Generating Developer Portfolio based on Coding Behaviour using GitHub Mining and Machine Learning

by

Tanjina Faria
24341261
Mazbha Ul Haque
24341266
S.M Hasnan Monir
24341278
Ashiqur Rahman Turzo
20301389

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
February 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:

Tanjina Faria
24341261

Mazbha Ul Haque
24341266

S.M Hasnan Monir
24341278

Ashiqur Rahman Turzo
20301389

Approval

The thesis/project titled "Generating Developer Portfolio based on Coding Behaviour using GitHub Mining and Machine Learning" submitted by

1. Tanjina Faria (24341261)
2. Mazbha Ul Haque (24341266)
3. S.M Hasnan Monir (24341278)
4. Ashiqur Rahman Turzo (20301389)

of Fall, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on February 1, 2025.

Examining Committee:

Supervisor:
(Member)

Md. Aquib Azmain
Lecturer
Department of Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Dr Md Golam Rabiul Alam
Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi
Associate Professor & Chairperson
Department of Computer Science and Engineering
Brac University

Abstract

The widespread use of open-source software repositories, such as Github, provides a rich data source for analyzing developer coding behavior. Manually curating developer portfolios to analyze their skills, contributions, and coding patterns would take a lot of time and be challenging. So, we will create a framework to automatically generate developer portfolios by mining Github repositories and applying machine learning techniques to model classify developer skills and predict work patterns.. The framework makes use of Github API data like commit history, pull requests, metrics of code complexity and collaboration patterns to extract features that help identify the developer coding styles and abilities. Machine learning algorithms, including clustering and classification, will use their technical strengths, domain background and quality of code to segment developers. The goal of producing these portfolios is to assist recruiters, project managers, and developers in better understanding the skill sets that exist within the network. As well as improve career growth strategies and identify opportunities for collaboration.

Keywords: GitHub mining, machine learning, developer portfolio, coding behavior analysis

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In this new era of involving Artificial Intelligence, the development landscape is changing faster than ever. The retirement of a software developer position is changing, and the list of skills required to get a developer job is growing increasingly larger [28]. In the current situation, developers are judged not only by their technical skills but also by their ability to showcase their work effectively in real-life projects [9]. To express this ability, a developer needs a good portfolio, as it plays the most important role in securing a job. Traditionally, portfolios have been manually created, which often requires a significant amount of time and effort [46] . Often, a skilled developer without corporate experience struggles to create a portfolio that is acceptable to job recruiters. Moreover, recruiters have limited means to verify whether an applicant is being truthful in their portfolio [14] . To address these issues, there should be an autonomous way to judge and verify a developer's skills [19]. This would allow developers to focus on the technical aspects rather than spending hours creating a portfolio . On the other hand, those who are willing to hire people for job will be able to select best candidates for the interviews allowing them to spend less time going thorough developer porfolios one by one manually. By mining their GitHub data, it is possible to extract meaningful insights into a developer's strengths, areas of expertise, and even soft skills like teamwork and communication [34] [27]. Machine learning models can be trained to analyze coding patterns, recent project involvement, and evaluate the quality of contributions. There are multiple reasons why generating developer portfolios based on technical skills, work patterns, and contribution metrics using GitHub mining and machine learning is essential in this era. Firstly, manual portfolio creation can be time-consuming, and it can be difficult for new developers to create an industry-standard portfolio. Secondly, as a job recruiter, verifying portfolio data can be time-consuming when there are many applicants, and verifying their requirements is challenging considering the variety of tech stacks in the development area. For example, if a company needs a MERN stack developer, a portfolio generated based on a developer's GitHub commits can easily indicate whether they are proficient in MERN or not. Lastly, developers tend to hide their inabilities and highlight their strengths when they manually create their portfolios. An autonomous system that creates portfolios based on their GitHub data certainly solves this issue.
The stakes are high in the era of software-driven innovation across industries, espe-

cially now with artificial intelligence. Developers today are more productive than ever with the help of AI. With the power of GitHub mining and machine learning, we can unlock a new dimension of understanding about developer skills, work patterns, and technical contributions in a new way.

1.2 Rationale of the Study or Motivation

The machine learning approach to creating developer portfolios is not an entirely new concept. However, while reviewing papers related to this topic, we have found that most of the GitHub mining-based research is outdated, and very little research has been published after 2020. We believe we can contribute to this space, as not many people have worked on it recently. Additionally, we have not found any paper that is exactly the same as our work. We think we can contribute a little to this area while learning a thing or two about machine learning.

1.3 Problem Statement

In today's software development landscape, developers are increasingly evaluated based on their coding skills, contributions, soft skills, and overall collaborative performance on platforms like GitHub. A structured developer portfolio makes sure a developer is able. Even when given a developer's GitHub profile, we have to "browse carefully and patiently" to judge the ability of a developer. When a company receives hundreds of job applications with submitted portfolios, it becomes almost impossible to go through every application and judge accurately. A machine learning based portfolio generator can be used to solve this problem. This study aims to use GitHub mining and machine learning methods to create an intelligent system that can generate detailed and accurate developer portfolios based on their coding behavior on their own GitHub public repositories. This research study intends to answer the following research questions with these aims.
RQ1: In what ways can GitHub data be mined for better understanding of developer expertise, behavioral patterns, and skill proficiency?
RQ2: Which algorithms and models can be used to analyse the extracted features and identify the patterns which may help in determining a developer's strengths and weaknesses?
RQ3: How can the insights obtained from the machine learning model be utilized to create a working portfolio?

1.4 Objective

The main aim of our study is to develop a system that generates developer portfolio-making predictions based on the mining of their coding behaviors using GitHub and machine learning techniques. We will try to implement a method to extract relevant data from Github repositories for the first instance. We first have to look into GitHub API, web scraping techniques, and other mining techniques.
The goal of such a metric is to collect detailed information about a developer's coding style, his contribution to the open-source community and how he collaborates with

others. In order to ensure that our system can handle large amounts of data, we need to build a well-structured data extraction pipeline.
The coding behaviors will be analysed through repository data. It involves evaluating supervised learning models meant for classifications, as well as unsupervised models meant for recognizing patterns in the data. Following this, we will endeavour to develop a portfolio that is visually appealing based on the output from GitHub mining machine and learnings analysis. An effective architecture portfolio must showcase a specific theme that resonates with the architect's work. We also want to properly validate the portfolios that we generate. The research objectives will help us in developing a new system that speeds up both developer and recruiter processes with a translator that gives clear cuts on developer skills.

1.5 Methodology in Brief

The methodology for this research can be divided into four key parts: Data Collection, Data Preprocessing, Machine Learning Analysis, and Portfolio Generation.

1.5.1 Data Collection

The data will be collected from GitHub. We will utilize the GitHub API to extract publicly available repositories, commit history, and issues. Additionally, we will employ web scraping tools if necessary.

1.5.2 Data Preprocessing

Cleaning and structuring the raw data to remove noise and missing values is necessary to train a machine learning model. Different GitHub repositories use different programming languages and patterns, and some repositories may not contain any code at all. These aspects need to be taken into consideration during data preprocessing.

1.5.3 Machine Learning Analysis

Supervised and unsupervised learning models will be implemented to generate insights into a developer's growth over time, technical skills, and soft skills like collaboration and communication.

1.5.4 Portfolio Generation

We will develop a dynamic and visually appealing portfolio template that highlights the insights extracted from our machine learning model.
Finally, we will validate the generated portfolios to ensure that the model is producing usable and accurate portfolios.

1.6 Scope and Challenges

The scope of this research is to design and implement a machine learning model that can intelligently provide developers with a portfolio based on their GitHub

profile. This will provide insights into what the developer is capable of, helping both developers and job recruiters. Developers will not need to waste hours creating a standard portfolio, and recruiters will be able to judge applicants more easily. However, the main challenges include working with datasets containing various programming languages. The software development field has many tech stacks, and JavaScript alone has many frameworks. This will create challenges, as we may need to classify different programming languages for training, which requires significant manual work. Data accessibility can also be a challenge, as many developers tend to keep their GitHub repositories private. Data quality and noise can be major issues, as not every developer is skilled, and not every repository contains quality code. Another challenge is that people love to customize their own things. While our research aims to create a system that automates the process of portfolio generation, some developers may prefer to create their portfolios manually. Additionally, bias in the machine learning model may not accurately represent the overview of a developer's GitHub profile.

Chapter 2

Literature Review

2.1 Review of Existing Research

Liang et al. [34] looked at how to identify open-source software (OSS) skills by analyzing activity data from GitHub. Even though OSS development involves many different abilities, there aren't many tools that can spot and measure these skills effectively. To address this, the authors created **DISKO** (Detecting Skills in OSS) a system that translates measurable GitHub activities, called *signals*, into clear skill indicators [34]. The goal is to help recognize what contributors are good at so teams can form more easily, mentors can be matched, and project roles can be assigned more accurately [34].

Their research used a mix of literature review and real-world data analysis [34]. First, they reviewed previous studies to figure out which OSS skills matter most and what activities reflect them [34]. After that, DISKO collected data from GitHub through the GraphQL and REST APIs and GHTorrent, which was used to calculate the skill scores of contributors [34]. DISKO's results were compared to self-evaluations of 455 OSS contributors [34] to understand the functioning of the system. The results proved that DISKO result is accurate and its success rate is noted between 77% and 97% [34]. The results also showed that the ability to welcome newcomers or remain active in local efforts counted just as much as any technical skill [34]. Over 50% of contributors said they would be willing to share their skills publicly if they were detected by DISKO [34]. The tool was also especially effective for JavaScript and Python languages and common OSS practices. One limitation of DISKO is that it finds it difficult to discriminate between the medium and high proficiency. The authors conclude that the signal-weighting scheme employed in DISKO may need future refinement [34].

According to authors DISKO may enhance contributors' visibility, support team building and foster mentorship while enhancements may arise from community feedback and altering how different signals are weighted [34]. The experts also acknowledge a number of challenges, which include: reliance on self-reported skills creates a bias, it is more complex to define and measure soft skills, and showing detection transparency is key for users' trust [34].

To sum up, the paper presents a practical method for extracting both technical and social skills from GitHub activity, showing good accuracy, and indicates ways for further validation and improvement [34]. The writers warn that because DISKO was designed around GitHub, its design may miss contributions made outside of GitHub.

Also, they say that results will vary depending on the projects and communities, and thus broader testing should be done to confirm the generalizability [34].
The authors of the paper "Mining the Usage of Reactive Programming APIs: A Study on GitHub and Stack Overflow" authored by Zimmerle et al., have presented a research idea regarding Reactive Programming (RP) APIs, and more specifically the Reactive Extensions (Rx) libraries. Polyglot in nature, Rx Libraries support multiple languages and libraries. This study was undertaken to find out how much the developers are using Rx libraries in open source projects. The study was conducted by means of mining Github repositories and Stack Overflow posts.
According to the work from [38], the authors mined GitHub repositories to analyze Rx operators and for mining, they used GitHub API. They focused on repositories with 10 or more popular libraries, from these three stars and also the most popular libraries: RxJava, RxJS, and RxSwift. They mined a total of 2670 repositories. Using a script, the operators in the mined repositories were searched. The search used regular expressions to identify operator invocations. Filters were applied to reduce false positives in the results. For Stack Overflow mining, Zimmerle et al. used Latent Dirichlet Allocation (LDA) to identify topics from posts. They mined a total of 47,404 posts related to RxJava, RxJS, and RxSwift. Topics of the posts were considered based on popularity and difficulty, and the popularity of a post was measured using average views, favorites, and scores of the posts.
After mining, Zimmerle et al. found that the majority of Rx operators are used in open-source projects. RxJava showed 94.1% usage, RxJS 100%, and RxSwift 98.5%. Overall, 95.2% of operators are used across all libraries. The most used operators are subscribe, map, just, and create. Library-specific variants are the least used operators. 23 topics were identified from Stack Overflow posts, and most of them were related to Stream Abstraction (36.4% of posts), including Stream Manipulation and Stream Creation and Composition. iOS Development had the longest median time to receive an accepted answer, indicating it is a challenging topic. After combining the Stack Overflow posts and GitHub repository results, the researchers found that the most common operators were subscribe, map, and filter. The paper [38] focused on the most common Rx operators so that developers can focus on them and also provided an idea of GitHub and Stack Overflow post mining. The topics that were identified in Stack Overflow posts could help developers discover common challenges and solutions in RP. Additionally, there are some operators that are rarely used, which gives an idea of what operators API developers should focus on and reduce the API surface. There are some flaws in the research, such as the regular expressions possibly giving false positives, and the focus on GitHub and Stack Overflow, which may not represent the entire developer community. The paper [38] provides a comprehensive idea about Rx operators in open-source projects, GitHub and Stack Overflow post mining, and the challenges faced by developers.
The paper [32] begins with how capturing and analyzing real data is very necessary in software organizations as it helps to understand the quality of code written by the programmers. The authors, Ardimento et al., said that analyzing the code depends on the development process and is not formally specified and also costly and complex. So, they proposed an incremental process mining method to model and understand how programmers interact with integrated development environments (IDE) and version control systems (VCS). The authors were particularly interested in novice programmers, and their goal was to give feedback on their programming tasks and

