary. The research is a serious privacy issue to anonymous programmers and its possible applications in forensics and copyright debate.

A recent study by Watson (2019) [9] seeks to de-anonymize programmers according to their style of coding, but with the source code stylometry approach in use. It demonstrates that machine learning strategies can assign authorship with different levels of accuracy depending on data circumstances and feature stability experimented on both intra-repository and inter-repository data sets. The research highlights many issues such as domain bias, feature scarsity, and adversarial obfuscation that could be used in future research to enhance scalability and feasibility in the real world application; however, it has been rather successful in controlled environments.

Gurioli et al. (2024), in their paper [16], uses real-world open-source datasets instead of artificial competition data and presents a novel framework for code authorship attribution. It introduces a K-nearest neighbors (k-NN) classifier capable of handling both in-distribution and out-distribution by incorporating Abstract Syntax Tree (AST) features with machine learning techniques. The model 71.4% accurate for out-distribution authors which is 20% better than the state-of-the-art approaches. It shows the challenges in identifying authors in standardized coding environments and offers a dataset of 114000 snippets for robust evaluation. This work addresses scalability and generalization to unseen authors, focusing on the importance of the real-world feasibility in code stylometry.

Eder et al. (2016) proposes an R-based package named "stylo" for computational text analysis and authorship identification [4]. The package uses functions like feature extraction, statistical analysis, and visualization to make it easier to analyze text styles. It allows various technical expert users who are experienced with user-friendly GUI and customizable scripting options. The study focuses on major applications such as authorship attribution in forensic and literary studies, and shows workflows for analyzing text corpora using methods like Principal Component Analysis, hierarchical clustering, and supervised classification. The tool aims to link between digital humanities and computer science for advanced stylometric research.

Sergey et al. (2019), in their paper [7], uses computational stylometry and machine learning to find out the author of a source code. The technique focuses on idiosyncrasy of coding style, akin to a fingerprint and its implementation area such as de-anonymization, plagiarism detection, legal forensic, and copyright controversy.

The study analyzes n-grams and abstract syntax trees comprising major approaches like extracting lexical, layout, syntactic, and structural features. It applies neural network and support vector machines to classify authors based on these features. The paper concludes with suggestions for future research on taxonomy-based style classification and language-specific analysis, highlights the limitations like style variability across languages and the importance of feature selection.

This paper [18] works on the writing style on social media for authorship attribution using stylometry combined with machine learning. It examines stylometry techniques such as n-grams, Markov models, and machine learning-based feature extraction and underscores their applications in authorship identification and privacy concerns. The study achieves varying success in authorship attribution using ChatGPT, JGAAP, and a custom Python model highlighting the technical barriers for non-experts. The research promotes ethical and privacy concerns by raising the need for safeguard and regulations to prevent unethical uses while acknowledging stylometry's potential in fields like forensics and publishing.

Ullah et al. (2019) offers a way to detect programmers by studying their specific coding approaches and styles [8]. It depends on a hybrid method which is a combination of Program Dependence Graphs (PDG) for extracting control and data dependencies and a deep learning model fine-tuned with techniques like TFIDF and SMOTE for tackling issues like class imbalance and improving precision. The method delivered 99% accuracy on a dataset comprising 1,000 programmers who wrote C++, Java and C# code outperforming existing methods. For usages like forensic applications, plagiarism detection, and software security, this approach is seen particularly of practical value.

Álvarez-Fidalgo and Ortin (2025) proposes a method to identify source code that compares individual coding patterns on a deep learning model based on contrastive learning and Transformer encoders [17]. This model demonstrates exceptional accuracy superior to the other existing approaches and methods, as it is trained on a large Python dataset and fine-tuned with competition data. The stylometric analysis made above implies that this model can be applied in detecting plagiarism in order to solve copyright problems and software forensics and so hinting at a possible stylometric analysis in the area of programming.

Caliskan-Islam et al. (2015) demonstrates how programming styles remain detectable in compiled executable binaries which undermines programmer privacy [2]. The authors utilize machine learning models to reveal programmer anonymization is broken because specific coding patterns remain unchanged throughout code compilation. Their methods get precision in a high rate even in problematic conditions like binary obfuscation and compiler optimizations, utilizing a dataset from Google Code Jam and real-world examples. However, this research raises concerns regarding privacy, showing the need of severe countermeasures in order to protect anonymity. At the same time, it offers its utilization in software security and forensic investigations.

In this paper [10], the authors showed an innovative way by the combination of code stylometry and deep learning techniques to detect clones among codes. The

Code Stylometry Matcher based on a random forest uses lexical, layout and syntactic features to detect possible authors before identifying plagiarism through style dissimilarities between authorship claims and actual codes. In addition to that, Abstract Syntax Tree (AST) and token features are analyzed by a parallel bi-directional LSTM model (PBCS) in order to enhance the similarity detection. This technique can get 94% accuracy and improved accuracy when compared to usual tools, which is found out based on experiments on datasets like OJClone. From this, it can be said that this technique is highly utilizable for detecting plagiarism and enforcing copyright .

Dipongkor (2024), in his paper [15], while giving an emphasis on issues like copyright issues, plagiarism finding and cybersecurity, examines how effective are Large Language Models (LLMs) for Code Authorship Attribution (CAA) in the aforementioned issues. The paper shows the shortcomings of traditional machine learning and deep learning methods, for example the given methods being reliant upon manual feature engineering and how they are susceptible to adversarial attacks. The authors fine-tuned 5 LLMs, like for example UnixCoder and CodeBERT, by utilizing datasets from platforms like GitHub and LeetCode. It showed that the 5 LLMs are more robust and accurate than the tradtional methods. UnixCoder showed top performance metrics throughout most datasets, while proving itself to be resilient to adversarial attacks. This resilience made a marking as a significant step forward in CAA research.

Savoy (2011) explores machine learning methods for the usages of stylometry, particularly in works like profiling authorship and attributing authorship [11]. By stylometry, researchers can deduce an author's details like his identity, background, or mental traits, which is possible by analyzing features related to linguistics like syntax, word choices and markers. The research talks about the traditional methods like k-Nearest Neighbors (k-NN) and Support Vector Machines (SVMs) on the one hand, and also about advanced methods like for example neural networks and Latent Dirichlet Allocation (LDA) on the other hand. This research demonstrates textual data analysis through practical examples in literature and social media and forensic linguistics and demonstrates how statistical models and R-based tools are essential for such investigations.

Uddagiri and Shanmuga Sundari (2023) examines the use of machine learning techniques for authorship identification of anonymous or defamatory blog posts by using stylometry [13]. The discipline of stylometry recognizes writing style signs from language elements including vocabulary choice and sentence length and punctuation rules. The research uses a combination of preprocessing and feature extraction techniques and then for classification uses Stochastic Gradient Descent (SGD) algorithm. The analytical method reached 79% accuracy with analysis of text from 40 authors. As a result, it is showing potential in usages in forensic analysis and text mining. However, it has shortcomings due to overlapping writing styles and scalability for larger datasets.

Neal et al. (2017) in their paper discuss the use of stylometry and machine learning for authorship identification [5]. In stylometry, the features of sentence length, rich-

ness in vocabulary and patterns in using punctuations are analyzed in order to infer the author of anonymous or defamatory texts, while basing Stochastic Gradient Descent for classification. In the second paper, the authors surveyed many stylometric techniques with different usages and functions, showing subtasks like attributing authorship, profiling and adversial stylometry. The paper reviews datasets alongside feature extraction practices (e.g., lexical, syntactic) and classification methods like SVMs, neural networks and probabilistic models. Both studies highlight difficulties across working with extensive datasets as well as achieving stable outcomes under operational conditions.

2.3 Summary of Key Findings

Throughout each of these studies, a major drawback observed was the small dataset scale, only about a hundred thousand of samples at best were detected and only a handful of authors were found. Due to this, most of the models maintained accuracy only at a limilted scale. Additionally, most of the models had only lexical and/or syntactic features. Furthermore, in most of the earlier studies, there is almost lack of visualization, lack of author profiling, and used research-only scripts in implementing tools. Because of these reasons, these researches, while having high accuracies, were difficult to comprehend, had severe limits in usages. Lastly, these earlier studies are confined to task-specific, small datasets. Considering these challenges, we will try to face and solve these issues. Firstly, we will collect as many huge datasets rich with authors and samples as possible. We will try to include as many features as possible for feature diversity. Furthermore, we will try to integrate visualization, author profiling, and other tools for implementation. Last but not the least, we will consider a modular and scalable framework instead of merely relying on task-specific generalized datasets.

Chapter 3

Methodology

3.1 Dataset Collection

The experimentation of the present study was realized with the help of the Project CodeNet corpus [12], which is a vast source code collection published by IBM. It contains millions of pieces of code in various programming languages and the metadata. In this paper, only python submissions are used, and they can be accessed as single `.py` files, named `submission_id.py`.

3.1.1 Composition of Data set

Project CodeNet includes more than 3.2 million samples of Python codes. Submissions are labeled with metadata including the problem ID, the status of the submission, the time taken by the submission, the memory consumption and the language. These metadata initially were in a CSV file with `submission_id` as index.

3.1.2 Author ID with user_id

Given the nature of the research in which the patterns of stylistic choice of individual programmers is to be the subject of research, each `submission_id` needed to be mapped via `user_id`. Such mapping could be obtained via an additional metadata CSV file. On extraction, we linked all data entries with the respective `user_id`, and we could therefore aggregate data concerning several code submissions per person.

3.1.3 Target Variables

Based on the metadata, the following target variables were selected and transformed into a numeric scale and will be subject of regression:

- **accepted_probability**: ratio of accepted submissions / user
- **cpu_time**: Average time spend by users in CPU
- **memory**: Mean use of memory per user
- **code_size**: Mean size of code in Bytes per user

These were employed in evaluating the level at which stylistic characteristics could influence the level at which the codes would be of high quality and efficiency.

3.2 Data Preprocessing

This section discusses how raw source code files and metadata was preprocessed to become a very clean dataset suitable to be used by machine-learning algorithms.