analyze their coding behaviors.
The paper initially discusses different types of process mining, such as Petri Nets, which use a graphical representation of the process, and declarative process mining, which describes processes as sets of constraints. However, the authors chose the WoMan framework for their study because it is incremental, declarative, and can easily handle complex processes. In this framework, conformance can be checked, new workflows can be discovered, and the next action in the process can be predicted. In the WoMan framework, log elements are in a 7-tuple $\langle T, E, P, C, A, N, R \rangle$ representing timestamp, event type, process instance, case identifier, activity, execution count, and result, respectively.
Subsequently, Ardimento et al. used a framework consisting of three main components: the log processor, log exporter, and WoMan miner tool. The log processor is a plugin used in an integrated IDE to capture all human-computer interaction events, including high-level and low-level events, and then create an XML file of it. The log exporter captures the logs and makes them suitable for the WoMan miner. Finally, the WoMan miner tool uses the logs to discover and refine process models of programmers' coding behaviors. The authors conducted the research with six second-year students. They were given a simple software task to implement in Java using the Eclipse IDE, equipped with the log processor plugin.
After giving the project to the students, Ardimento et al. found significant behavioral differences. For example, 70% of students executed the development sessions, meaning most of them did not check the effect of their code changes. Also, only 1% of sessions involved students using debugging tools. The authors analyzed the learning curve and found that most of the students showed stable coding behaviors after 10 sessions, while only two students developed new behaviors throughout the project. There were some limitations, such as the sample size being limited to six students and the language being Java, which generalizes the overall result. Also, the outcome may have been influenced by both existing Java proficiency among participants and external factors such as student motivation.
This paper [24] by Xiong et al. presents an approach to understand developer behavior by linking and analyzing data from two trusted and most used software communities, GitHub and Stackoverflow. The authors proposed a method to track developers across GitHub and Stackoverflow using machine learning, natural language processing, and statistical analysis.
The authors raised three questions:

- Do a developer's activities in GitHub reflect their activities in Stackoverflow?
- Are topics developers concern about relevant in both GitHub and Stackoverflow?
- Which activities in Stackoverflow are more relevant?

To find the answers, the paper [24] begins with the problems that can arise when attempting this task. Email and username are usually used for uniquely identifying users across the internet. During the research process, they were unable to get the users' email addresses from stackoverflow which made the task difficult to track and link users. Therefore, to solve this the researchers came up with the idea of using Classification and Regression Tree (CART) decision tree to link accounts across these two platforms based on similar features like username, user behavior, and

writing styles. This approach later proved to be effective, as it outperformed previous methods like TBIT and Bird's method. The authors achieved higher precision and F-Score by mining developer behaviors using their own method. They used T-Graph analysis, LDA-based topic clustering, and cross-tagging to identify developers behaviour.
Firstly, the authors extracted the data to train the model. Then, they used four-string matching algorithms to find similarities between GitHub and Stackoverflow usernames. Levenshtein distance and longest common subsequence were selected for the final model. The authors also extracted writing style features like length, vocabulary richness, digits, punctuations, and frequently used words. They constructed a dataset by manually linking GitHub and StackOverflow accounts with evidence to verify that they belong to the same person. The authors' method achieved a higher F-Score (0.75) compared to TBIL (0.718) and Bird's method.
The experiment results show that:

- Developers who are active in committing issues on GitHub are also likely to commit the same issue on Stackoverflow.
- The topics that developers are concerned about are similar to those who engage in Stackoverflow.
- Developers' concerns mainly revolve around the projects they are committed to in their GitHub profile.

This paper [24] contributes to the field of software development by linking developers' profiles across GitHub and Stackoverflow and provides valuable insights into the interconnected nature of software development and knowledge-sharing activities.
According to the paper [35], Pina et al has developed a GitHub mining tool called Sonarlizer Xplorer that mines public GitHub repositories to find technical debt in the code base using a tool called SonarQube. The paper in [35] describes GitHub Xplorer which is a tool for mining repositories whereas Sonarlizer is for analyzing the mined code using SonarQube. SonarQube is an open-source platform to inspect codebases to detect bugs, smells, and security vulnerabilities in over 27 programming languages. The authors emphasized the need for large datasets in software engineering research, particularly when working with code analysis. To address this requirement, Sonarlizer Xplorer was developed to gather vast amounts of data to find technical debt at scale.
Subsequently, Pina et al. described that the tool was running for over four months to collect data from almost 57 million GitHub repositories, with 609,884 of them being Java projects. Although only 45,994 of them were analyzed successfully. Java codes need specific environments, configuration files, and versions to compile before they can be analyzed, which explains the low success rate.
In the model training part, the authors of the paper [35] described that the most common approach to predicting patterns in machine learning is clustering. For this project, they collected data and split it into 80% training and 20% validation. They also applied data validation and repeated the steps multiple times to avoid bias in the training data.
The overall process of Sonarlizer Xplorer is as follows: GitHub Xplorer uses the GitHub API to discover repositories and stores the URLs of repositories in a MongoDB database. Then, Sonarlizer queries this database and clones the repositories

into the local machine in order to analyze them. After cloning the repository, the Sonarlizer tool performs a SonarQube analysis. SonarQube has its own set of rules to identify technical debt. If there is any Java code in the repository that needs to be compiled, it first compiles it before analyzing it. It stores the information in PostgreSQL database after identifying technical debts. The commit list in the repository will also be examined by GitHub Xplorer to find contributors.
In addition, the authors state that the extensibility of the tool allows end users to add further data from the GitHub API. The paper [35] ends by pointing out future improvements like adding repositories from other platforms like GitLab and BitBucket. It could transform Sonarlizer Xplorer into an invaluable resource for scrutinizing patterns and behaviors around technical debt.
According to paper [11], there are great promises as well as perils when one thinks about mining Github, which is probably one of the most famous software platforms that programmers can see today. According to the authors, an online survey was conducted with more than 240 developers participating, and a manual inspection of a random sample of 434 projects was performed. Heuristics were applied to identify merged pull requests to avert some threats.
The authors mentioned the promises and perils of mining GitHub in this paper [11]. They talked about "fork and pull" model of GitHub to enhance community engagement with the open source community. Furthermore, a public API is provided by GitHub which can be used for development practices and understanding the network structure of GitHub.
Conversely, the writers also investigated the hazards of extracting information from GitHub. Most of the repositories in GitHub are personal and single committer most of the time. The report stated that 14% of the repositories in GitHub are used just as a cloud storage and not for software. These are mostly used to store data or related to the project. In addition, they noted that almost 40% of all pull requests do not show as merged, which could cause incorrect assumptions about the project's activity. Almost 50% of GitHub registered users don't have a public activity at all. They neither maintain a public repository nor contribute to any open-source project. Another challenge that is mentioned deals on the limits of the Github API which does provide all the required research data which makes it unavailable. Moreover, alterations in features and interface of GitHub can cause inconsistencies in data structures. Furthermore, many active users of the platform would make use of third party tools for issue tracking and internal communication which poses a blockage to data collection for research.
According to Kalliamvakou et al. (2015), mining Github for software engineering research brings many opportunities, but also poses serious challenges. Researchers can improve the validity and reliability of any research work by understanding the dangers and adjusting methods accordingly.
This study has tried to address the roles played by elite developers in OSS projects concerning the type of activities performed by them over time and the way different types of evolving activities impact different project results. The paper carried a methodology that discusses an empirical strategy to research the elite developer.
The methodology involved project selection, data preparations, identification of elite developers, categorization of developer activities, measurement of project outcomes, and statistical analysis. 20 large OSS projects were selected from GitHub; of those, 11 were company-sponsored, while the remaining 9 were non-company-sponsored.

The projects had to be sizable and popular so a high number of commits and contributors, the data of public repositories had to be available, and structured development processes such as pull requests and issue tracking needed to be applied. In the identification part, developers were identified as elite in case they had write permissions on the repository and the activities of elite developers were compared with those of non-elite developers. During categorizing, activities were divided into four parts, i) Communicative Activities (issue discussions and commit comments, ii) Organizational Activities (task assignments), iii) Supportive Activities (documentation and branch management), and iv) Typical Activities (writing and reviewing code). The measurements in the project outcomes were done based on productivity and quality metrics. The data analysis was done using Panel Data Econometric Models, in which the various relationships of activities of elite developers with project outcomes were analyzed; Fixed-effects regression models, to account for project-specific factors; and finally, ANOVA tests and Tukey's HSD test were applied to examine how elite developers' activities changed over time [27].
This research encompassed around three questions:

- RQ1: What do elite developers do in addition to contributing typical technical code in OSS projects? [27]
- RQ2: How do an OSS project's elite developers' effort distributions evolve along with the growth of the project? [27]
- RQ3: What is the relationship between an OSS project's elite developers' effort distributions and the project's outcomes in terms of productivity and quality? [27]

The results to these questions were:

- **RQ1 (Elite Developers' Activities):** Elite developers are very active concerning secondary development activities, such as communication, documentation, and project management. In the majority of projects, elite developers are responsible for more than 50% of all activities. The supporting and organizational activities grow over time. [27]
- **RQ2 (The Evolution of Elite Developers' Activities):** In the project evolution, elite developers start to take more care about the communication and supporting activities and reduce their coding activities. Statistical tests confirm the hypotheses about the decreasing technical and increasing administrative and coordination activities, which often reduces the actual amount of time available for development work. [27]
- **RQ3 (Elite Developers' Activities' Impacts on Project Outcomes):** Elite developers' activities on productivity are all negative. When elite developers spend more time on communicative and supportive tasks, the number of new commits decreases. Company-sponsored projects face a larger drop in productivity compared to non-company-sponsored projects in this case. Contrary to that, elite developers have mixed effects on quality: increased time spent on supportive tasks improves bug fix rates, while too much focus on communication correlates with lower project quality. [27]

The study also has its limitations. One of these is the developers' role shift: the switch from coding to non-coding tasks affects both developer motivation and retention, although this has barely been studied so far. Other gaps include non-elite developers' contribution influence: while elite developers are considered in the present research, there is a failure to understand how non-elite contributors fill up the technical holes left by these elite developers.
The paper [31] focuses on how developers use GitHub Actions and how their adoption affects repository activities. The study has found that while only a few repositories use GitHub Actions, their impact is generally positive. It leads to more rejected pull requests, and the number of commits in merged pull requests went down. There was also no significant impact on merge time.
Lastly, among the paper's limitations are the absence of cost and performance analyses and that the study solely depends on the discussions from the GitHub issue and does not factor in the experience, interviews, and opinions from the developers themselves.

2.2 Summary of Key Findings

According to Liang et al., the DISKO tool achieves an accuracy of 77% to 97% while identifying skills from GitHub data. Soft skills can be easily detected by DISKO, but it struggles to detect high and medium-level skills.
In the paper [38] titled "Mining the Usage of Reactive Programming APIs," the authors discovered that Reactive Programming (RP) APIs are mostly used in open-source projects. Around 95.2% of operators are used properly. On Stack Overflow, developers mostly post about stream manipulation, indicating it is the most challenging topic. The authors also stated that since the study was limited to GitHub and Stack Overflow, it may not represent the entire developer community.
According to Xiong et al., developers that frequently have problems with their GitHub projects, and therefore post a great number of GitHub Issues, tend to be asking a large amount of questions on Stack Overflow. Typically, the type of projects developers work on in GitHub are the same as the topics they write about in Stack Overflow, suggesting that both are in the same area of technology. However, over time, developers' Stack Overflow questions evolve to reflect their current GitHub projects. Furthermore, while a developer's GitHub work and Stack Overflow answers make his/her skills apparent; neither their questions nor their comments are an indication of their ability or skill.
Generally, novice developers do not take into account how their changes will affect their code. In addition, most developers exhibit consistent behavior across all 10 development sessions. The WoMan framework described by Ardimento et al. is effective for incremental process mining. The SonarLizer Xplorer can effectively mine GitHub data. A group of researchers mined data using it and found that the SonarLizer Xplorer could easily analyze 57 million GitHub repositories. In addition, the SonarLizer Xplorer is flexible and allows the user to include additional data retrieved from the GitHub API.
Wang et al. found that 50% of open source (OSS) projects were completed by elite developers. These elite developers began the project coding, and as the project grew, they transitioned into a management/administrative role. This reduced the productivity of the elite developers but increased the bug fix rate. Sponsoring a

project with a company resulted in lower productivity from the coders compared to when the project was sponsored by someone other than a company.
A significant advantage of GitHub is that there is a wealth of data available for researchers studying software engineering, but much of this data is either private or for use outside of software. Also, since 40% of the pull requests submitted do not appear as merged on GitHub, researchers may draw incorrect conclusions.