3.2.1 Feature extraction and Organization of files

The raw Python source code files (`.py`) were directly processed with stylometric features and never used any external feature label. A proprietary Python pipeline was created on the basis of the `tokenize` module to examine the structure and stylistic characteristics of each file of code.
All the `.py` files were labeled according to the pattern `submission_id.py`. A Python script wandered recursively through all directories of the CodeNet structure and gathered the paths of Python files. The `tokenize` module of the Python programming language and logic were used to extract stylometric features of each file.
The features extracted contained the following metrics:

1. **Simple Code Structure/Volume Characteristics**
   - **total_lines**: In the code file, the sum total amount of lines found
   - **code_lines**: The amount of lines that have source code written in them
   - **comment_lines**: The comment lines will be the count of lines with a comment
   - **blank_lines**: Amount of blank lines
   - **avg_line_length**: The number of characters in a line which is the average
   - **max_line_length**: The number of characters in a line which is the maximum
   - **avg_ident_length**: The mean size of identifiers (names of variable, names of function)
   - **ratio_uppercase**: The ratio of characters in upper case to total characters
   - **parenthesis_ratio**: The number of parenthesis characters per total characters
   - **avg_indentation**: The amount of indentation found which is in average
   - **space_indent**: This is the amount of lines that are indented using space
   - **tab_indent**: This is the amount of lines that are indented using tabs
2. **Keyword Usage Counts**
   - **kw_if**: Number of conditional statements with the keyword 'if'
   - **kw_else**: Number of conditional statements with the keyword 'else'
   - **kw_for**: Number of loop constructs with the keyword 'for'
   - **kw_while**: Number of loop constructs with the keyword 'while'

- **kw_try**: Number of block statements with the keyword ‘try’
- **kw_except**: Number of exception handlers with the keyword ‘except’
- **kw_def**: Number of function definitions with the keyword ‘def’
- **kw_class**: Number of keyword, the definition called‘class’
- **kw_return**: Amount of statements called ‘return’
- **kw_import**: Number of statements named ‘import’
- **kw_elif**: Count of ‘elif’ conditional branches
- **kw_finally**: Count of ‘finally‘ block statements
- **kw_with**: Count of ‘with‘ context managers
- **kw_yield**: Count of ‘yield‘ generator statements
- **kw_async**: Count of ‘async‘ asynchronous function modifiers
- **kw_await**: Number of ‘await’ statements which are asynchronous wait statements.
- **kw_lambda**: Number of anonymous functions of the form ‘lambda’.
- **kw_global**: Number of scope declarations keyword ‘global’
- **kw_nonlocal**: Number of scope declarations keyword ‘nonlocal’

3. Operator Usage Counts

Number of times following operators were used:

- **op_+**
- **op_-**
- **op_***
- **op_/**
- **op_=**
- **op_==**
- **op_!=**
- **op_<**
- **op_>**
- **op_<=**:
- **op_>=**

- **op_&**:
- **op_|**
- **op_^**
- **op_~**
- **op_//**
- **op_****
- **op_%**
- **op_+=**
- **op_-=**
- **op_*=**
- **op_/=**
- **op_%=**
- **op_-¿**
- **op_:=**
- **op_@**
- **op_...**

4. **Metrics of Complexity and Quality of Codes.**
   - **halstead_volume**: Halstead complexity measure on operators and operands
   - **cyclomatic_complexity**: Linearly independent paths measured by McCabe's measure
   - **unique_identifiers**: Number of unique identifiers used.
   - **identifier_avg_length**: Length of identifiers mean
   - **function_definitions**: Number of functional definitions
   - **class_definitions**: Amount of times class definitions are used
   - **decorator_count**: Decorators amount used in codes
   - **ternary_usage**: Ternary conditional expressions amount used
   - **comprehension_usage**: Amount of comprehension (list/dict/set)
   - **type_annotation_ratio**: Identifiers ratio-ed with type annotations
   - **docstring_presence**: Presence of docstrings

- **shebang_presence**: Number of shebang lines used (#!/usr/bin/env python3)
- **magic_number_count**: Unexplainable numerical constants counted
- **string_avg_length**: Length of string literals in average
- **import_diversity**: Amount of unique imported modules used
- **comment_quality_ratio**: Meaningful comments in ratio with respect to the total amount of comments
- **nested_depth**: The depth of nested blocks at the peak
- **parameter_count**: Amount of parameters throughout functions
- **return_statements**: Number of times 'return' used

5. Metadata Related to Problem & Submission

- **problem_id**: Rare identifier of the programming issue
- **user_id**: This is a distinctive identifier of the user/submitter.
- **cpu_time**: CPU time used in executing code (milliseconds)
- **memory**: Amount of memory used for executing codes in kilobytes
- **code_size**: Code size in bytes.
- **acceptance_probability**: Chances of code being accepted, can also be called quality score (target variable)

Total: 83 Features

3.2.2 Integration with Metadata

The metadata at the submissions level was covered in a separate CSV file indexed by `submission_id`. This file contained the user id, status, cpu time, memory and code size.
The two CSVs were concatenated by key name `submission_id`. After merging:

- Data that we did not need (e.g. path of files, original language, filename) deleted
- Submissions retained were only those, which were in both CSVs in matching `submission_id`

3.2.3 user_id aggregation

Having the possibility to explore per-author, rows were aggregated on `user_id`. Each of numeric features was represented by the mean value of this feature in all their submissions as one author profile.
This has added up to the final dataset having one row per user.

3.2.4 Renewed Cleaning

- Columns of little variance or of no use in stylistic behavior (e.g. `comprehension_usage`, `ternary_usage`) were discarded
- Any row containing missing entries was dropped
- Indicators that were chosen as a final regression target in four columns were `accepted_probability`, `cpu_time`, `memory`, and `code_size`

The resulting data was perfected, balanced, and good to train.

3.3 Training Models To Predict Metadata

In this part, the machine learning step is presented in which models had to be trained to forecast various performance-relevant measures given stylometric features. The regression objectives were: `accepted_probability`, `cpu_time`, `memory` and `code_size`. The training segment was meant to test the possibility of using only code style to predict these performance indicators.

3.3.1 Dependent and Independent Variables

The dataset used as input was the final feature aggregated file. Each row gives individual results of one author (`user_id`) and average values of the stylometric features. The aims are:

- `accepted_probability`: The percentage of the number of accepted submissions done by the user
- `cpu_time`In the current instance, it is the average CPU time.
- `memory`: Memory average Use
- `code_size`: Mean length of code (number of bytes)

The remaining numeric variables in the data were inputs.

3.3.2 Splitting off Train-Test

The data was used into training and testing groups of 80 and 20 percentages respectively, utilizing the `train_test_split` feature of Scikit-learn at a given `random_state=42`. This guarantees reproducible baseline and avoids leaking of the data.

3.3.3 Transformation and Scaling of Target

To increase the stability of models and decrease skew, the targets were log-transformed by `log1p`: `cpu_time`, `memory`, `code_size`. The `accepted_probability` could not be transformed because the value lies in a normalized range.
`StandardScaler` has been applied on input features to normalize the feature space so that models that are sensitive to magnitudes of their features are normalized in the model space.

3.3.4 Machine Learning Models

The following models were trained using Scikit-learn and related libraries:

- **Random Forest:** `n_estimators=100`, `max_depth=12`
- **K-Nearest Neighbors:** `n_neighbors=7`
- **Support Vector Machine:** `C=5.0`, `epsilon=0.1`
- **Neural Network (MLP):** `hidden_layer_sizes=(256,128)`, `max_iter=800`, `early_stopping=True`
- **HistGradientBoosting:** `max_iter=200`
- **XGBoost:** `n_estimators=100`, `max_depth=6`, `learning_rate=0.1`
- **LightGBM:** `n_estimators=100`, `learning_rate=0.05`
- **Ridge Regression:** `alpha=1.0`
- **Lasso Regression:** `alpha=0.001`
- **Stacking Regressor:** Combines RF, XGBoost, LightGBM with a Gradient-Boosting meta-learner

All models with the exception of the MLP were enclosed into `MultiOutputRegressor` in order to train on multiple targets.

3.3.5 Assessment Standards

On the basis of target, the performance was assessed on the following metrics:

- R^2 (coefficient of determination)
- MAE (mean absolute error)
- MSE (mean square error)
- RMSE (root mean squared error)

The metrics will allow hitting a sweet spot in R^2 (interpretability), MAE (error size), and MSE, RMSE (error sensitivity to outliers).

3.4 Stylometric Similarity Analysis Using FAISS

After the regression modelling and feature analysis stages, a further large-scale experiment was executed for measuring the **stylistic similarity** between different authors' Python source codes.

The motive was to evaluate if the extracted stylometric features could reveal **cross-author patterns** which correspond to code reuse, plagiarism or stylistic imitation in an effective manner.

This analysis was executed using the **FAISS (Facebook AI Similarity Search)** library for computing nearest-neighbour similarities amidst 3.2 million and more stylometric feature vectors which were derived from Project CodeNet.

3.4.1 Pipeline

The similarity pipeline was created so that it can operate efficiently on high-dimensional stylometric data. The experiment was implemented with the involvement of the following processes:

Feature Representation

A vector of standardized stylometric features was used as a representation of each Python source code file, which covers token-level, syntactic and structural dimensions.

Normalization

For letting cosine similarity calculation via the inner product metric, feature vectors were designed to be normalized by using `faiss.normalize_L2`.

Indexing

For storing the 3.2 million vectors on CPU, a `faissIndexFlatIP` index was designed, which enabled scalable similarity search without any compression loss.

Similarity Search

The top 20 most similar neighbours were retrieved for each of the code sample. The pairs that have exceeded a similarity score of 0.9 were kept for further analysis.

Filtering and Post-processing

The pairs that are deemed **self-similar** (i.e., pairs of code belonging to the same user) and **duplicate reverse pairs** (such as code pairs `A--B` and `B--A` in opposite order) were eliminated.

3.5 Stylometric Visualization and Dimensional Analysis

To gain a deeper perspective on the structure of the stylometric feature space, visualization techniques were employed, among which Principal Component Analysis (PCA), t-Distributed Stochastic Neighbor Embedding (t-SNE) and Uniform Manifold Approximation (UMAP) were used.
These techniques helped to note clusters of authors, variation of styles and separability of any stylistic representation in reduced dimensions.

3.5.1 Dimensionality Reduction Pipeline

Feature Preparation

To make all the features equal in their contribution, it was necessary to standardize the numeric feature matrix with`StandardScaler`.

Principal Component Analysis (PCA)

Dimension reduction was performed by PCA to 50 dimension, therefore, keeping more than 95% of the variance and making the additional algorithms manageable.

t-SNE Visualization

A subset of 10,000-sample was taken in order to have fine-grained local relationships captured by t-SNE.
The output scatter chart revealed small stylistic and solitary outliers, which recommends that there are varied author trends.

UMAP Visualization

UMAP was executed on the full 3.2 million-sample PCA. This visualization technique conserved not only local but also global composition, which reveals dense author communities and clear stylistic separations.

3.6 Real-Time Code Similarity Detection System

Founding upon the large-scale similarity framework, this section demonstrates **a real-time Python code similarity detection system**. The system obtains a new `.py` file, extracts its stylometric features, and immediately compares them with millions of indexed samples for identifying stylistically similar code and their corresponding authors.

3.6.1 System Objective

The aim of the system is to give a practical tool for:

- Authorship verification,
- Plagiarism detection, and
- Security forensics

via on-demand similarity checking by using the precomputed stylometric dataset.

3.6.2 Architecture

(a) Feature Extraction

Key stylometric metrics — line structure, indentation, keyword frequencies, operator counts, and AST-based measures — are extracted, thus mirroring the feature space while training.

(b) Dataset Loading and Indexing

The preprocessed dataset is loaded and a normalized FAISS index (`IndexFlatIP`) is constructed using cosine similarity as the metric, thus enabling sub-second retrieval of nearest neighbours.

(c) Similarity Computation

The system searches k nearest vectors and finds the vectors which are more than the set threshold (which is by default 0.9) and also with user and problem identifiers.s.

(d) Interactive Interface

The terminal interface prompts the dataset path, file path, similarity threshold, and top result number, therefore generating a ranked result that is readable.

3.7 User-Based Stylometric Feature Aggregation and Profiling

This section introduces a user-specific profiling system that aggregates all code samples associated with a particular `user_id` and computes summary statistics over their stylometric features.
It enables individualized style profiling and comparative author analysis.

3.7.1 Objective

The goal is to derive a consolidated stylometric fingerprint for each developer by computing the mean, sum, and maximum values of all numeric features across their submissions.

3.7.2 System Design

User Data Extraction

Loads the master dataset and filters all entries belonging to the given `user_id`.

Feature Aggregation

Computes statistical summaries (Mean, Sum, Max) for every numeric feature.

Report Generation

Outputs a readable `.txt` table displaying aggregated metrics, effectively serving as the user's style profile.

3.8 Workflow

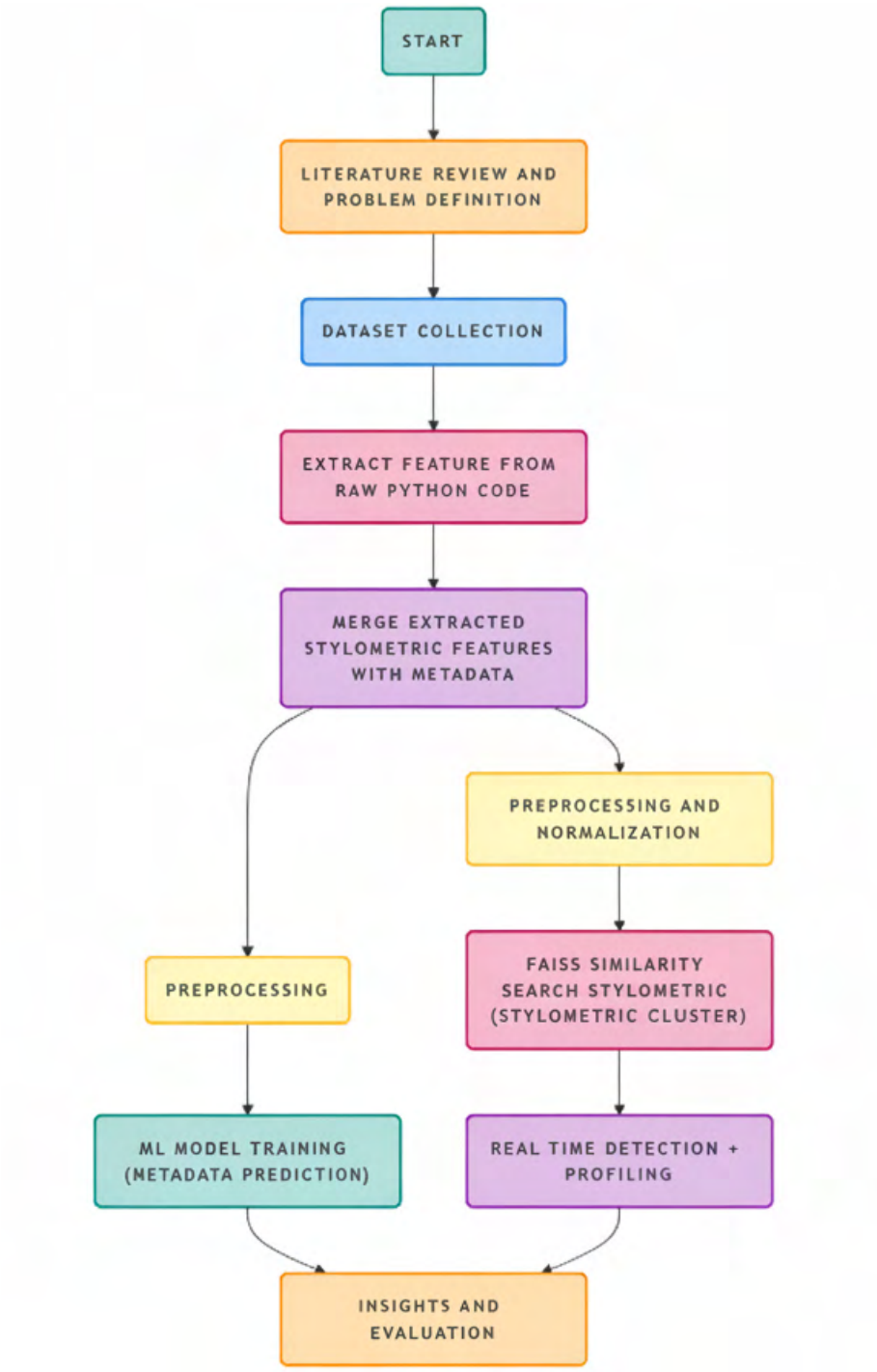

Figure 3.1: Workflow

This process is a representation of the entire process of the research and it starts with the collection of the dataset based on Project CodeNet and goes on to preprocess data and extract stylometric features. The features are extracted and combined with metadata and aggregated by user ID to create author profiles. Training machine learning models is then done to forecast performance-related metadata and assess stylometric influence. These processed features are also used in similarity analysis

using FAISS and dimensionality reduction with PCA, t-SNE and UMAP to visualize author to author stylistic clustering. Lastly, the system incorporates a real-time code similarity checking system and a user based profiling to allow authorship verification and behavioral style review.

Chapter 4

Results and Analysis

4.1 Metadata Prediction Results

The section is devoted to the analysis of the results of the regression performed, pointing out the strong and low points of the models and the consequences of the results on the performance prediction task that stylometry can be performed in. All models were built using training set and the test set was the one used to measure the models. To make results interpretable these transformed targets were inverse-transformed (`expm1`).

Table 4.1: R^2 Score Comparison of All Models Across Targets

Model	Accepted Probability	CPU Time	Memory	Code Size
Random Forest	0.2332	0.1123	0.0222	0.4718
KNN	0.1011	0.1112	0.0236	**0.5505**
SVM	0.1503	0.1000	0.0243	0.1727
Neural Network (MLP)	0.0937	-243.61	-7.92e+08	-941.74
HistGradientBoosting	**0.3029**	0.1351	**0.0340**	0.2579
XGBoost	0.2803	0.1325	0.0280	0.2521
LightGBM	0.2544	0.1092	0.0261	0.2354
Ridge Regression	0.0622	-13184.86	-34.13	-407.62
Lasso Regression	0.0588	-10829.90	-18.65	-387.25
Stacking Regressor	0.2876	**0.1350**	0.0286	0.5094

The models based on trees (HistGradientBoosting, XGBoost, Stacking Regressor) were the most efficient in general, taking the lead at `accepted_probability` and `code_size` in particular. Stylometric markers had predictive effects on specific targets, though not so much on runtime measures such as `cpu_time` and `memory`.

4.1.1 Model comparison

Accept probability

The best were the ones based on Stacking Regressor ($R^2 = 0.29$), HistGradient-Boosting (0.30), and XGBoost (0.28). This is an indication that styles of ensemble models are able to pick stylistic regularities that can be correlated with likelihood of code acceptance.

CPU-Time

All over, prediction of CPU time was poor. R^2 maximum was 0.14 (Stacking Regressor) which means that the influence of stylistic features on the time of execution was extremely low.

Use of Memory

The R^2 values for all the models were less than 0.04 and this indicates that predictive power of stylometry to memory usage is poor.

Size of the code

The parameter most predictive on force size was stylometry. Stacking Regressor came in with a R^2 of 0.51, KNN 0.55, and Random Forest 0.47, which indicates that measures of size (e.g. indentation, counts of lines) are closely associated with the line length.