Chapter 3

Requirements, Impacts and Constraints

3.1 Ethical Issuses

Our work involves collecting data personal data from GitHub. Although it might sound unethical, but we are working with only publicly available data by GitHub's official API. User may not use their original name, email, location in GitHub and also there is option to hide repositories from public viewing which we can not access or did not access during our research. According to GitHub's privacy policy, publicly available data on GitHub is intentionally shareable by design [33]. Optional details like username, bio, photo, job title, and location, Code, files, issues, pull requests, and commit histories in public repos and commit metadata falls under this policy. But if a user explicitly ask not to use their data from their GitHub profile, it is better to avoid [33]. From our findings, it is possible to find pattern of commits made by user [11]. Timing of commits may expose user's daily habit or expose their real location. But our research does not include any of these.

3.2 Economic Analysis

Our research work can be used as a timing saving measure by both job candidates and job recruiter. A job recruiter can mass produce portfolios with in minutes and see the expertise of the applicants using our system [28]. As it uses data directly from GitHub, it can provide a good insight about an applicant's proficiency, eliminates the manual digging through repositories, outdated resumes, or self-reported statistics [18]. Our system shows actual contribution across the repositories, consistency over a period of time, collaboration pattern which may play a key role in corporate job, code adaption by others from stars, forks and watches, and creates a behavioral pattern. Job applicants may use it to find out their weakness or the missing piece required from their GitHub profile, which they can identify and work on it [14].

3.3 Limitations

Our system can only analyse data from public GitHub resources. By design, the GitHub platform and API expose only information that users have made public and

do not provide unrestricted access to private repositories or other private metadata of third parties [44]. Consequently, if a developer has a substantial amount of work stored in private repositories, those commits, pull requests, and related contribution signals will not appear in the generated portfolio, which may lead to an incomplete or potentially biased representation of their activity.

In principle, an individual developer could supply a personal access token and explicitly authorise access to their own private repositories, thereby mitigating this limitation for self-generated portfolios. However, if a recruiter or third party uses our system without such explicit consent, developers whose strongest work resides in private repositories may be systematically disadvantaged, as their portfolios would under-represent their true activity and skills.

Chapter 4

Work Plan

Firstly, we collected dataset from public repositories. In order to collect the dataset, we made a python GUI which can collect user data we need. Then we merged all users' data into one single file to create dataset, cleaned our datasets for quality data and went for data preprocessing. Thirdly, we created a train, test and validation set for the models. After that, we selected and implemented machine learning models to develop a system which can provide a detailed insight of a GitHub profile. Finally, we used the data provided by our trained model to create a portfolio. We made a tool to easily create and manage portfolios.

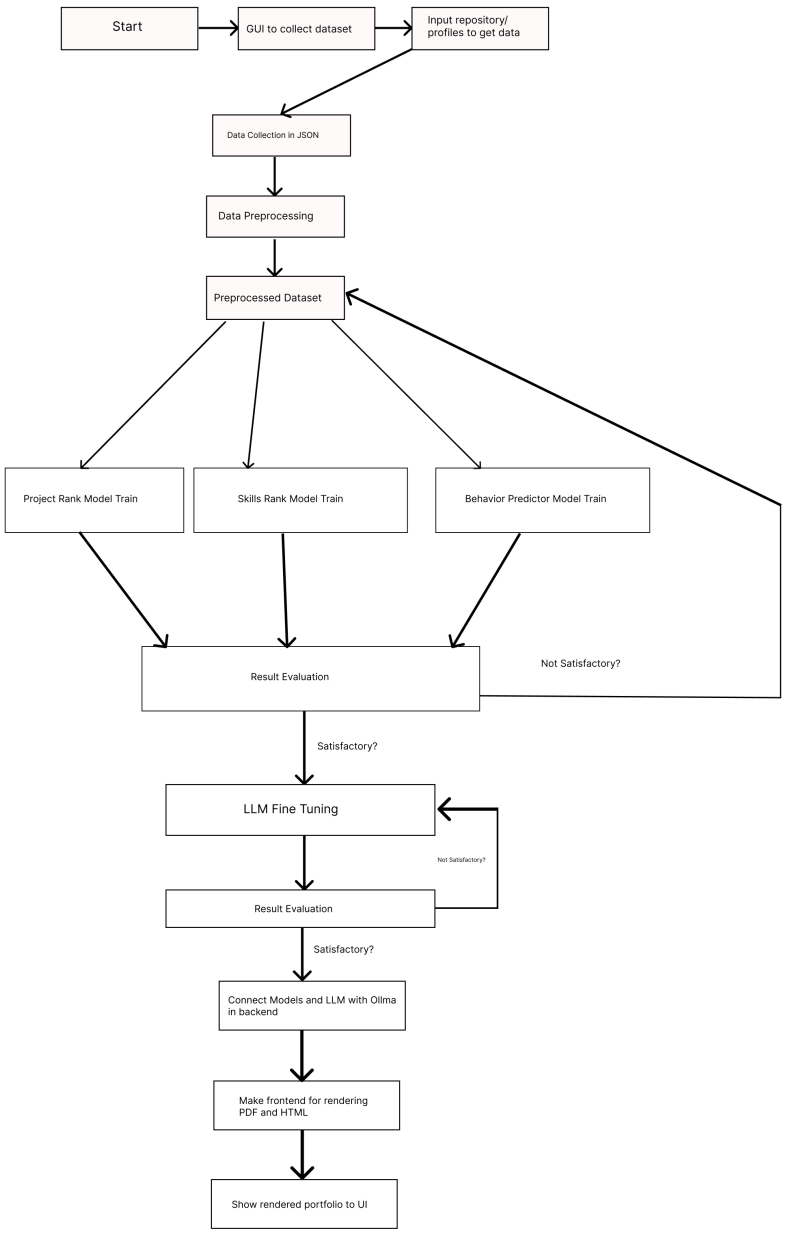

Figure 4.1: System workflow diagram showing the complete process from data collection to portfolio generation

Chapter 5

Dataset and Feature Engineering

5.1 Source and Acquisition

In order to collect the dataset, a Python-based GUI tool was developed. The aim of the GUI was to gather data from GitHub's official API. The GUI has two sections: the first section where if we input the URL of a user's profile then it would fetch all available data from that user's profile. And the second section of the GUI is if we input the URL of a public repository then first it will fetch the list of the contributors of that repository and then it will fetch the data of all contributors' profiles. A repository contributor is a user in this case. And a user has a profile, and the profile has all the information. Our target is to collect the users' individual data not the repository itself. Official repositories of React JS, JavaScript, Python, Laravel, Vue JS, Tailwind, TensorFlow, Nuxt JS, Astro JS and others were used to collect the dataset.
Under the hood the GUI uses GitHub's official API. GitHub's official API sends back the request data in GraphQL format which would be very complex to work with so we formatted the data in a different way. Large-scale empirical studies on GitHub often rely on dedicated infrastructures such as GHTorrent, which continuously mirror GitHub's public API into a research-oriented database [7]. In contrast, here we query the live GitHub GraphQL API through a custom Python GUI to collect a focused set of developers and repositories.
Two primary transformation processes were implemented for the JSON exports: a user-level consolidation and a repository-level flattening. Together, these processes convert heterogeneous API responses into normalized, analysis-ready tables.

- **User-level consolidation.** The user-level process aggregates, validates, and harmonizes profile and activity information for each developer. It standardizes account metadata (e.g., identifiers, timestamps, location/company fields), reconciles social-graph counts (followers/following), and compiles contribution activity (commits, pull requests, issues, reviews). To support downstream analysis, it also derives exposure-aware indicators such as account age (in days) and rate-based summaries (e.g., recent contributions and code-change statistics). Convenience ratios (e.g., following-to-followers) and portfolio-normalized averages (e.g., stars per repository) are computed to mitigate heavy-tailed distributions and profile sparsity.

- **Repository-level flattening.** The repository-level process constructs one

record per repository referenced in the exports. It resolves ownership and visibility, attaches popularity metrics (stars, forks, watchers), captures primary language and language-byte summaries, and preserves maintenance/security signals (releases, deployments, vulnerability alerts). When multiple repository sources exist (e.g., owned, contributed-to, started/starred, or forked lists), they are deduplicated and merged into a unified schema keyed by repository identity.

5.2 Limitations in GitHub's Response Data

The landscape of publicly available datasets for analyzing developer behavior presents several challenges:

- **Limited API Access and Rate Constraints:** GitHub's API restrictions limit the depth and breadth of data collection, particularly for granular commit-level details, private repository activities, and comprehensive temporal histories. This results in incomplete snapshots of developer behavior [11].
- **Proxy Label Dependency:** Existing datasets typically rely on proxy metrics (stars, forks, followers) as indicators of developer skill or influence, which may not directly correlate with actual coding proficiency, problem-solving ability, or real-world impact. These metrics can be influenced by factors unrelated to technical competence (e.g., social networking, repository visibility, topic popularity).
- **Static Snapshots vs. Dynamic Behavior:** Most datasets provide point-in-time snapshots rather than longitudinal data, making it difficult to capture learning trajectories, skill evolution, and temporal patterns in developer productivity.
- **Data Sparsity and Missing Values:** Real-world GitHub data contains significant missing information (private repositories, deleted content, incomplete profiles), requiring robust imputation strategies and careful feature engineering to handle gaps.

How We Addressed These Limitations

1. **Comprehensive Feature Engineering:** We engineered a comprehensive set of features spanning activity patterns, collaboration metrics, language diversity, repository lifecycle stages, and productivity indicators, transforming raw metrics into meaningful behavioral signals.
2. **Temporal Bias Mitigation:** All activity-based metrics were normalized by account age (e.g., commits per year, repo creation rate) to ensure fair comparison between developers with different GitHub tenure, preventing bias toward older accounts.
3. **Temporal Validation Strategy:** We implemented temporal train-test splits using account creation dates to simulate real-world deployment scenarios and prevent data leakage, ensuring the model's predictive validity on future data.

4. **Multi-dimensional Behavioral Modeling:** Rather than relying on single metrics, we captured developer behavior across multiple dimensions (productivity, consistency, collaboration, multitasking, language diversity) to create holistic profiles.

5.3 Dataset Description

Dataset Scope and Size

The users dataset comprises 7,378 unique developers described by 69 raw features from GitHub's API. After feature engineering (described in Section 5.4), these are transformed into model-specific feature sets: 39 features for behaviour classification, 43 features for skills classification, and 28+ repository-level features plus the full 43 user-level features for ranking.

User-level Features

- **Identity & Profile:**

  username, login, profile_url, database_id, name, email, avatar_url, bio, company, location, website_url, twitter_username, pronouns

- **Account Timestamps & Exposure:**

  created_at, updated_at, fetched_at, account_age_days

- **Social Graph & Affiliations:**

  followers_count, following_count, watching_count, organizations_count, sponsors_count, sponsoring_count, sponsorships_as_sponsor_count, sponsorships_as_maintainer_count, following_to_followers_ratio

- **Repository Summary:**

  total_repositories, starred_repositories_count, forked_repositories_count, total_started_repos, total_forked_repos, total_languages, stars_per_repo, forks_per_repo

- **Contributions - Lifetime & Annual:**

  has_any_contributions, has_activity_in_past, has_restricted_contributions, total_contributions, contributions_this_year, total_commit_contributions, total_pr_contributions, total_pr_review_contributions, total_issue_contributions, total_issues, total_commit_comments, total_issue_comments, total_discussion_comments, total_gist_comments, total_recent_prs, total_recent_issues, total_recent_comments

- **Recent Code-Change Statistics:**

  total_recent_additions, total_recent_deletions, total_recent_files_changed, unique_repos_committed, avg_additions_per_commit, avg_deletions_per_commit, avg_files_per_commit

- **Security & Gists:**

  public_keys_count, gists_count

Repository-level Features

- **Identity, Ownership & Visibility:**

  repo_id, name, name_with_owner, owner_login, url, visibility, is_private, is_fork, source

- **Timestamps & Maintenance:**

  created_at, updated_at, pushed_at

- **Popularity & Community Signals:**

  stargazer_count, fork_count, watchers_count, releases_count, deployments_count, vulnerability_alerts_count

- **Language Profile & Size:**

  primary_language, languages_total_count, languages_total_size

- **Description:**

  description

5.4 Feature Engineering

From the raw features, we derive a compact set of portfolio-oriented features that summarize a developer's behaviour in ways that are intelligible to both humans and models. Each indicator is a transparent transformation of observable counts, using add-one smoothing (to avoid division by zero), per-account-year normalization (to account for tenure), log transforms for heavy-tailed popularity measures, and quantile-capped min-max scaling (to prevent outlier dominance). The features are organized into six dimensions: activity, technical skills, collaboration & community, project quality & maintenance, influence & leadership, and behavioural patterns so that the resulting portfolio can highlight how much a developer contributes, how they code, and how the community responds.