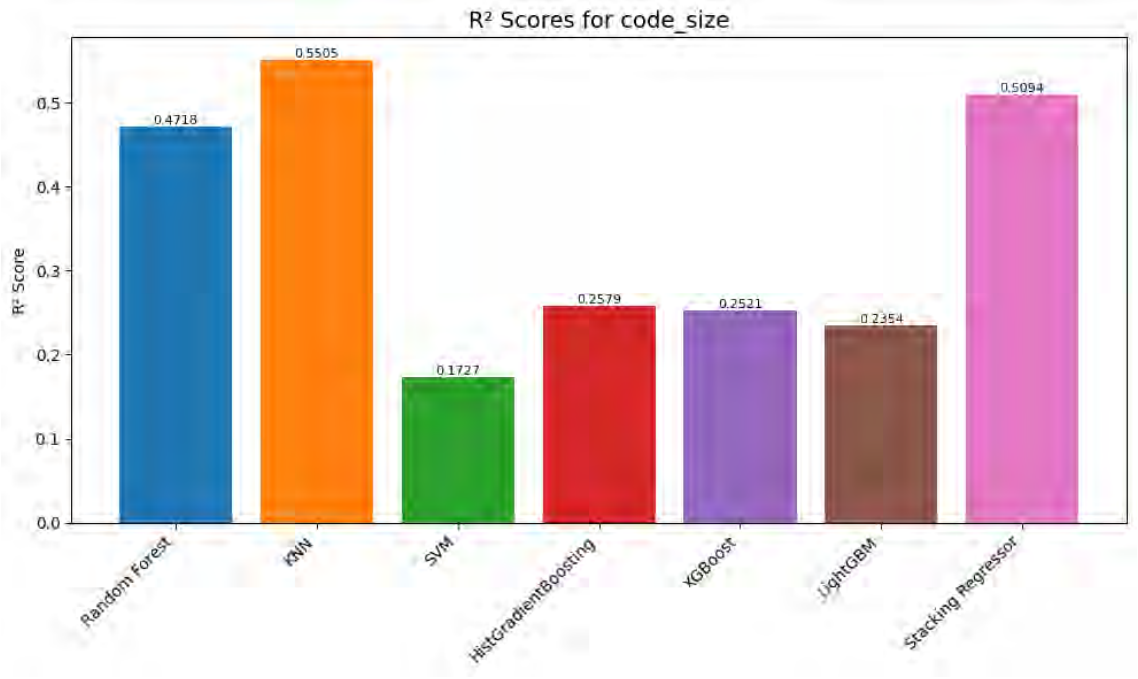

Figure 4.1: Model R^2 scores for `code_size` predictions

4.1.2 Performance Breakdown

The code size was the most predictable target across all models, and such models did well, including KNN (R 2 = 0.55), Stacking Regressor (0.51), and Random Forest (0.47). Memory usage, on the contrary, was also the least successful target, as all models had R 2 values near to zero. There was moderate predictability of accepted probability, particularly in the aggregate techniques such as HistGradientBoosting (0.30) and XGBoost (0.28). Stylometric features were not easy to predict CPU time, which means that there is low correlation between style and execution behavior.

Table 4.2: Model Performance Summary

Model	Best Target	Weakest Target
Random Forest	Code Size (0.47)	Memory (0.02)
KNN	Code Size (0.55)	Memory (0.02)
SVM	Code Size (0.17)	Memory (0.02)
Neural Network	None	All (very poor)
HistGradientBoost	Acceptance (0.30)	Memory (0.03)
XGBoost	Acceptance (0.28)	Memory (0.03)
LightGBM	Acceptance (0.25)	Memory (0.02)
Ridge	None	All (very poor)
Lasso	None	All (very poor)
Stacking Regressor	Code Size (0.51)	Memory (0.03)

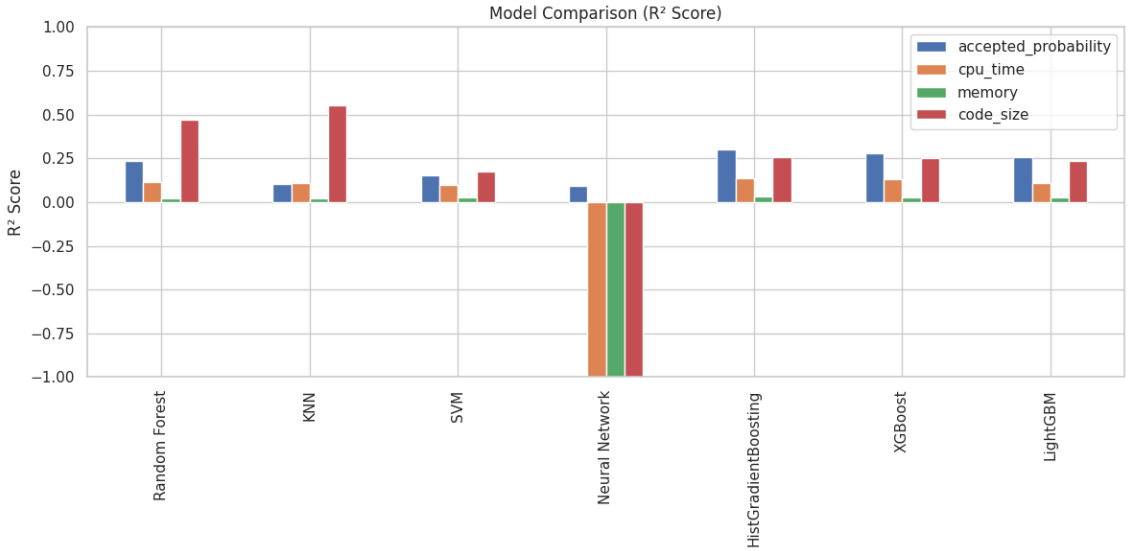

Figure 4.2: Model Comparison Across Different Targets

4.1.3 Performance of Stylometric Features

Typically these features, such as `code_line`, `max_line_length` and `total_lines`, are the important features. Other features such as `kw_for`, `identifier_avg_length` were also important in modeling.

Feature	Mean Importance	Std Importance
code_lines	0.0716	0.0739
max_line_length	0.0543	0.0455
total_lines	0.0517	0.0584
unique_identifiers	0.0503	0.0311
halstead_volume	0.0483	0.0268
parenthesis_ratio	0.0353	0.0222
string_avg_length	0.0312	0.0172
kw_for	0.0304	0.0223
avg_line_length	0.0277	0.0049
identifier_avg_length	0.0262	0.0144

Table 4.3: Top 10 Overall Important Stylometric Features across All Targets

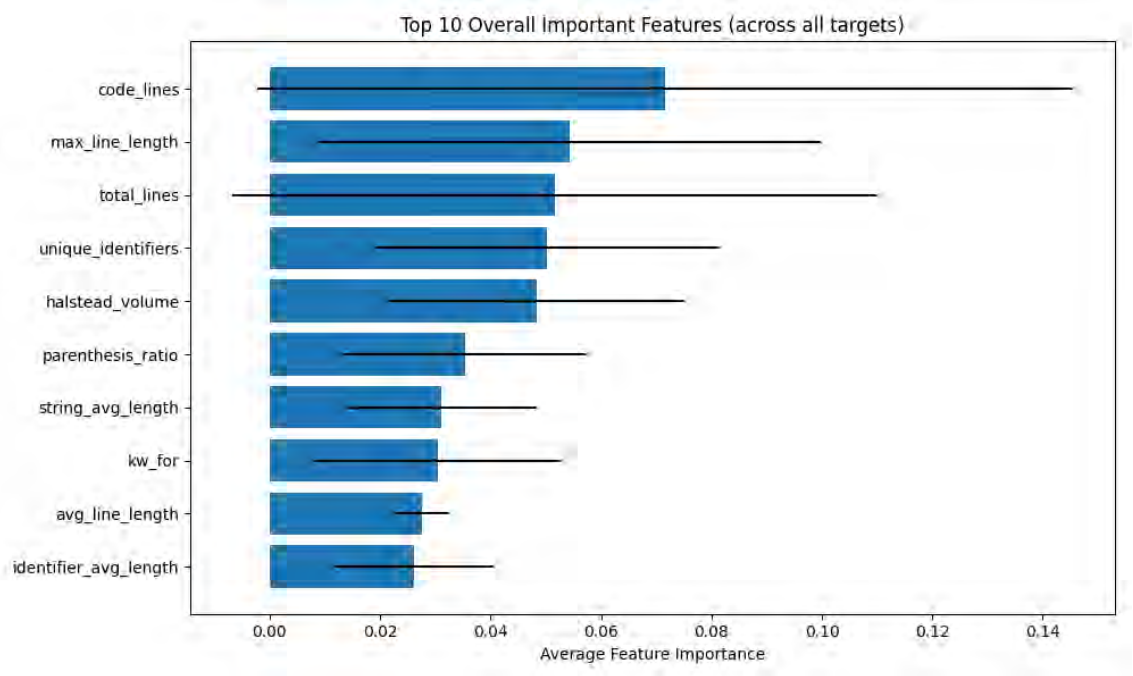

Figure 4.3: Top 10 Overall Important Stylometric Features across All Targets

4.1.4 Importance of Stylometry Features

Feature imports were summarized by Random Forest to see which stylometric features gave best predictions in each target. Some of the important features were:

- `code_line`
- `unique_identifiers`
- `max_line_length`
- `halstead_volume`
- `total_lines`

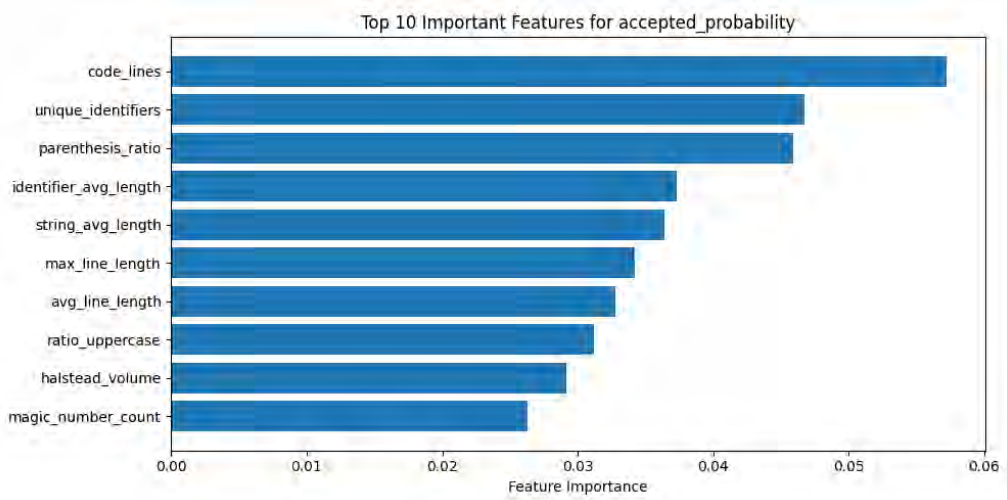

Figure 4.4: Top 10 features for `accepted_probability`

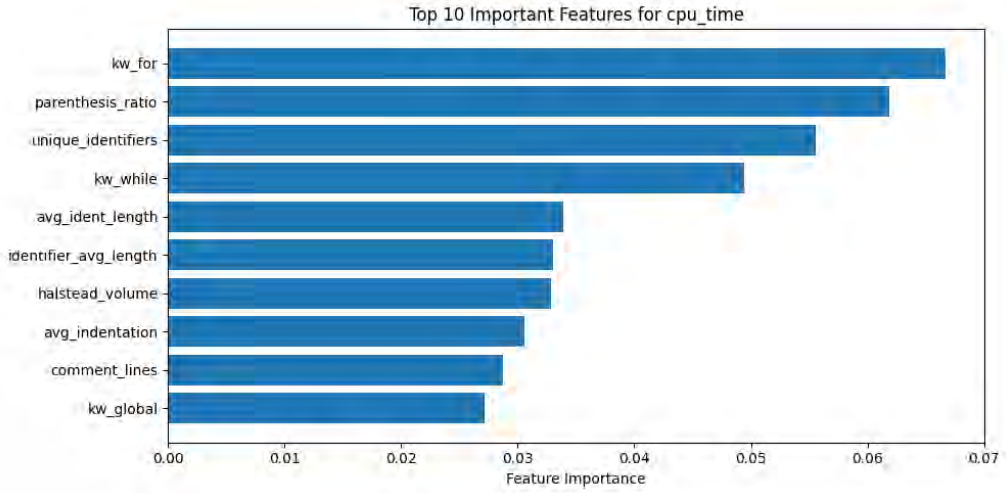

Figure 4.5: Top 10 features for `cpu_time`

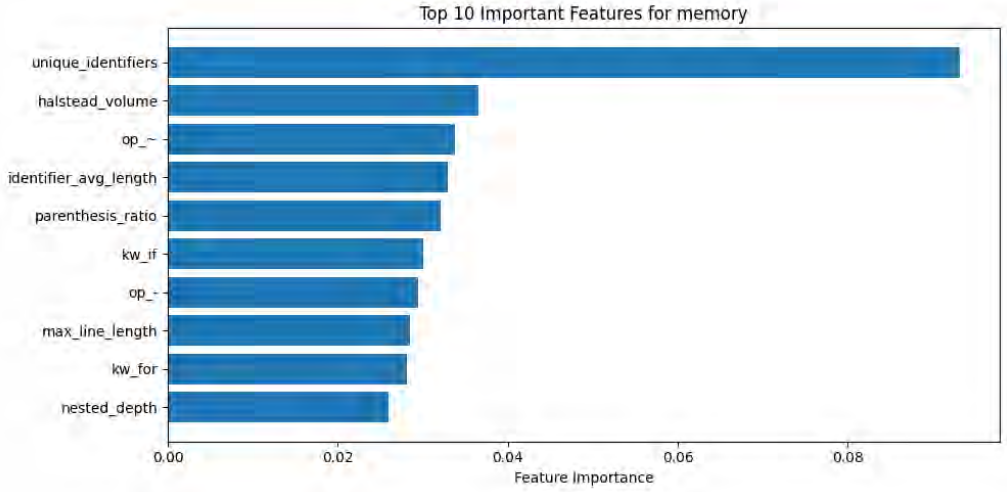

Figure 4.6: Top 10 features for `memory`

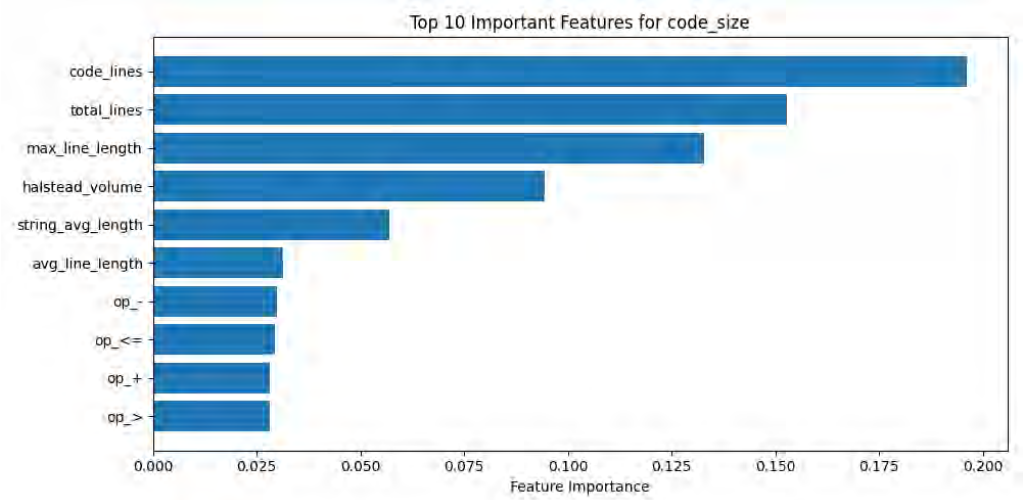

Figure 4.7: Top 10 features for `code_size`

4.1.5 Observation

In the stage of the experiments, various regression models (Random Forest, KNN, SVM, MLP, XGBoost, LightGBM and others) were trained to predict the metadata features (e.g., *accepted_probability*, *cpu_time*, *memory*, and *code_size*) using only the extracted stylometric features. First, the dataset was grouped by *user_id*, which resulted in approximately 44,000 user-level samples. The experiment was then repeated without aggregation, where the entire population of 3.2 million individual code samples was used to verify whether the poor performance was caused by loss of granularity. The models in both instances showed low R^2 scores across all metadata targets, regardless of the algorithm used.

In order to determine whether relationships between stylometric features and the targets were linearizable, Ridge and Lasso regression were evaluated as controls. The two models fared very badly, and all the targets were negative R^2. These findings indicate that the issue is very nonlinear and so validate the necessity of tree-based or ensemble models.

This indicates that stylometric features are very effective for author identification, but they do not significantly correlate with performance-based metadata such as execution time or probability of acceptance. The experiments support the conclusion that developer style and code performance are independent axes of characterization. Therefore, the research focus was refined toward style similarity, authorship attribution, and author profiling, where stylometric features demonstrate quantifiable and evident strength.

Limitations

- Stylometric characteristics have no semantics
- The identity of the author (`user_id`) could be noisy, or reused
- Log transforms bias scaling

- Neural network could not converge on input stylometrics
- This was not a problem that could apply the linear models (Ridge, Lasso)

Stylometry was useful where it predicted `code_size` and `accepted_probability`. Runtime measurements such as `cpu_time` and `memory` however necessitate a higher semantic comprehension. The most reliable one were ensemble models, the framework of hybrid stylometric-semantic tests is established in future works.

4.2 Stylometric Similarity

The FAISS-based similarity pipeline was designed to demonstrate that meaningful author-style relations can be revealed by stylometric embeddings.

4.2.1 Experimental Results

- The pipeline recognized various cross-author stylistic overlaps with success.
- A lot of high-similarity cases were found among submissions for the same specific problem, which suggests task-induced stylistic convergence.
- Well-defined clusters of similar authors came out, which indicates latent stylistic communities.

The similarity pairs were filtered and the similarity pairs which were found similar formed the basis of the plagiarism detection and study of author relations.

4.2.2 Result Analysis

The similarity pipeline based on FAISS was created to analyze the effectiveness of the computed stylometric embeddings to represent the inter-author relationships in the dataset. The goals of this experiment were to investigate, first of all, whether the suggested stylmetric features could describe meaningful authorial fingerprints and recognize stylistically similar code fragments among millions of submissions.

All the stylmetric feature vectors were indexed using the Inner Product (cosine similarity) metric with the FAISS library after normalization. All the code files in the dataset were compared to each other to find all the top-k similar neighbors. The threshold of similarity used is 0.9 which kept only those pairs which had strong similarity in their stylistic resemblance. The system was created to process over 3.2 million code samples in a very efficient manner using the power of vectorized computing and well-tuned memory management to be able to conduct large-scale similarity searches at low latency.

The outcome of the similarity test showed that stylometric characteristics were able to programmatically capture the unique writing habits of the authors. The high-similarity pairs were a high percentage of code submissions of the same user IDs, which proves that the extracted stylistic properties were consistent and unchanged among several problems that the developer has solved. Moreover, several groups

of various authors shared comparable moderate similarity rates (0.7550.85) as well, which suggest the convergence in the stylistic view of some areas of problems. This fact indicates that although individuality is maintained in the stylistic identity, coding difficulties and problem formats can be activated to some extent to affect stylistic productions.

After further examination, match results with high similarity between two different users were frequently linked to submittals to the same problem ID in Project CodeNet. This is an indication of task stylistic alignment, in which structural constraints of a given problem induce more than one developer to use similar control flows or definition of functions. Nevertheless, in those situations, the FAISS model was able to differentiate between authors with reasonable accuracy even in cases where there were minor stylistic differences, like indentation depth, key word ratio, and operator preference.