A. Activity Features

Activity intensity score. The aggregate volume of contributions across commits, pull requests, issues, and PR reviews, computed row-wise:

$$\text{activity\_intensity\_score} = \Sigma\{\text{commits, PRs, issues, PR reviews}\}$$

Contribution consistency. Let

$$a = (a_1, a_2, a_3, a_4)$$

denote the four activity channels (commits, PRs, issues, PR reviews). The code first forms an inverse coefficient-of-variation like ratio and then maps it to $(0, 1]$:

$$r = \frac{sd(a)}{1 + \text{mean}(a)}, \quad \text{contribution\_consistency} = \frac{1}{1 + r}$$

Repository creation rate. New repositories per account-year:

$$\text{repo\_creation\_rate} = \frac{\text{total\_repositories}}{\left(\frac{\text{account\_age\_days}}{365} + 1\right)}$$

Recent activity ratio. Current-year contributions relative to lifetime:

$$\text{recent\_activity\_ratio} = \frac{\text{contributions\_this\_year}}{1 + \text{total\_contributions}}$$

Code change rate. Average of additions and deletions per commit:

$$\text{code\_change\_rate} = \frac{\text{avg\_additions\_per\_commit} + \text{avg\_deletions\_per\_commit}}{2}$$

Development velocity. Recent code churn normalized by the count of repositories actually touched:

$$\text{development\_velocity} = \frac{\text{total\_recent\_additions} + \text{total\_recent\_deletions}}{\text{unique\_repos\_committed} + 1}$$

Multitasking score. A blended index that combines breadth and coverage:

$$\text{multitasking\_index} = 0.5 \times \text{breadth} + 0.5 \times \text{coverage}$$

Breadth. Cohort-wise min-max normalization of the number of unique repositories the developer contributed to in the window:

$$\text{breadth} = \text{minmax01}(U)$$

where U is the count of distinct repositories with at least one commit/PR by the developer.

Coverage. The fraction of the developer's active repositories that they touched in the window:

$$\text{coverage} = \frac{U}{A + 1}$$

where A is the number of the developer's repositories that were active in the window (the +1 prevents division by zero).

B. Technical Skills Features

Tech stack breadth. Proxy by total distinct languages seen for the user:

$$\text{tech\_stack\_breadth} = \text{total\_languages}$$

Language specialization. Dominance of the top primary language across the user's repositories:

$$\max p_i \quad \text{where} \quad p_i \text{ is the proportion of repos in language } j \text{ for user } i$$

Language balance (diversity). Simpson's diversity:

$$1 - \Sigma p_i$$

Repository language diversity. Count of distinct primary languages across the portfolio.

Average languages per repository. Mean of feature `languages_total_count` aggregated by owner.

C. Collaboration & Community Features

Community engagement score. Sum of issue, PR review, discussion, and commit comments.
Collaboration ratio. Issue + PR contributions relative to commits:

$$\text{collaboration\_ratio} = \frac{\text{total\_pr\_contributions} + \text{total\_issue\_contributions}}{\text{total\_commit\_contributions} + 1}$$

Fork contribution rate. Share of forked repositories in the user's repo count.
Mentorship score. Reviews and helpful comments per account-year:

$$\text{mentorship\_score} = \frac{\text{total\_pr\_review\_contributions} + 0.5 \times \text{total\_gist\_comments}}{\left(\frac{\text{account\_age\_days}}{365} + 1\right)}$$

Network influence. Log-transformed, weighted combination of followers, stars, and forks:

$$\text{network\_influence} = \log\left(1 + 2 \times \text{followers} + 0.5 \times \text{stars} + 1.5 \times \text{forks}\right)$$

Social coding index. Robustly scaled engagement per account-year, using a 90th-percentile cap and a $\log(1 + x)$ transform.

D. Project Quality Features

Repository active score. Fraction of repositories that are currently active:

$$\text{repo\_active\_score} = \frac{\text{active\_repos}}{\text{total\_repositories} + 1}$$

Showcase score. Pinned repositories scaled by the GitHub UI cap:

$$\text{showcase\_score} = \frac{\text{pinned\_items\_count}}{6}$$

Maintenance score. Decays with days since last push:

$$\text{maintenance\_score} = \frac{1}{1 + \frac{\text{avg\_days\_since\_last\_push}}{30}}$$

Code review participation (and index).

$$\text{code\_review\_participation} = \frac{\text{PR reviews}}{\text{PRs} + 1}$$

A composite `code_review_index` further blends intensity-per-year with this ratio.

E. Influence & Leadership Features

Reputation score. Weighted log-sum of followers, stars, forks, and sponsoring signals (weights $0.3, 0.3, 0.2, 0.1, 0.1$).

Impact factor. Stars and forks per repository:

$$\text{impact\_factor} = \frac{\text{total\_stars\_received} + 2 \times \text{total\_forks\_received}}{\text{total\_repositories} + 1}$$

Influence growth rate. Followers accumulated per account-year.
Viral repository score. Peak repository reception relative to the developer's own average:

$$\text{viral\_repo\_score} = \frac{\text{max\_stars\_repo}}{\text{avg\_stars\_per\_repo} + 1}$$

F. Behavioral Pattern Features

Maintainer score. Share of started (original) repositories among started+forked:

$$\text{maintainer\_score} = \frac{\text{total\_started\_repos}}{\text{total\_started\_repos} + \text{total\_forked\_repos} + 1}$$

Team player score. Blend of capped organization count, collaboration ratio, and fork contribution rate:

$$\text{team\_player\_score} = 0.35 \times \text{org\_norm} + 0.45 \times \text{collab\_norm} + 0.20 \times \text{fork\_rate}$$

Here,

$$\text{org\_norm} = \text{Take organizations\_count, apply } \log(1+x);\ \text{winsorize at the training set's 5th/90th percentiles; min-max scale to } [0,1].$$

$$\text{collab\_norm} = \text{Compute } \frac{\text{PR} + \text{issues}}{\text{commits} + 1}\text{; winsorize at 5th/90th percentiles (training); min-max to } [0,1].$$

$$\text{fork\_rate} = \frac{\text{forked\_repositories\_count}}{\text{total\_repositories} + 1}\text{; already in } [0,1),\ \text{no further scaling.}$$

Generalist score. Robustly scaled breadth of languages.
Work consistency. Stability of stars across repositories, computed as an inverse variance ratio:

$$\text{work\_consistency} = \frac{1}{1 + \frac{\text{stars\_std\_dev}}{\text{avg\_stars\_per\_repo} + \varepsilon}}$$

Learning velocity. Product of breadth and recentness:

$$\text{learning\_velocity} = \text{generalist\_score} \times \text{recent\_activity\_ratio}$$

Innovation index. Geometric blend of originality (started repos) and reception (avg stars):

$$\text{innovation\_index} = \sqrt{\text{started\_n} \times \text{avgstars\_n}}$$

Here, `started_n` = normalized number of original ("started") repositories for the developer. `avgstars_n` = normalized average stars per repository for the developer.

Feature Engineering Pipeline

Although the previous subsections define each engineered indicator individually, in practice the transformation from raw GitHub exports to the final modelling matrix follows a reproducible four-stage pipeline. This pipeline is applied once to the consolidated user-level table and repository-level table, and its output is reused across all three downstream models (behaviour classifier, skills classifier, and ranking model).

Step 1: Raw data extraction and tabularisation

Raw signals are first extracted from the GitHub GraphQL API and converted into normalized, analysis-ready tables as described in Sections 5.3. At the user level, this includes:

- Account metadata and temporal exposure (created_at, account_age_days)
- Social-graph statistics (followers_count, following_count, organizations_count, sponsors_count, sponsoring_count)
- Contribution counts (total_commit_contributions, total_pr_contributions, total_issue_contributions, total_pr_review_contributions, total_contributions, contributions_this_year)
- Summary portfolio descriptors (total_repositories, total_started_repos, total_forked_repos, total_languages)
- Recent code-change statistics (total_recent_additions, total_recent_deletions, unique_repos_committed, avg_additions_per_commit, avg_deletions_per_commit, avg_files_per_commit)

At the repository level, each record captures ownership and visibility (owner_login, is_private, is_fork), popularity (stargazer_count, fork_count, watchers_count), maintenance timestamps (created_at, updated_at, pushed_at), and language byte summaries (primary_language, languages_total_count, languages_total_size).

Step 2: Repository-to-user aggregation

The repository table is then aggregated back to the user level to obtain portfolio-level statistics. Grouping by owner_login, the code computes:

- popularity aggregates such as mean and maximum stars and forks per repository (avg_stars_per_repo, max_stars_repo, avg_forks_per_repo, max_forks_repo) and an overall public_repo_ratio
- language distribution statistics (per-user counts of primary languages, repo_language_diversity, and the mean number of languages per repository avg_languages_per_repo)
- portfolio size and activity proxies, including the total number of repositories associated with a user and the fraction that are currently active, which combine with user-level active_repos to form repo_active_score.

These aggregates provide a compact summary of each developer's portfolio, while still allowing the models to distinguish between, for example, a single highly starred project and a large number of moderately popular repositories.

Step 3: Derived behavioural, technical, and influence features

Building on these aggregates, the pipeline applies the feature definitions detailed in Sections 5.4 A-F.
In the implementation this is organized as a sequence of modular functions – create_developer_activity_features, create_technical_skills_features, create_collaboration_features, create_project_quality_features, create_developer_influence_features, and create_behavioral_patterns – applied in order to the combined user-repository table.
Representative examples include:

- **Activity and productivity**: activity_intensity_score (total volume of commits, PRs, issues, and reviews), repo_creation_rate (new repositories per account-year), recent_activity_ratio, code_change_rate, development_velocity, and a blended multitasking_score that combines breadth and coverage of active repositories.
- **Technical skills**: tech_stack_breadth (total distinct languages), language_specialization (dominance of the primary language), language_balance (Simpson diversity), repo_language_diversity, and avg_languages_per_repo.
- **Collaboration and community**: community_engagement_score (sum of issue, PR-review, discussion, and commit comments), collaboration_ratio ((PR+ issue) contributions relative to commits), fork_contribution_rate, mentorship_score, social_coding_index, and network_influence.
- **Behavioural patterns**: maintainer_score, team_player_score, generalist_score, work_consistency, learning_velocity, and innovation_index, which together operationalise the behavioural dimensions used in the multi-task modelling framework.

Step 4: Normalisation and robust scaling

Many of the above features are expressed as rates per account-year or ratios bounded to the $[0, 1]$ interval by design, using add-one smoothing to avoid division by zero. Heavy-tailed popularity and engagement metrics (e.g., followers, stars, forks, sponsorships) are transformed with $\log(1+x)$ before being combined into composite scores such as reputation_score and network_influence. For several high-level indices (e.g., generalist_score, social_coding_index, learning_velocity, innovation_index), the implementation further applies quantile-capped min-max scaling, where values above the 90th percentile are clipped before rescaling to $[0, 1]$, in order to reduce the influence of extreme outliers.
The resulting matrix of engineered numerical features undergoes preprocessing through model-specific pipelines that apply median imputation for missing values and standardization for feature scaling, prior to training the ranking, behaviour, and skills models.

Preprocessing and Special Handling

Before model training, all features undergo rigorous preprocessing:

- **Time Handling**: All datetime fields (createdAt, updatedAt, pushedAt) are converted to timezone-naive UTC before calculating age or recency metrics.
- **Missing Data**: Numeric missing values are filled with 0; strings with empty strings; division-by-zero is protected via $\max(\text{value}, 1)$ in denominators.
- **Language Aggregation**: Language signals are extracted from both primaryLanguage.name (weighted $3\times$) and languages.edges (weighted by log of byte size) to create robust language_diversity and tech_stack_breadth metrics.
- **Scaling**: All features are median-imputed and variance-standardized (without mean-centering) to preserve sparsity and ensure compatibility with XGBoost's histogram algorithm.