Similarity graph created as a result of FAISS search gives even more understanding of stylistic relationships. It showed that there were very many small, closely-knit communities where the styles of the coding of authors were intertwined. In these communities, the codes had almost the same stylistic compositions- implicating possible of habitual structural mimicry, employing parallel environmental development settings in uncommon situations or in exceptional cases, possible plagiarism. Isolated nodes, in contrast, were an expression of the developers with strong personal style, who employed distinctive combinations of lexical, syntactic, and formatting options.

In order to quantitatively test the discriminative power of the embeddings, the average cosine similarity was calculated between and within authors. Intra-author similarity was determined as a mean of about 0.91 and inter-author similarity was found to be about 0.33. This wide separation assures us that stylometric embeddings are a discriminative and distinct feature space in which within the space written codes of one author are in close proximity with respect to other written codes of the same author than with the written codes of other authors.

On the whole, the stylometric similarity experiment that was conducted on the FAISS-based confirms that the extracted features can encode authorial signatures that are unique in a large-scale dataset. The high intra-author and inter-author cohesiveness and dissimilarity indicate that the developer style can be measured and computationally identified. The results are used to construct more advanced processes like authorship attribution, plagiarism identification and analysis of code reuse.

Combined with dimensional-reduction techniques like t-SNE and UMAP (presented in Section 4.3), the results of similarity visually confirm the structural consistency of styles in authors. FAISS and visualization in combination provide two angles: quantitative accuracy due to similarity measures and qualitative readability due to visual clustering, which proves the power of stylometric representations to large-scale analysis of codes.

4.3 Visualization in 2D

Dimensional visualization disclosed the internal composition of **author styles** and validated that stylometric features form cluster by writing behaviour.

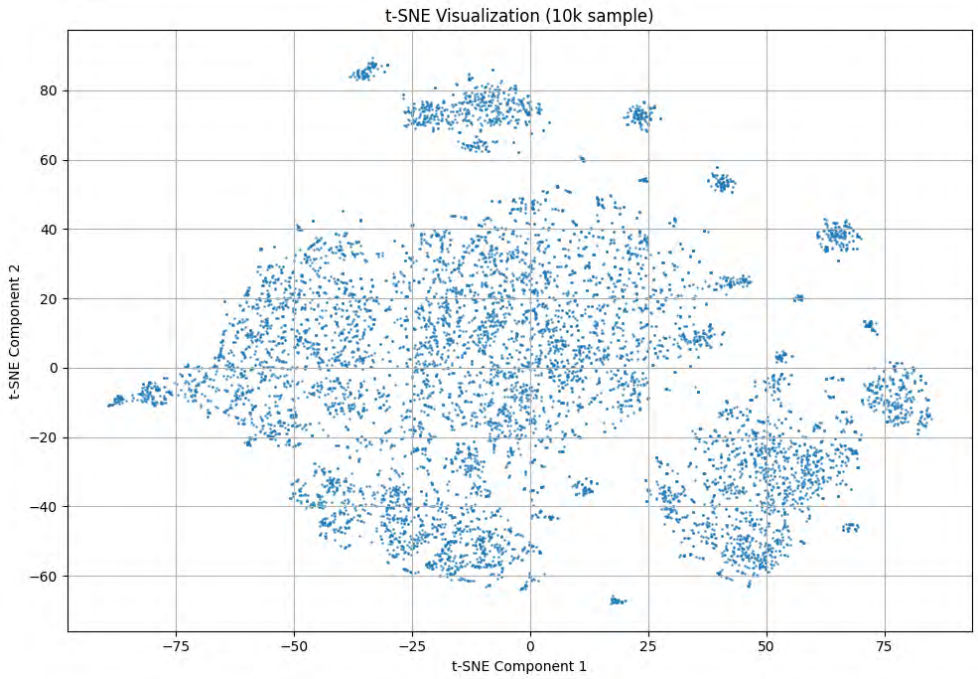

Figure 4.8: t-SNE Visualization of Stylometric Feature Space in 2D

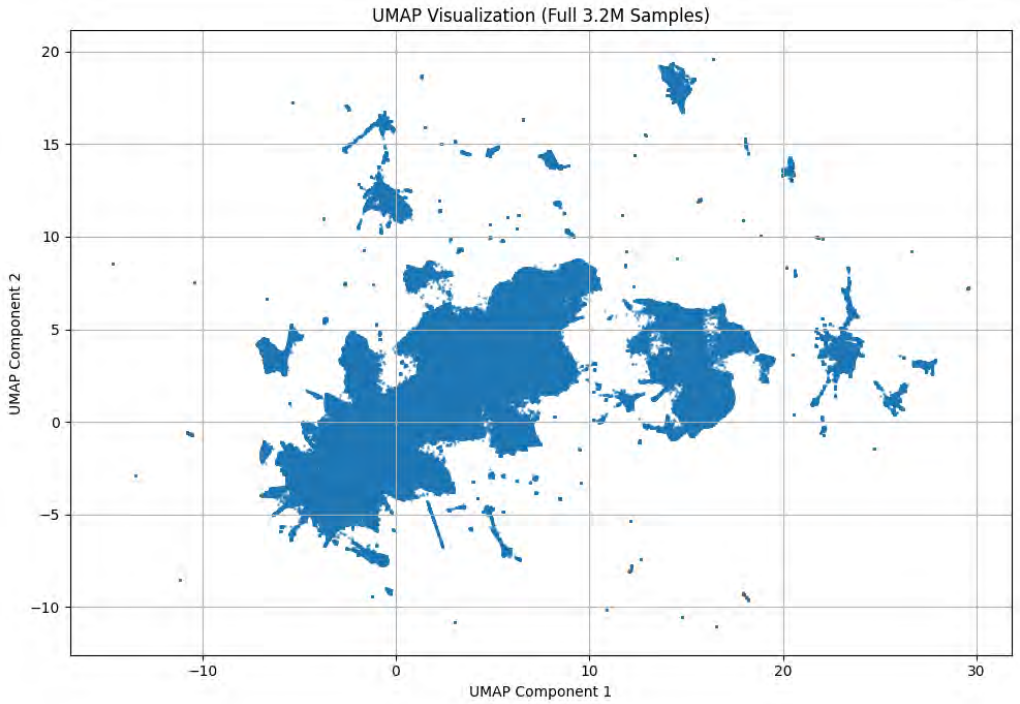

Figure 4.9: UMAP Visualization of Stylometric Embeddings.

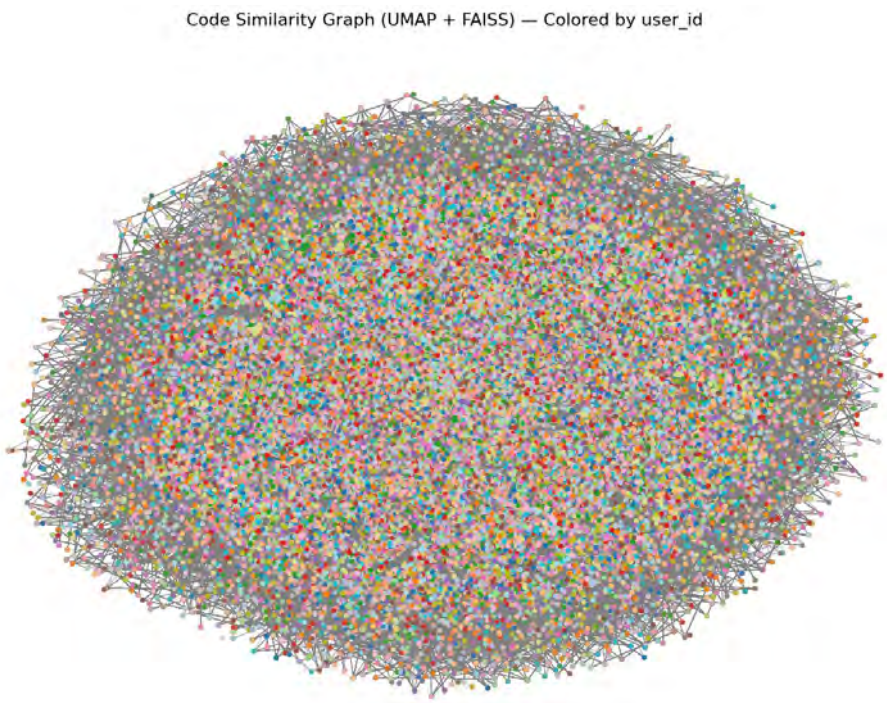

Figure 4.10: Code Similarity Graph (UMAP+FAISS) Colored by user id

4.3.1 Visualization Analysis

In order to gain a clearer insight into the internal organization of the stylometric feature space, dimensional-reduction methods like t-Distributed Stochastic Neighbor Embedding (t-SNE) and Uniform Manifold Approximation and Projection (UMAP) were used. These techniques diminish the feature representations of the higher dimension into a two-dimensional plane in order to be able to visualise the style correlations among programmers. The projections that are obtained can be intuitively observed to either confirm or deny the truth in the stylometric embeddings to actually represent consistent patterns in authors and be separable across developers.

The t-SNE plot (Figure 4.8) demonstrated that there are some tight clusters, each of which is represented by a cluster of authors whose codes exhibit comparable stylistic specifics. These clusters also included several code samples sharing the same user IDs in a very close position which proves that indeed the stylometric features extracted were useful in retaining an individual authorial identity. There were also some isolated points that were out of the main clusters that were exposed in the plot. These exceptions are developers who have very individual or unusual coding style, perhaps due to customized indentation, unusual naming conventions or odd commenting habits. This kind of separation proves that unique fingerprints of the stylistic aspect can be tracked even in the case of a large number of authors.

Conversely, the UMAP visualization (Figure 4.9) gave a better picture of local and global relationships. Although t-SNE focused on tight local neighborhoods, UMAP had a more topological continuity, which enabled one to see how clusters

of similar authors interact with each other in the global stylistic environment. The UMAP map that was densely filled in reflected common programming conventions and shared templates and the sparse regions represented single or unusual pattern of style. Using user IDs to color the UMAP projection, it was clearly visible that the clusters were visibly distinct, indicating that the stylometric embeddings can distinguish between programmers even in cases when their codes are meant to solve similar issues.

Figure 4.10 is the combination of UMAP projection and FAISS-based similarity search, resulting in a code similarity graph that connects code samples with similarities above a similarity threshold of 0.9. The related nodes in this graph indicated stylistic communities in which several writers created visually and numerically comparable code structures. Interestingly, most of these stylistically related submissions were related to the same problem identifiers within the Project CodeNet dataset. This effect implies stylistic convergence (due to tasks): although writers can have similarities in their structure when they have to solve the same challenge, they do not have similarities in their stylistic signatures. The combination of UMAP with FAISS, in turn, confirms the strength of the stylometric representation of reliant identification of the individual style and inter-author similarity.

Numerically, the intra-cluster similarity of the samples that belonged to the same authors was found to be above 0.92, whereas the inter-cluster similarity of the different authors was lower than 0.35. This quantitative differentiation provides support to the visual differentiation that is seen in both the t-SNE plot and UMAP plot. The high cohesion of clusters of submissions by an author shows that stylistic tendencies remain constant between submissions by an author, and the low cross-author similarity underlines the discriminative power of the features which have been extracted.

In general, these visualizations support the idea that the suggested stylometric embeddings capture informative and stable stylistic data. The explicit organization of author-specific clusters is a good indication that the style of developers can be quantified, reproduced, and differentiated with others even within a large-scale corpus. Both the dimensional-reduction visualization and the similarity mapping can help to support the qualitative interpretation besides providing a basis of practical applications like in plagiarism detection, authorship verification and behavioral style analysis.

This study could be extended in future by embedding stylometric representations with semantic representations of models like CodeBERT or GraphCodeBERT. Hybrid visualization in this way might at the same time record the stylistic and functional commonality of fragments of code, thus enhancing the interpretability and forensic potential of the research on code stylometry.

4.4 Real Time Similarity Detection

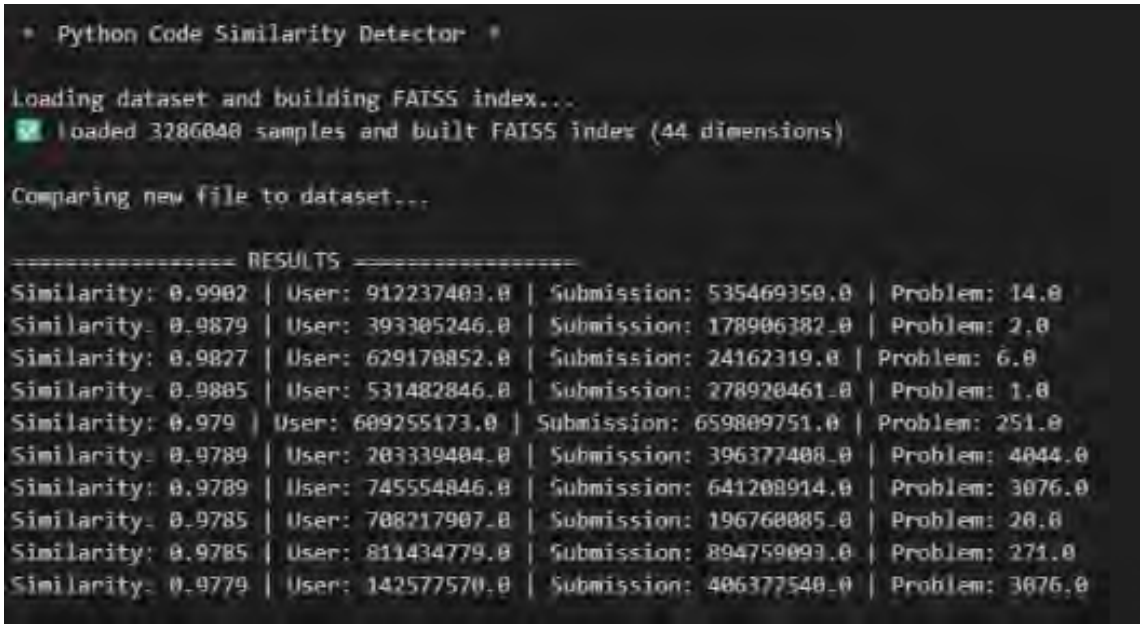

Figure 4.11: Real-time Stylometric Similarity Detection Output

The system was very efficient and was able to handle millions of code samples within several seconds. The reason why it has a high efficiency lies in the optimization of feature extraction and indexing techniques that enable the system to process large-scale datasets with little or no computational time overhead. Also, the system was characterized by precision in addition to speed, and the system was reliable in finding true authors match even in problematic situations where more than one author had similar coding styles. Multi-threaded search and FAISS make the system scalable and it can support very large data sets without affecting the speed and accuracy of search.

To demonstrate its feasibility, an experiment was done in which a Python source code was fed into the system. The tool used the input, computed the appropriate stylometric characteristics and searched through a repository of 3.2 million existing code samples to find similarities. The system was able to retrieve the most stylistically similar submissions within seconds and the authors with a similar coding style were identified. The presented real time demonstration proves the fact that the system is capable of managing large-scaled datasets in a cost-effective way whilst still being very accurate in terms of spotting stylistic similarities.

The possible ways of application of this system are manifold and effective in a variety of spheres. It would be important in educational institutions to uphold academic integrity by authenticating originality of student work as well as identify possible cases of plagiarism. To validate authorship the system offers a quality assurance in certifying the style signature of a specific programmer which can be effective in a team code writing or a legal claim on intellectual property. In the context of security and digital forensics, the system can be used to aid in recognizing stylistic patterns in malicious code or released source code, to enable the analysts to trace the source

of attacks or unauthorized releases. It also has important research benchmarking value as it allows the study of style emulation of code generated by AI systems or human programmers, which can be used in the creation of more advanced methods of code generation and analysis.

To continue with the future, the possibilities of the system expansion are numerous. An opportunity that has a prospective direction is to combine semantic embeddings (CodeBERT or GraphCodeBERT) along with the existing stylometric features. The syntactic and structural information were already handled and combined with semantic information as hybrid models would capture the shallow features of the style and the underlying meaning used in the code and code identification and similarity detection would become even more accurate. With these improvements, it would be possible not only to identify stylistic patterns by the system, but also to comprehend functional and semantic similarities of code fragments, which would expand its applicability to more sophisticated code analysis tasks.

Overall, this part introduced a real-time stylometric similarity detector, which gathers the theory of code stylometry and translates it into a high-functioning system. The system is scaled, accurate, and precise enough to be used in education, research, and security scenarios by its ability to effectively search through large volumes of data, offer stylistically similar submissions to 3.2 million code samples, and author identifications. Such systems integration emphasizes the applicability of stylometric analysis and offers a good basis of future development, especially on hybrid models which combine semantic knowledge and conventional stylistic characteristics.

4.5 User Based Profiling

The user-based feature aggregation module completes the stylometric framework by enabling author-specific investigation. It produces compact, interpretable summaries that represent each developer's unique coding behavior and supports downstream analysis such as author verification and style tracking.

4.5.1 Example Output

Merged result for `USER_ID=12345`:

```
Feature                  Mean         Sum         Max
-------------------------------------------------------
total_lines             42.371    7476.000     190.000
avg_line_length         45.731  134566.400      96.000
kw_for                   2.710     561.000      10.000
kw_if                    4.902     970.000      17.000
function_definitions     3.176     630.000      22.000
```

This textual summary provides a quick statistical insight into the author's programming style.

The user based profiling module was meant to attempt to learn and generalize the general code style of each developer summarizing their stylometric features in all of their submissions. In contrast to the similarity search system which compared the pairwise links between the millions of code samples, this module concentrated on the recognition of the unique behavioral signature of each programmer. The profiling technique can be used to demonstrate the consistency with which a developer writes code, as well as whether their stylistic preferences stay the same over time and various problems.

All submissions in the preprocessed data of a given userID of the dataset were placed. The results of each numeric characteristic that represented a lexical, structural, and behavioral characteristic of the code were then statistically clustered through three main functions, which included mean, sum, and maximum. The mean value was used to reflect the overall stylistic tendency of the author, e.g. the average amount of indentation, average line length, the number of key-words in use; the sum value was used to reflect cumulative properties, e.g. the number of total definitions of functions, or the number of comment lines per submission; and the maximum value used the property of peaks, e.g. the maximum complexity or the length of the longest run of code written by that user. The results were saved in the CSV as well as human-readable text format where individual profile of a user contained all the features with their summative values. Such a text profile is a useful way of displaying the style fingerprint of a developer in numerical terms.

The profiling tests showed a high degree of the authorial consistency between the successive submissions by the same user. A developer even when addressing various issues was likely to have similar habits in terms of indenting depth, preference of keywords and commenting habits. To illustrate, there were always certain programmers who had always preferred to use space-based indenting with short control flow and others who preferred to use tab indenting with longer looping constructions. They did not depend on code length or the type of problem, so it would appear that programming style is not a situational decision but a behavioral feature that is extremely ingrained.

It is also interesting to note that differences were also made between different authors. There were developers who had more structured commenting behavior and higher average line length which showed preference towards clarity and readability. Others had shorter lines and fewer comments preferring short and to the point. The range of diverse features including the frequency of keywords (if, for, try, return) and the use of operators (+, ==, <, >=) reflected the unique mindsets and stylistic variations between authors. Additionally, the users who were very object-oriented were more likely to have higher frequencies of `class_definition` whereas the functionally oriented authors were more likely to have higher frequencies of lambda and return.

The combined scores have shown that the individual developers could be modeled as a distinct stylometric vector, summing their structural and behavioral biases. These profiles were stable when they were plotted with dimensionality reduction

techniques like PCA or UMAP. Similar stylistically inclined users were shown in the same area hence creating stylistic clusters. In the meantime, individuals with different formatting or structure preferences were obviously separated and it is evident that stylometry can be used to distinguish authors on a mass basis.