Chapter 6

Machine learning model design and target construction

This chapter defines three prediction tasks that, taken together, form a complete view of each developer's portfolio. The first task produces a quantitative ranking using regression. The second predicts technical skills as a multi-label classification problem. The third models behavioural patterns, also as a multi-label classification problem. For each task, we describe how the targets are constructed, which models are used, how they are trained, and which hyperparameters are chosen.

6.1 Multi-Task Prediction Framework

The system uses a multi-task prediction setup in which three related tasks are built on top of the same feature representation. The aim is to measure developers on various aspects consistently, rather than relying on single metrics [21]. The three tasks are:

- Task 1: Project Ranking - a regression problem that predicts a continuous repository impact score summarizing each project's visibility, activity, and community engagement on GitHub.
- Task 2: Technical Skills Classification - a multi-label classification problem that identifies the programming languages, frameworks, and technology areas in which the developer shows evidence of proficiency.
- Task 3: Behavioural Pattern Classification - a multi-label classification problem that predicts high-level working styles and contribution patterns (e.g., maintainer, team player, innovator, learner).

The three models are trained separately, but they all use the same engineered feature space introduced in chapter 5. In practice, this means that features are computed once and then reused across all tasks. Sharing the feature pipeline makes the system more efficient at inference time and allows each model to be optimized for its own objective without re-designing features from scratch.
This framework reflects the fact that a single reputation score is not enough to describe a developer. A highly ranked developer may be very strong in one or two specific technologies, while someone with a lower overall reputation may have

a broader but shallower skill set. Behavioural patterns also add another largely independent dimension: they show how a developer works, rather than simply what they know or how visible they are. When it comes to real-life recruitment and matching scenarios, users are often interested in combinations of these views; for example, "Python developers with high reputation who also exhibit team-player behaviour." [46]

6.2 Task 1: Project Ranking

6.2.1 Repository Impact Score Construction

The project rank parameter expresses each GitHub repository's impact as a scalar continuous variable that summarizes its overall visibility, technical substance, and maintenance level in the ecosystem. This variable serves as the regression target for the project ranking models. It combines seven repository-level indicators, each weighted according to its contribution to perceived project impact: stars, forks, watchers, total code size, language diversity, total commits, and recent activity.
Formally, we first construct a set of normalized components. For each raw metric $x \in \{\texttt{repo\_stars}, \texttt{repo\_forks}, \texttt{repo\_watchers}, \texttt{repo\_lang\_size}, \texttt{repo\_lang\_count}, \texttt{total\_commits}, \texttt{recency\_score}\}$ we apply a robust normalization:

- convert to numeric and replace missing values with 0,
- clip values to the 90th percentile of the training cohort,
- divide by this clipped maximum so that the resulting value lies in $[0, 1]$.

Intuitively, this treats the 90th percentile as "full credit" and compresses only the most extreme outliers, while preserving discrimination among the majority of projects. Let

- $\texttt{stars\_norm}$ = normalized star count,
- $\texttt{forks\_norm}$ = normalized fork count,
- $\texttt{watchers\_norm}$ = normalized watcher count,
- $\texttt{lang\_size\_norm}$ = normalized total code size,
- $\texttt{lang\_count\_norm}$ = normalized number of languages,
- $\texttt{commits\_norm}$ = normalized total commits,
- $\texttt{recency\_norm}$ = normalized recent-activity score.

The raw project impact is then defined as a convex combination of these seven components: $\text{impact}_{\text{raw}} = 0.30 \times \texttt{stars\_norm} + 0.20 \times \texttt{forks\_norm} + 0.10 \times \texttt{watchers\_norm} + 0.15 \times \texttt{lang\_size\_norm} + 0.10 \times \texttt{lang\_count\_norm} + 0.10 \times \texttt{commits\_norm} + 0.05 \times \texttt{recency\_norm}$. Because all components are in $[0, 1]$ and the weights sum to 1, $\text{impact}_{\text{raw}}$ also lies in $[0, 1]$. For interpretability, we map this to a convenient 0–100 scale:

$$\text{impact\_score} = 100 \times \text{impact}_{\text{raw}}.$$

The robust normalization is specifically chosen for heavy-tailed GitHub metrics such as stars and forks, where a small number of "viral" projects can have extremely large values. By clipping at the 90th percentile and scaling, the score remains sensitive to differences among typical projects while preventing a single highly popular repository from dominating the scale. This yields an impact measure that is less susceptible to outliers and more stable under incremental changes to a project.
The weighting scheme prioritizes community validation and reuse over purely structural properties. Stars (0.30) capture how many users explicitly endorse or bookmark the repository, making them the dominant signal of perceived usefulness. Forks (0.20) measure code reuse and experimentation, indicating that others build on top of the project. Watchers (0.10) reflect sustained interest in the project's evolution over time.
Technical substance and complexity are incorporated through total code size (0.15) and language diversity (0.10). Larger, multi-language repositories are more likely to represent substantial systems or libraries, but these signals are kept secondary to community reaction. Development effort is captured via total commits (0.10), rewarding projects with sustained contribution histories without allowing raw activity volume to dominate the score. Finally, recent activity (0.05) ensures that actively maintained projects receive a modest bonus compared to older but inactive repositories.
Interpreting the normalized score as a pseudo-percentile on a 0–100 scale allows us to qualitatively segment projects by impact. In our experiments, projects with impact scores above 80 typically correspond to flagship repositories with strong community traction and sustained maintenance. Scores between 80–60 indicate well-established, actively used projects; scores between 60–40 correspond to moderately visible repositories with some community uptake; and scores below 40 tend to represent niche, experimental, or low-activity projects that are still building their footprint in the ecosystem.

6.2.2 Model Selection and Architecture

Two complementary regression algorithms were implemented to predict repository impact scores, representing both gradient boosting and deep learning approaches to structured data modelling.

Feature set: The project ranking models operate on a numeric feature matrix $\mathbf{X}_{\text{rank}}$ built from the engineered portfolio features introduced in Chapter 5, together with the activity metrics used in the impact-score construction. Starting from the GUI-derived user- and repository-level tables , the feature engineering pipeline produces six families of indicators—activity, technical skills, collaboration & community, project quality & maintenance, influence & leadership, and behavioural patterns. These compact indicators are stored in the `model_features_shortlist` file and summarise how each developer works and how the community responds to them. For project ranking, this engineered feature set is combined with the seven robustly normalised components that define the repository impact score: stars, forks, watchers, total code size, language count, total commits, and recent activity. In the implementation, these seven raw components (`repo_stars`, `repo_forks`, `repo_watchers`, `repo_lang_size`, `repo_lang_count`, `total_commits`, `recency_score`) are used to

construct the impact target and are then dropped from $\mathbf{X}_{\text{rank}}$ to avoid leaking the label definition back into the inputs. After removing identifier columns and the impact target, each project (developer profile with its aggregated portfolio context) is represented by a d-dimensional numeric vector, allowing the models to learn how the broader behavioural feature set predicts the seven-component repository impact score.

Model 1: XGBoost Gradient Boosting Regressor

XGBoost is chosen as the main regression model thanks to its excellent empirical performance on structured heterogeneous feature spaces where it is not impacted by feature scaling or moderate multicollinearity [17]. The model leverages the `XGBRegressor` class, with the objective specified as regression with squared error. This choice ensures that the squared error between predicted and target ranking scores is minimized directly. The creation of a tree involves the use of a histogram-based algorithm that combines continuous features into discrete features using the binning process. In this way, the training can be done for medium to large tabular datasets.

The model has been fitted with 600 boosting rounds and a max tree depth of 6 to ensure the ensemble captures rich non-linear feature interactions whilst limiting the individual trees to stop over-specialisation. A learning rate of 0.05 ensures that each tree makes a small fix thereby promoting optimization and improving the generalization of the trees. To improve the ensemble regularization and minimize variance, XGBoost uses a stochastic subsampling at both the instance and feature levels: for every tree, 90% of the training examples are sampled and 90% of the features are to be considered on growing each tree. A fixed random seed is used to ensure reproducible training and evaluation.

The XGBoost regressor is embedded in a scikit-learn preprocessing and modelling pipeline. All the numerical features shall first go through a process known as median imputation. As the name suggests, all the missing entries across numerical features shall be filled with the median value of the particular feature computed on the training data. This robust approach reduces the effect of the outliers on the input values.

The imputed features are then standardized in terms of variance using scikit-learn's standardization routine without mean-centering, a choice that is compatible with the sparse matrices produced by the column transformer. The preprocessing pipeline is fitted exclusively on the training split and subsequently applied to validation and test data, thereby preventing information leakage. Overall, the chosen configuration balances model capacity and regularization: a relatively deep ensemble with 600 trees and depth 6, combined with a conservative learning rate and subsampling, yields a flexible yet stable model for project ranking.

Model 2: Neural Network (MLP) Regressor

As a complementary non-linear baseline, a feedforward neural network (multi-layer perceptron, MLP) is trained to predict the same ranking target from the engineered feature set. The network operates on the full preprocessed feature vector and learns a dense representation through stacked fully connected layers with dropout regularization. The architecture consists of an input layer of dimension ddd, corresponding

to the number of numerical features, followed by three hidden layers. The first hidden layer contains 256 ReLU units, followed by a dropout layer with rate 0.2, which randomly deactivates 20% of the units during training to reduce co-adaptation and mitigate overfitting [12]. The second hidden layer reduces the representation to 128 ReLU units, again followed by dropout with the same rate, encouraging the model to focus on salient feature interactions. A third hidden layer with 64 ReLU units further compresses the latent representation into a compact form that feeds into the regression head. The
output layer is a single linear unit, which produces an unbounded scalar prediction for the ranking score, as appropriate for a regression task.
The network is trained using the Adam optimizer with a learning rate of 1×10^{-3} [15]. The loss function is mean squared error (MSE), which penalizes the squared deviation between predicted and true ranking scores; during training, root mean squared error (RMSE) is tracked as a more interpretable metric in the same units as the target. Training proceeds for up to 200 epochs, with early stopping employed to prevent overfitting. An early-stopping callback monitors the validation RMSE and terminates training if this metric does not improve for 12 consecutive epochs, restoring the model weights corresponding to the best validation performance [2]. Mini-batch training is performed with a batch size of 256, and 20% of the training data is reserved as a validation subset used exclusively for early-stopping decisions. Prior to being fed into the neural network, the input features undergo a separate preprocessing pipeline. Missing values are handled using median imputation, as for the tree-based model. The imputed features are then standardized to zero mean and unit variance using scikit-learn's standardization routine with mean-centering enabled. In the case of fully connected networks, this choice benefits gradient-based optimization by "stabilising the scale of activations and gradients across layers."
The XGBoost and MLP models can assist with the ranking problem in complementary and powerful ways. The XGBoost specializes in tree-based partitioning of the feature space, with strong inductive biases for tabular data. Meanwhile, the MLP specializes in learning dense, distributed representations that can capture other nonlinear structures in the same feature space.

6.2.3 Training Procedure

Both regression models follow a standardized training pipeline to ensure fair comparison and reproducibility. The procedure consists of feature preprocessing, data splitting, model training, and final evaluation on a held-out test set.

Feature preprocessing

The models operate on the full set of engineered numerical features described in Chapter 5. Before training, missing values in these features are handled using median imputation, where each missing entry is replaced by the median of the corresponding feature computed on the training data. This simple but robust strategy reduces the influence of extreme outliers on the imputed values. After imputation, the features are standardized, but with slightly different configurations for the two model families.
For the XGBoost regressor, features are scaled in terms of variance without mean-centering. This preserves sparsity in the transformed design matrix and is well

suited to the column-transformer pipeline used in the implementation. For the neural network, features are standardized to zero mean and unit variance, which is beneficial for gradient-based optimization in fully connected networks, as it stabilizes the scale of activations and gradients across layers. In both cases, the preprocessing parameters (medians and scaling statistics) are fitted exclusively on the training split and then applied to validation and test data, thereby preventing information leakage from the test set into the training process.

Data splitting

To evaluate generalisation performance, the dataset is split into three disjoint subsets using a fixed random seed. First, 20% of the data is held out as a test set, and the remaining 80% is reserved for model development (training + validation). The development portion is then split again into 75% training and 25% validation, resulting in an overall partition of 60% training, 20% validation, and 20% test data. This deterministic two-stage split ensures that both the XGBoost model and the neural network are trained and evaluated on exactly the same examples, enabling a fair, like-for-like comparison. No stratification is applied: the composite ranking target is treated as a continuous variable, and the random split preserves its empirical distribution up to sampling variability.