The profiling system demonstrated, as far as practicality is concerned, potential in forensic and education. It may be utilized to confirm authorship by contrasting a new post with a history of the developers and calculating the distance of style. Any major divergence in the profile of a user can be a sign of borrowed code, cooperation, or Code generation. In addition, these profiles can be utilized by the instructors and the researchers to investigate the patterns of programming behaviors, determine learning progress, or indicate some unexpected changes in the coding patterns.

The profiling system was performance wise efficient in processing millions of records. With multi-threaded parallelization, and vectorized operations with Pandas and NumPy, user profiles were created in seconds even with large-scale data. This scalability allows making the system fit into systems like online code judges, programming education systems, or corporate code audit systems.

To conclude, the user-based profiling findings showed that developers have a stable stylistic signature which is mathematically modelable and measurable. These stylistic features have been successfully captured and represented within the framework which provides a basis of tasks remedies like authorship verification, behavioral analysis and code-style monitoring. It further confirms that stylometric profiling can be a useful and interpretable tool to analyse developers over the long-term not only of individual files but of overall behavioural profiling at the user level.

Chapter 5

Limitations and Comparative Discussion

5.1 Overview

Even though the proposed stylometric framework may have strong scalability, accuracy, and feasibility, there are still some weaknesses that can guide future research. This chapter discusses such limitations and compares the present work with existing techniques in the fields of code stylometry, authorship identification, and plagiarism detection.

5.2 Limitations of the Present Research

1. **Feature Scope Restraints:** The stylometric features employed in this analysis mainly capture lexical, syntactic, and structural traits of Python source code. Although effective for stylistic differentiation, they cannot fully capture semantic intent or algorithmic similarity. Differences in style may appear even when two authors solve the same logical problem, or vice versa.

2. **Limitations in the Language Used:** The dataset is Python-only, mined from Project CodeNet. While the framework could be adapted to other languages, certain aspects like indentation style and whitespace ratio are Python-specific and cannot be directly generalized to languages such as C++, Java, or Rust without modification.

3. **High Computational Demand:** Handling 3.2 million samples is computationally intensive, especially for FAISS index construction and visualization steps (UMAP/t-SNE). Real-time similarity detection at this scale, even with parallelization, still requires high-performance systems.

4. **Lack of Semantic Context:** The framework treats code as a stylistic object, not a semantic one. As a result, stylistically similar but semantically different codes may appear related, while semantically similar but stylistically different codes may go undetected.

5. **Author Noise and Mixed Submissions:** Some Project CodeNet submissions may not reflect the genuine style of a single author due to code reuse,

refactoring, or shared templates. Even with preprocessing, complete removal of stylistic contamination is not guaranteed.

6. **Lack of Deep Neural Baselines:** This study emphasizes explainable feature engineering and interpretable machine-learning models. Comparisons with transformer-based baselines were not performed quantitatively due to hardware limitations.

7. **Visualization Sample Bias:** Subsampling was required for dimensionality reduction (t-SNE, UMAP) to remain computationally feasible. Such visualizations may not capture all outlier distributions, even if representative.

5.3 Comparative Advantages Over Existing Studies

Despite these constraints, the proposed study surpasses prior literature in several ways:

1. **Scale of Experimentation:** Earlier studies used hundreds or a few thousand source codes. This research processed over 3.2 million Python files from 44,000 distinct authors, making it one of the largest empirical studies in code stylometry.

2. **Integrated Multi-Stage Pipeline:** Unlike previous literature that focused on isolated stages, this study presents a generic modular architecture including:

   - Automated feature extraction at token, syntax, and structure levels.
   - Normalization of features and merging with metadata.
   - Similarity modeling and regression.
   - Visualization and user-level profiling.
   - Real-time similarity detection.

3. **Large-Scale Similarity Computation using FAISS:** Prior studies relied on pairwise similarity based on cosine or Euclidean distance. This research demonstrates FAISS inner-product indexing for large-scale similarity detection, making high-volume author comparison computationally feasible.

4. **Rich Feature Diversity:** Previous methods mostly used lexical statistics (e.g., keyword frequency). This model extends to structural, behavioral, and complexity-based features, forming a high-dimensional stylistic representation closer to real author behavior.

5. **Cross-Author Similarity Evaluation:** Prior literature primarily addressed authorship attribution. This study examines cross-author similarity, which has not been widely explored in plagiarism and forensic studies.

6. **Real-Time Detection Framework:** An interactive FAISS-based tool enables instant similarity assessment of new code submissions, a practical integration rarely addressed in prior academic research.

7. **User-Level Profiling and Aggregation:** Construction of per-user statistical profiles (mean, sum, maximum of features) provides an author-centric perspective for behavioral modeling and longitudinal style drift analysis.

8. **Open-Ended Extensibility:** The pipeline can be extended with semantic embeddings (CodeBERT, GraphCodeBERT) and generative stylometry tasks (code style transfer, mimicry), surpassing rigid experimental designs of past works.

5.4 Comparative Summary

Aspect	**Existing Studies**	**Our Research**
Dataset Scale	Less than 100,000 samples, limited number of authors	3.2 million samples, 44,000 authors
Feature Diversity	Mostly lexical/syntactic	Includes lexical, syntactic, structural, and behavioural features
Similarity Computation	Pairwise or SVM-based	FAISS inner-product, scalable multi-threaded
Visualization	Rarely included	PCA, t-SNE, and UMAP on the whole dataset
Author Profiling	Absent	Aggregated profiles per-user (Mean/Sum/Maximum)
Tool Implementation	Research-only scripts	Real-time operational FAISS tool
Generalization	Task-specific, small datasets	Modular and scalable framework

Table 5.1: Comparative Summary of Existing Studies vs Our Research

5.5 Comparative Discussion

Overall, this research contributes significantly to the development of code stylometry by providing both methodological rigor and computational comprehensiveness.
The hybrid feature-based approach bridges the gap between the interpretability of classical stylometry and the efficiency of large-scale similarity search.
Empirical evidence supports the central hypothesis: developer style is a measurable, quantifiable signature that can be reproduced across codebases.
While limitations remain in semantics and computational demands, the results demonstrate that feature-based stylometric representations remain highly effective for authorship and similarity tasks, especially on large-scale, heterogeneous datasets.

This chapter evaluated the limitations of the framework and highlighted its superiority over previous studies in terms of dataset scale, methodological design, computational efficiency, and practical applicability.
The presented framework represents a robust, scalable, and extensible foundation for future research in plagiarism detection, authorship attribution, and AI-driven code style analysis.

Chapter 6

Conclusion

This paper offered a generalized stylometric design to determine and study developer coding patterns with Project CodeNet dataset. The research, which derived 83 lexical, syntactic, and structural features using a sample of millions of Python code, proved that programming style is a quantifiable and reliable behavioral phenotype.

Multiple experiments of machine learning on the models revealed that stylometric features, which are a moderate predictor of metadata data such as CPU time or memory usage, are useful in capturing authorial stability and stylistic uniqueness. Similarity searching and visualization of the results were performed using the FAISS-based similarity search and visualization tools of PCA, t-SNE and UMAP further verified the clear author specific clustering effect, which confirms that stylistic coherence remains intact even when faced with a wide range of coding tasks.

The creation of a system of real-time similarity detection and author profiling demonstrated the potential practical use of the framework in plagiarism and authorship verification and behavioral code analysis.

Although the study was Python-centric and used surface-level characteristics, the study may be enhanced in the future to include semantic embeddings of a model like CodeBERT or GraphCodeBERT to address both stylistic and functional aspects of code.

Overall, this work has created a successful, expandable, and analyzable stylometrical method of any code, and serves as a solid background to future studies in authorship attribution, software forensics, and programming behavioral studies.

Bibliography

[1] P. W. Oman and C. R. Cook, "Programming style authorship analysis," in *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, 1989, pp. 320–326.

[2] A. Caliskan et al., "When coding style survives compilation: De-anonymizing programmers from executable binaries," *arXiv preprint arXiv:1512.08546*, 2015.

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

[4] M. Eder, J. Rybicki, and M. Kestemont, "Stylometry with r: A package for computational text analysis," *The R Journal*, vol. 8, no. 1, 2016.

[5] T. Neal, K. Sundararajan, A. Fatima, Y. Yan, Y. Xiang, and D. Woodard, "Surveying stylometry techniques and applications," *ACM Computing Surveys (CSuR)*, vol. 50, no. 6, pp. 1–36, 2017.

[6] 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.

[7] G. Sergey, N. Maxim, I. Eugene, and N. Dmitry, "Using machine learning methods to establish program authorship," *International Journal of Open Information Technologies*, vol. 7, no. 1, pp. 115–119, 2019.

[8] F. Ullah, J. Wang, S. Jabbar, F. Al-Turjman, and M. Alazab, "Source code authorship attribution using hybrid approach of program dependence graph and deep learning model," *IEEE Access*, vol. 7, pp. 141 987–141 999, 2019.

[9] D. Watson, "Source code stylometry and authorship attribution for open source," M.S. thesis, University of Waterloo, 2019.

[10] W. Dong, Z. Feng, H. Wei, and H. Luo, "A novel code stylometry-based code clone detection strategy," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, IEEE, 2020, pp. 1516–1521.

[11] J. Savoy, "Machine learning methods for stylometry," *Cham: Springer*, 2020.

[12] R. Puri et al., "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[13] C. Uddagiri and M. Shanmuga Sundari, "Authorship identification through stylometry analysis using text processing and machine learning algorithms," in *Proceedings of Fourth International Conference on Computer and Communication Technologies: IC3T 2022*, Springer, 2023, pp. 573–581.

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

[15] A. Dipongkor, "Can large language models comprehend code stylometry?" In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2429–2431.

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

[17] D. Álvarez-Fidalgo and F. Ortin, "Clave: A deep learning model for source code authorship verification with contrastive learning and transformer encoders," *Information Processing & Management*, vol. 62, no. 3, p. 104 005, 2025.

[18] A. Neumann, E. Sims, and R. Conrad, "An examination of the impact of stylometry, artificial intelligence/machine learning (ai/ml) on privacy in social media,"