Model training and validation

The XGBoost regressor is trained on the training split after preprocessing. Since gradient-boosted trees incorporate several forms of built-in regularisation— such as limited tree depth, shrinkage via a small learning rate, and stochastic subsampling of instances and features—the model is fitted once on the training data and monitored on the validation set. In the implementation, the `XGBRegressor` is given the validation data as an `eval_set` so that its optimisation statistics can be inspected, but no explicit early-stopping criterion is applied. After training, the fitted model is evaluated on both the validation set (for model selection and diagnostics) and the held-out test set (for final reporting).
The neural network follows a slightly different training protocol that includes internal validation and early stopping. After preprocessing, the same validation split described above (comprising 20% of the full data) is used as *validation data* during training. The multi-layer perceptron (MLP) is then optimised on the remaining 60% of the data for up to 200 epochs using mini-batches of size 256, with the Adam optimiser and mean squared error (MSE) loss. An early-stopping callback monitors the validation root mean squared error (RMSE) and halts training if this metric does not improve for 12 consecutive epochs, restoring the model parameters corresponding to the best observed validation performance. A `ReduceLROnPlateau` callback further decreases the learning rate when validation RMSE stagnates, helping the optimiser to refine the solution.

6.2.4 Model Output Format

The project ranking models produce a continuous scalar impact score for each repository, representing its predicted composite impact on the 0–100 scale defined in Section 6.2.2. On this scale, a higher score directly corresponds to a higher-ranked and

more impactful project: repositories with more points are placed above those with fewer points.
In the portfolio generation pipeline, all repositories associated with a given developer are therefore sorted by their predicted impact score in descending order. The top three highest-scoring repositories are selected as that developer's "top 3 projects" and highlighted in the generated portfolio, providing a concise summary of the most impactful work in their GitHub profile.

6.3 Task 2: Technical Skills

6.3.1 Technical Skills Labels

Technical skills classification is formulated as a multi-label problem, reflecting the fact that individual developers typically work with several programming languages and technology ecosystems in parallel [13]. Instead of forcing each developer into a single "primary" language, the system predicts a **set of binary skill indicators** per developer, yielding a more realistic, multi-stack profile.
In the current implementation, the label space is derived directly from the language information contained in the mined GitHub repositories. For each repository, the data collection pipeline records (i) the repository owner, (ii) the primary programming language associated with that repository, (iii) the total size of code written in that language (in bytes), and (iv) the number of distinct languages observed in that repository. Each distinct primary language observed across the corpus is treated as a candidate skill dimension. This yields a set of L language-centric skills (on the order of a few dozen), such as Python, JavaScript, TypeScript, Go, C/C++, Rust, and so on.
For each developer language pair, the system first constructs a **continuous proficiency score** that combines both code volume and language dominance. At the repository level, a language-dominance factor is assigned based on how many languages are used in that repository: single-language repositories receive a high dominance weight, while projects with many languages receive a lower weight. The effective lines of code for a repository in its primary language are then approximated by multiplying the byte size of that language by the dominance weight and dividing by a constant bytes-per-line factor. These per-repository quantities are aggregated per developer and language to obtain (a) the total estimated lines of code written in that language and (b) the average dominance of that language across the developer's repositories. The continuous proficiency score for a developer in a given language is defined as a weighted combination of these two components: a log-transformed code volume term and an average dominance term. Concretely, the log of the total lines of code (plus one) contributes 60% of the score, while the mean dominance contributes the remaining 40%. This produces a dense, non-negative matrix of developer–language proficiencies, in which higher values indicate stronger and more sustained engagement with that language.
Because raw proficiency values can vary across languages, the matrix is **globally normalized** so that scores are comparable across skill dimensions. The implementation computes the minimum and maximum non-zero proficiency scores across all developers and languages, and applies a min–max scaling to map all positive scores into the [0,1] interval while leaving zero scores unchanged. The resulting normalized

matrix preserves relative ordering within each language while placing all skills on a common scale. To obtain **binary skill labels** suitable for supervised multi-label classification, the normalized proficiencies are converted into 0/1 indicators using a data-driven thresholding scheme. For each language, the method considers only those developers with non-zero proficiency and chooses a threshold such that (i) at least a minimum number of positive examples (e.g. 50 developers) are retained and (ii) the resulting prevalence for that language is approximately 10% among eligible developers. Concretely, the normalized scores are sorted, and the threshold is taken as the score of the developer at the target rank implied by these criteria. Developers with scores above or equal to this threshold are assigned a label of 1 for that language, and all others receive 0. Languages with insufficient support are flagged but retained, with their low prevalence explicitly documented in a table of label statistics (positives, prevalence, and threshold per skill).
The final skills label matrix therefore consists of one binary column per language-level skill, plus an identifier column aligning labels with the engineered feature matrix. Each row corresponds to a single developer, and each skill column indicates whether the developer demonstrates a sufficiently strong and consistent proficiency in that language according to the above volume- and dominance-based criteria. This matrix is treated as fixed multi-label ground truth throughout the subsequent modelling and evaluation stages.

6.3.2 Model Selection and One-vs-Rest Strategy

The technical skills prediction task is inherently multi-label: a single developer can, and often does, exhibit proficiency in multiple languages simultaneously. With L language-level skills, a naïve label powerset approach would implicitly consider up to 2^L distinct label combinations, leading to an extremely large and sparse output space and many combinations with little or no empirical support [5]. This motivates a decomposition strategy that scales more gracefully.
The system therefore adopts a One-vs-Rest (OvR) formulation. In this setup, the multi-label problem is decomposed into L independent binary classification tasks, one per skill dimension. For each language skill, a separate classifier is trained to distinguish developers who exhibit that skill (positive class) from those who do not (negative class). At prediction time, all L classifiers operate on the same feature representation, and their outputs are combined into a multi-label prediction vector for each developer. Although OvR does not explicitly model label dependencies, correlations between skills are still captured implicitly through shared input features (for example, languages that often co-occur in the same repositories or development roles).

6.3.3 Skill Proficiency Label Construction

Because GitHub does not provide explicit skill labels for developers, this work constructs a language-level proficiency signal from repository-level language statistics, and then converts that signal into binary multi-label targets.
The construction follows three intuitive steps:

1. Repository-level dominance and lines of code

For each repository (r), the mined data provide:

- the repository owner ($u(r)$),
- the primary programming language ($f(r)$),
- the total number of bytes of code across all languages in that repository, denoted (size.),
- the number of distinct languages used in the repository, denoted (count.).

A language-dominance weight (w) is assigned based on (count.): repositories using a single language receive a high dominance weight (close to 1), whereas repositories with many languages receive a lower weight. In code, this is implemented as a simple look-up table (e.g. 0.95 for one language, 0.70 for two, gradually decreasing to 0.35 when many languages are present).
Using this weight, an effective byte volume for the primary language is computed as

$$\text{bytes}_\ell^{\text{eff}} = \text{size}_\ell \times w_\ell$$

Assuming roughly 50 bytes per line of code, the estimated number of lines of code (LOC) in the primary language of repository (r) is

$$LOC_r = \frac{\text{bytes}_\ell^{\text{eff}}}{50}$$

2. Developer-language proficiency scores

For each developer (u) and language (ℓ), we consider the set of repositories ($\mathcal{R}_u$) owned by (u) whose primary language is (ℓ). From these repositories we derive two quantities:

- Total estimated lines of code in language (ℓ):

$$T_{u,\ell} = \sum_{r \in \mathcal{R}_{u,\ell}} LOC_r$$

- Average dominance of language (ℓ) across those repositories:

$$D_{u,\ell} = \text{average of } w_r \text{ for } r \in \mathcal{R}_{u,\ell}$$

These are combined into a single continuous proficiency score ($p_{u,\ell}$) using a weighted sum of a log-scaled volume term and an average dominance term:

$$p_{u,\ell} = 0.60 \times \log\left(1 + T_{u,\ell}\right) + 0.40 \times D_{u,\ell}$$

In words, the score increases when a developer has written more code in a language (total lines of code), but also when that language tends to be the main focus of their repositories (high dominance). The logarithm dampens the effect of very large projects while preserving the ordering by code volume.

3. Global normalization and binary labels

Collecting $(p_{u,\ell})$ for all developers and languages yields a developer-language proficiency matrix. To make scores comparable across languages, all non-zero proficiency values are first mapped into the interval $[0, 1]$ using a simple global min-max transformation:

$$\tilde{p}_{u,\ell} = \frac{p_{u,\ell} - p_{\min}}{p_{\max} - p_{\min}} \quad \text{for } p_{u,\ell} > 0$$

where $(p_{\min})$ and $(p_{\max})$ are the minimum and maximum non-zero scores observed across all developers and languages. Scores equal to zero remain at zero. This produces a normalized proficiency matrix $(P = [\tilde{p}_{u,\ell}])$ with values between 0 and 1. Finally, each language (ℓ) is converted into a binary skill label by applying an adaptive threshold:

- consider only developers with $(\tilde{p}_{u,\ell} > 0)$ (those who have at least some activity in the language);

- sort their normalized scores from highest to lowest; - choose a cut-off such that at least a minimum number of positive examples (e.g. 50 developers) are retained and, where possible, roughly 10% of active developers for that language are labelled as positive; - assign $y_u = 1$ if $\beta_u \geq$ cut-off, and 0 otherwise
This yields a binary matrix $(Y = [y_u])$ where each column corresponds to a language-level skill, and each row corresponds to a developer. Languages with very few active developers are marked as having low support but are still included, with their thresholds and prevalence documented. The resulting matrix (Y) is treated as the fixed multi-label ground truth for training and evaluating the skills classifiers.

6.3.4 Feature Representation

The input representation X for the skills classifier is the engineered developer-level feature matrix constructed in the feature-engineering pipeline described in Chapter 4. Each row corresponds to a single GitHub user, and each column encodes an aggregate property of that user's activity or presence on the platform (for example, counts of repositories, stars, forks, contributions, followers, and other derived behavioural indicators). The exact feature inventory and construction rules are documented in Chapter 4; here it is sufficient to note that all variables used in the skills model are numerical, developer-level aggregates.
Before training, the feature matrix is aligned with the skills label matrix via a common developer identifier (`owner_login`). An inner join on this identifier yields a consolidated table in which each row contains both the developer's feature vector and the corresponding multi-label skill indicators. The feature block X is obtained by selecting all non-identifier, non-label columns, while the label block Y consists of the binary skill columns.
Preprocessing is implemented as a scikit-learn pipeline. For all numerical features, missing values are imputed using the median of each feature computed on the training split, and the resulting columns are standardized to have zero mean and unit variance. In the current configuration, no categorical features are used in the skills experiments, so the preprocessing reduces to this numeric imputation-and-scaling

step. The preprocessing pipeline is fitted once on the training data and then applied unchanged to validation and test sets, ensuring that no information from the held-out data affects the learned transformation.

6.3.5 Model selection and architecture

Model 1: Logistic Regression One-vs-Rest

The first skills model is a logistic regression classifier in a One-vs-Rest configuration. The pipeline consists of the shared preprocessing stage followed by a One-vs-Rest meta-estimator that wraps a regularised logistic regression base learner. For each skill label, the base learner fits a linear decision boundary with L2-regularised logistic loss, optimised using the `liblinear` solver, which is well suited to binary problems with potentially high-dimensional feature spaces [3].
To mitigate label imbalance where common languages may have orders of magnitude more positive examples than rare ones the logistic regression is configured with balanced class weights [4]. The contribution of each class to the loss is inversely proportional to its frequency, discouraging the model from collapsing into trivial "always negative" predictions for rare skills. The maximum number of optimisation iterations is set to 2,000 to ensure convergence across all skill-specific problems, and parallelisation is enabled so that multiple OvR classifiers can be trained concurrently. The output of this model is a matrix of predicted probabilities per developer and skill, which can be thresholded to obtain binary predictions.

Model 2: XGBoost One-vs-Rest

The second skills model replaces the linear base learner with a gradient-boosted decision tree ensemble implemented via XGBoost. Each skill-specific classifier is an XGBoost model configured for probabilistic binary classification with a logistic loss. The ensemble consists of 500 shallow trees with maximum depth six and a conservative learning rate, trained with a fixed random seed and parallel execution across CPU cores. Compared to the linear logistic baseline, the XGBoost OvR model can capture non-linear interactions between features and skill presence—for example, combinations of high language diversity with strong collaboration scores that are especially indicative of certain language skills. The internal evaluation metric is log-loss, so the training process directly optimises the quality of probabilistic predictions.
Both pipelines—logistic regression OvR and XGBoost OvR—are trained on the same training split and evaluated on the held-out test split using a comprehensive suite of multi-label metrics. For each model, the evaluation function computes accuracy (exact-match ratio), micro- and macro-averaged F1 scores, micro and macro precision and recall, Hamming loss, Jaccard similarity (micro and macro), and ranking-based metrics such as label-ranking average precision and coverage error. The two models are compared along these dimensions, and the model with the higher micro-averaged F1 score is designated as the primary skills classifier for subsequent threshold optimisation and portfolio generation, while both trained pipelines are retained for analysis and ablation.

6.3.6 Model output format

The skills classifier operates on a merged developer-level dataset where each row corresponds to a single GitHub user and each column in the label matrix corresponds to a language-level technical skill. For a dataset with N developers and L skills, the model produces:

1. A matrix of probability scores $\hat{P} \in [0,1]^{N \times L}$, where $\hat{P}_{i\ell}$ denotes the model's estimated probability that developer i possesses skill ℓ.
2. A corresponding matrix of binary predictions $\hat{Y} \in \{0,1\}^{N \times L}$, obtained by thresholding each probability according to the per-skill decision thresholds described in Section 6.3.7.

Each row of $\hat{Y}$ can be viewed as a multi-label skill vector for a developer. A value of 1 in position ℓ indicates that the developer is predicted to have meaningful proficiency in the corresponding language or technology, while 0 indicates insufficient evidence. The length of this vector equals the number of skill columns (i.e., the number of distinct languages for which the label construction step produced valid binary labels). For downstream use in the portfolio generation pipeline, the system also derives several summary views of the skill predictions:

- The predicted skill count per developer (number of active labels in $\hat{Y}_{i,\cdot}$), which quantifies how broad a developer's predicted skill set is.
- A ranked list of top-k skills per developer (typically $k = 5$), obtained by sorting the per-developer probability vector $\hat{P}_{i,\cdot}$ in descending order and selecting the highest-scoring skills along with their probabilities.
- A tabular representation of these top-k skills (developer identifier, skill name, probability) that can be directly consumed by the portfolio rendering components.

In practical terms, a prediction such as

$$\hat{Y}_{i,\cdot} = [1, 0, 1, 0, \ldots, 1]$$

indicates that developer i is predicted to possess the skills corresponding to the positions where the vector takes the value 1 (for example, Python, JavaScript, and Go), while lacking sufficient evidence for the remaining languages. The probability matrix $\hat{P}$ is preserved so that skilful trade-offs between precision and recall can be made via threshold adjustment without retraining the underlying models.

6.3.7 Threshold optimization

The One-vs-Rest classifiers naturally output probability scores for each developer–skill pair rather than hard labels. A naïve approach would apply a uniform threshold of 0.5 to all skills; however, this is rarely optimal in the presence of label imbalance and heterogeneous calibration properties across skills. Some languages are common and well supported, while others are rare; their score distributions differ substantially.

To address this, the system applies a per-skill threshold optimisation procedure to the probability outputs of the best-performing skills model (as selected in Section 6.3.2). Let
$\hat{P}_{\cdot,\ell}$ denote the vector of predicted probabilities for skill ℓ on the test split, and let $Y_{\cdot,\ell}$ be the corresponding ground-truth binary labels.
For each skill label ℓ:

1. The procedure first computes the F1 score obtained by thresholding $\hat{P}_{\cdot,\ell}$ at the default value 0.5, yielding a baseline F1 for that skill.

2. It then performs a one-dimensional grid search over a set of candidate thresholds in the interval $[0.2, 0.8]$, evaluating the F1 score at each candidate threshold by converting probabilities into binary predictions and comparing them with $Y_{\cdot,\ell}$.

3. The threshold that maximises the per-skill F1 score is selected as the optimal decision boundary for that skill. If a skill is degenerate on the test split (e.g., only one class is present), threshold tuning is skipped and the default value of 0.5 is retained.

The outcome is a dictionary of per-skill thresholds and a matrix of tuned predictions, where each column of $\hat{Y}$ has been computed using the skill-specific optimal threshold rather than a global 0.5 rule. The optimisation routine also records, for each skill, the baseline F1 at 0.5, the tuned F1, the absolute improvement, and the support (number of positive examples), and exports these statistics to CSV files for documentation and further analysis.
To assess the global effect of threshold optimisation, the system compares the multi-label evaluation metrics computed under the default 0.5 thresholding and under the tuned thresholds. For the tuned predictions, micro- and macro-averaged F1 scores and Hamming loss are recomputed and reported alongside the corresponding default values, highlighting how much the tailored thresholds improve performance. Typically, macro-averaged F1 benefits most from tuning, because rare skills receive thresholds that are better matched to their score distributions, whereas common skills often retain thresholds close to 0.5.
Finally, the distribution of optimal thresholds across skills is summarised (mean, median, minimum, maximum, and the proportion of skills choosing thresholds below or above 0.5), and histograms of the thresholds are plotted. These analyses confirm that many skills benefit from thresholds different from 0.5, and that the tuned decision rules strike a better balance between precision and recall across the diverse set of language-level skills.

6.3.8 Training Procedure

Data Preparation and Splitting

The training pipeline for skills classification begins by merging the engineered feature matrix with the binary skills label matrix described in Section 5.3.1. The feature table contains one row per developer and a set of behavioural, technical, and influence-related features; the label table contains the same developer identifier and one binary column per language skill. Before merging, identifiers are harmonised

to ensure that the same key is used in both tables. An inner join on this identifier yields a consolidated dataset in which each row corresponds to a single developer with both feature and label information. Summary statistics (feature count, skill count, and merge match rate) are reported to verify that the majority of feature rows successfully align with labels.
From this merged dataset, the feature matrix ($\boldsymbol{X}$) is extracted by selecting all non-identifier, non-label columns, and the label matrix ($\boldsymbol{Y}$) is formed by selecting the skill columns. The data are split into training and test sets with an 80/20 ratio using a fixed random seed. After splitting, the pipeline reports the shapes of (X_{ral}), (X_e), (Y_{ral}), and (Y_e), together with the average number of active labels per developer in each split. This provides an initial view of label cardinality (how many skills developers typically have) and ensures that the test set contains a representative sample of the overall label structure.

Label Distribution Analysis

Because multi-label performance is highly sensitive to label imbalance, the training procedure includes an explicit label distribution analysis. For each skill, the number of positive examples in the training and test splits is computed and sorted to obtain a ranked support profile. The most common and least common skills, their associated sample counts, and the resulting imbalance ratio are reported for both splits, together with the median support across skills. Visualisations are generated for the top 30 most frequent skills (vertical and horizontal bar charts), and a co-occurrence heatmap is produced for a subset of common skills to illustrate which languages tend to appear together at the developer level. These diagnostics both motivate the use of class weighting and provide context for interpreting macro-averaged metrics, which give equal weight to common and rare skills.

Model Training

With the merged data prepared and the preprocessing pipeline defined, both skills models are trained as follows:

1. The logistic regression OvR pipeline is fitted on the training data: the preprocessing transformer is first fitted on (X_{rai}) (imputing missing values and scaling numerical features), and then the One-vs-Rest wrapper trains one logistic regression classifier per skill using the corresponding column of (Y_{rai}) as the target. Class weights are automatically computed based on label frequencies in the training split via the balanced option, and the training process runs until convergence or until the iteration cap is reached.

2. The XGBoost OvR pipeline is trained analogously: the same preprocessing is applied to (X_{rai}), after which the One-vs-Rest meta-estimator fits one XGBoost classifier per skill on the transformed features and corresponding labels. Because the preprocessing step is shared, both models see identically transformed inputs, making their performance directly comparable.

3. After training, each model is evaluated on the test split via the common evaluation function, producing a dictionary of multi-label metrics. These results

are assembled into a comparison table, and the model with the higher micro-averaged F1 score is selected as the preferred skills classifier. The per-label performance of the chosen model is then analysed: for each skill, the support and F1 score are computed, and a summary is produced highlighting the top-performing and most challenging skills. A dedicated visualisation shows F1 scores for the top 30 skills by performance, annotated with their support, to provide an interpretable overview of where the model performs reliably and where predictions remain noisy.

Finally, the probability outputs of the best model are passed to the threshold optimisation step in 6.3.7, and the resulting per-skill thresholds are used to derive the tuned binary predictions. Both the models and their associated preprocessing pipelines are stored for later use in portfolio generation and for potential ablation or robustness studies, ensuring that the reported results are fully reproducible from the underlying code.

6.4 Task 3: Behavioral Pattern

6.4.1 Behavioural Pattern Labels

Behavioural classification aims to characterize developers along a small set of interpretable archetypes that capture their predominant working styles and contribution patterns. In this work, four such behavioural dimensions are considered: Maintainer, Team Player, Innovator, and Learner. Each dimension is first represented by a continuous behaviour score derived from GitHub activity signals (Section 5.4), and is subsequently converted into a binary label via a percentile-based thresholding scheme.

Maintainer

Maintainers are developers who initiate and sustain software projects over the long term, serving as the backbone of open-source communities. Unlike casual contributors who submit drive-by patches, maintainers assume ownership roles by establishing project goals, guiding strategic direction, and continuously managing codebase evolution [6]. Beyond writing code, maintainers perform critical stewardship tasks including community cultivation, contributor mentorship, and contribution vetting to ensure alignment with project vision [29].
Maintainers transparently communicate project goals, devote time to onboarding new members, and ensure long-term sustainability responsibilities that corporate developers in managed environments rarely undertake [6]. Empirical studies confirm this pattern in successful projects like Linux and Apache, where core developers functioning as maintainers oversee the majority of decisions and updates, anchoring project continuity and quality.
Operationally, maintainers serve as core contributors or committers who perform code reviews, release management, bug triage, and community management [42]. The 2020 FOSS Contributor Survey reveals that maintainers allocate substantial time to routine maintenance tasks documentation, bug reports, and administration often exceeding their preferred allocation [42]. This distribution aligns with the

"onion model" of OSS communities, wherein a small core of maintainers performs most coordination and review work while peripheral contributors engage sporadically.
From a motivational perspective, maintainers align with stewardship and intrinsic motivation models. Surveys indicate that maintainers are driven primarily by non-monetary rewards including problem interest, community reputation, and learning satisfaction, rather than financial incentives [45] [42]. This corresponds with Self-Determination Theory's emphasis on intrinsic motivation and goal-orientation theory's demonstration that learning goals sustain persistent effort in challenging tasks [29] [45]. The Maintainer archetype emphasizes stability, consistency, and community stewardship, representing organizational behavior concepts of stewardship and exploitation the refinement of existing assets [6] [42].

Team Player

Team Players are collaboration-oriented developers who prioritize collective goals over individual achievement and actively engage with others' work. They share information freely, coordinate tasks, and solicit teammates' ideas, functioning as supportive integrators who resolve conflicts, encourage cohesion, and trust teammates with responsibility [37]. In agile teams, team orientation is defined as the propensity to consider others' behavior during group interaction and to value team goals over individual objectives [37]. Team Players demonstrate this orientation by evaluating alternative solutions proposed by colleagues and openly appraising peer input to determine optimal approaches [37].
Empirical research on GitHub and OSS ecosystems reveals wide variation in developer collaboration patterns, with Team Players exhibiting high collaboration intensity through participation in group projects, code reviews, and organizational coordination [22]. These developers facilitate knowledge sharing, mutual support, and coordinated problem-solving, serving as connectors who bind communities together across multiple repositories [22] [23].
From a psychological perspective, Team Players align with the Big Five trait of agreeableness characterized by cooperation, consideration, and supportiveness which correlates with constructive team contribution [26]. Team-oriented behavior facilitates mechanisms such as mutual performance monitoring, peer feedback, and shared mental models in agile teams [37]. Studies demonstrate that teamwork quality, encompassing communication, coordination, and collaboration, explains up to 81% of variance in team success metrics [23]. Conversely, teams lacking such integrative contributors suffer from "lone wolf" dynamics that undermine collective effectiveness [37].
The Team Player archetype thus represents a distinct behavioral pattern validated by field evidence and organizational theory, where developers prioritize collective success, readily assist others, integrate feedback, and align with team goals [22], [23], [37].

Innovator

The Innovator archetype denotes developers driven by novelty, creativity, and impact, who create new projects or features that gain significant community traction.

Rather than focusing on incremental maintenance, Innovators pursue exploration and change by proactively introducing original ideas, tools, or frameworks, often serving as early adopters of emerging technologies. Scholarly literature explicitly links open-source development with innovation dynamics, characterizing it as user-driven innovation where individuals "advance innovation by solving problems valuable to the community" [39]. Innovators launch new repositories, implement bold features, and experiment with cutting-edge technologies; successful projects accumulate users, contributors, and stars as indicators of popularity.

This pattern is empirically observable on platforms like GitHub, where a subset of developers disproportionately accounts for launching influential projects such as popular libraries and frameworks [39]. From an organizational psychology perspective, this archetype aligns with innovative work behavior, defined as "the intentional generation, promotion, and realization of new ideas within a role or organization" [20]. Such individuals exhibit creativity, proactiveness, and risk-taking by suggesting new products or processes and championing their implementation [20]. In team contexts, they fulfill visionary roles, seeding groups with novel concepts.

Psychologically, Innovators align with creativity and proactivity theories. Individuals with proactive personality traits initiate change and engage in innovative acts; empirical studies demonstrate that proactive personality strongly predicts innovative behavior [25]. Software teams depend on members who adopt growth mindsets toward change. Open-source analyses reveal that Innovators propose new modules and undertake major refactorings rather than merely patching bugs. While the Innovator archetype carries inherent risk not all new ideas succeed it represents a recognized pattern where individual innovativeness drives project evolution [25]. The archetype thus captures a distinct behavioral pattern of developers who drive technical innovation and project creation, leaving broad impact through the introduction of new ideas and technologies [20], [39].

Learner

The Learner archetype represents growth-oriented developers who continuously expand their skills and knowledge base by venturing into new programming languages, frameworks, or domains while maintaining current activity levels. Empirical evidence from open-source communities demonstrates the prevalence of this pattern: in a survey of 2,700 open-source contributors, the primary motivation for joining and sustaining participation in OSS projects was "to learn and develop new skills" [8]. This learning motivation constitutes a statistically confirmed driver of community engagement rather than merely anecdotal evidence.

Developers with high Learner tendencies demonstrate polyglot characteristics by contributing across diverse projects and programming languages, exhibiting recent and sustained activity as they acquire new tools [8]. Organizational psychology provides theoretical grounding for this archetype through learning goal orientation and intrinsic motivation for mastery. Humans possess "a desire toward mastery, spontaneous interest, and exploration" in their work [8], which in software development—a rapidly evolving field—manifests as continual technical skill acquisition. Researchers observe that learning in technology represents an inherent, ongoing process; given software engineering's complexity and constant evolution, the drive to learn confers both personal and project benefits [8].

The Learner archetype mirrors psychological concepts of mastery orientation and growth mindset, wherein individuals seek challenges to develop competence. Practically, these developers experiment with new programming languages in side projects, pursue online courses, and actively engage with documentation to advance their capabilities. This behavioral pattern is particularly evident in open-source environments, which many developers utilize as learning laboratories through hands-on practice. The Learner archetype is thus justified by both community observations and theoretical frameworks: a significant fraction of developers are primarily motivated by personal improvement and exploration, making continuous learning a defining feature of their participation [8].
A cohort-based amplification procedure transforms the four continuous scores into discrete behavioral labels used for multi-label classification (whereby these scores are discretized into binary targets). To compute the score distribution across all developers for each behavior, a threshold is established at the 70th percentile. Developers whose score on a behavior is at or above this percentile are labeled positive for that behaviour, and the rest are labeled negative. In other words, about the top 30 percent of developers are indicated to show that pattern strongly along each behavior dimension. This method based on percentile avoids hard thresholds rooted on woven counts. In addition, it ensures that each of the behavioral labels would be having a sufficient number of positive examples along with comparable numbers for model training.
The resulting behavioral labels are non-exclusive: a single developer may simultaneously be classified, for example, as both a Maintainer and an Innovator, or as a Team Player and a Learner. This multi-label formulation is essential for capturing the multifaceted nature of real developer personas, where sustained maintenance, collaborative work, innovation, and continuous learning often coexist rather than forming mutually exclusive categories.

6.4.2 Model Selection and One-vs-Rest Architecture

Behavioural pattern classification adopts the same One-vs-Rest (OvR) decomposition strategy as the technical skills models, but with a different choice of base classifiers. The task is formulated as four independent binary problems, one per behavioural archetype (Maintainer, Team Player, Innovator, Learner). For each behaviour, the classifier predicts whether the corresponding label is active or not. All four classifiers operate on the common feature space constructed in Chapter 4, but the continuous behavioural scores themselves (e.g., `maintainer_score`, `team_player_score`) are removed from the input feature set to avoid trivial circular predictions based on the proxy features used to define the labels.
Feature Set (39 features): The behavioral classifier operates on a comprehensive feature set derived from GitHub activity. These include:

- **Direct Metrics:**

  `total_repos, total_stars, total_forks, total_commits, total_prs, total_issues, total_pr_reviews, followers, following, language_diversity, account_age_days`

- **Activity Rates:**

  `commits_per_day, prs_per_day, issues_per_day, repo_creation_rate, active_repos`

- **Collaboration Intensity:**

  `collaboration_score`, `pr_review_ratio`, `code_review_participation`, `code_review_index`, `mentorship_score`

- **Repository Quality:**

  `avg_stars_per_repo`, `avg_forks_per_repo`, `max_stars_repo`, `max_forks_repo`, `repo_active_score`

- **Social Influence:**

  `follower_ratio`, `network_influence`, `influence_growth_rate`, `social_coding_index`, `community_engagement_score`, `reputation_score`

- **Advanced Behavioral:**

  `leadership_score`, `development_velocity`, `contribution_consistency`, `work_consistency`, `generalist_score`, `impact_factor`, `viral_repo_score`

Two model families are evaluated for this task. The first is a logistic regression baseline that provides a linear decision boundary in the engineered feature space. The second is a Support Vector Machine (SVM) with a radial basis function (RBF) kernel, chosen to capture potential non-linear relationships between the features and behavioural labels [1]. Both models are wrapped in a scikit-learn `OneVsRestClassifier`, which instantiates one binary sub-model for each behaviour while sharing the same preprocessing strategy within each model family.

Model 1: Logistic Regression One-vs-Rest

For the logistic regression model, the pipeline consists of a preprocessing stage followed by a One-vs-Rest logistic regression classifier. Preprocessing applies median imputation to all numerical features, replacing missing values with the median computed on the training split, and then scales each feature by its standard deviation without mean-centering. This variance-only standardization (using a scaler configured not to subtract the mean) brings features onto a comparable scale and is compatible with sparse internal representations. The base logistic regression classifier uses L2L regularization and is optimized with the `liblinear` solver, which is well suited to binary classification problems in moderate- to high-dimensional spaces. The maximum number of iterations is set to 2,000 to ensure convergence even for more difficult behaviours, and class weights are set to "balanced", so that the loss contribution from positive and negative examples is inversely proportional to their frequency. A fixed random seed is used, and parallelization across CPU cores is enabled via the OvR wrapper, allowing the four behaviour-specific classifiers to be trained concurrently. This model serves as an interpretable linear baseline with explicit handling of class imbalance.

Model 2: Support Vector Machine with RBF Kernel (One-vs-Rest)

The SVM model uses the same input feature set but a different preprocessing configuration and learning mechanism. Median imputation is again applied to the numerical features, but the subsequent scaling step both centers each feature to

zero mean and rescales it to unit variance. Mean-centering is important for kernel-based methods such as SVMs, as it tends to improve numerical stability and the effectiveness of the RBF kernel.
The base classifier is a Support Vector Machine with a radial basis function (RBF) kernel. The RBF kernel is defined as

$$k(x, x') = \exp\left(-\gamma \|x - x'\|^2\right),$$

with γ set to the "scale" heuristic in `scikit-learn`, which adapts the kernel width to the feature dimensionality and variance. This kernel choice allows the model to capture non-linear relationships between the engineered features and the behavioural labels.
The regularization parameter C is set to 1.0, trading off margin maximization against tolerance of training errors in a moderate way, and class weights are again set to `balanced` to compensate for the approximate 30–70 split between positive and negative examples induced by the percentile-based label construction. Probability estimates are enabled, which triggers internal Platt scaling to convert SVM decision scores into calibrated probabilities; these probabilities are later used for threshold optimization and ranking-based evaluation metrics. As with logistic regression, a fixed random seed and parallel training across the four OvR sub-models are employed.
In summary, behavioural classification relies on a linear One-vs-Rest logistic regression model as a strong, well-regularized baseline, and on an SVM with an RBF kernel as a more expressive non-linear alternative. Both models share the same engineered features and label definitions but differ in their inductive biases and preprocessing requirements, enabling a direct comparison between linear and non-linear decision surfaces for modelling developer behavioural patterns.

6.4.3 Model Output Format

The behavioral classifier produces a 4-element binary vector for each developer, where each position corresponds to one behavioral archetype:

- Position 0: Maintainer - Sustains and owns original projects
- Position 1: Team Player - Collaborates extensively through reviews and PRs
- Position 2: Innovator - Creates influential, community-valued repositories
- Position 3: Learner - Actively explores multiple languages and technologies

Each element can be 1 (behavior present) or 0 (behavior absent). Developers may exhibit multiple behaviors simultaneously. For example:

- $[1, 0, 1, 0]$ = Maintainer + Innovator
- $[1, 1, 0, 1]$ = Maintainer + Team Player + Learner
- $[0, 1, 1, 1]$ = Team Player + Innovator + Learner

6.4.4 Data Preparation and Training Procedure

Data Splitting and Feature Selection

The behavioural classification pipeline begins by constructing feature and label matrices from the engineered dataset. The label matrix contains four binary columns corresponding to the behavioural archetypes defined in Section 5.4 (Maintainer, Team Player, Innovator, Learner). These labels are obtained by applying a percentile-based threshold to the underlying continuous proxy scores (`maintainer_score`, `team_player_score`, `innovation_index`, and `learning_velocity`). A behaviour is marked as present when the corresponding score lies at or above the 70th percentile (i.e., within the top 30% of the developer population).

To avoid trivial label leakage, the proxy score features used to construct these labels are excluded from the input features. Concretely, the feature matrix is obtained by dropping the developer identifier, the four behavioural score columns, and the four binary label columns from the combined dataset. The remaining features are purely numerical and span activity, collaboration, skills, quality, and influence dimensions introduced in Chapter 5.

The resulting dataset is split into a training set (60%), a validation set (20%), and a test set (20%) using a random partition with a fixed seed (`42`) to ensure reproducibility. After the split, basic label statistics are computed on the training set: for each behavioural pattern, the number and proportion of positive examples are reported, and the average number of behaviours per developer is calculated. Because each label is constructed using a 70th-percentile cut-off on its proxy score, each behaviour is active for roughly one third of developers, although the exact prevalence can vary slightly due to ties and differences in score distributions. These statistics summarize both class imbalance and label co-occurrence in the multi-label setting.

Preprocessing Pipelines

Two preprocessing configurations are used, reflecting the different numerical requirements of the logistic regression and SVM classifiers.

For the logistic regression model, preprocessing is implemented within a single scikit-learn pipeline. All input features first undergo median imputation, replacing missing values with the median computed on the training set. The imputed features are then rescaled using variance-only standardization, where each feature is divided by its training-set standard deviation but not mean-centered. This brings features to a comparable scale while preserving any sparsity patterns in the design matrix. The imputation and scaling parameters are learned on the training data and applied unchanged to both training and test sets.

For SVM model, preprocessing is done at a later step of transformation. In a similar fashion to before, median imputation is applied, but the subsequent scaling both centers to zero mean each feature and scales it to unit variance. This complete standardization is good for SVMs with RBF kernels, which rely on a distance-based computation that is sensitive to feature vectors' relative positions and scales. The transformer is fitted on the training features, after which standardized training and test arrays are obtained that are passed directly to the SVM classifier.

The design provides each model family with a feature representation tailored to its

numerical assumptions and optimization behavior while both feature representations would be based on the underlying engineered features.

Training Procedure: Logistic Regression One-vs-Rest

The first behavioural model uses logistic regression in a One-vs-Rest configuration. The complete pipeline, consisting of preprocessing followed by the One-vs-Rest classifier, is trained on the raw training feature matrix and the four-column label matrix. Internally, the pipeline fits the imputation and scaling steps on the training data, transforms the features, and then trains four independent logistic regression models, one for each behavioural label.

Each logistic regression sub-model optimizes an L2L-regularised logistic loss using the `liblinear` solver. The maximum number of iterations is set to a high value (2 000) to ensure convergence even for challenging decision boundaries. Class imbalance is handled by using class weights that are inversely proportional to class frequencies within each binary problem, so that errors on the minority (positive) class contribute more to the loss than errors on the majority class. This prevents the model from converging to a trivial majority-class predictor and encourages it to learn meaningful boundaries for all behaviours. Training is parallelised across behaviours, which reduces wall-clock time on multi-core hardware.

Once training has completed, the logistic regression pipeline can produce, for each developer, both binary predictions and probability estimates for each of the four behavioural dimensions, using the same preprocessing steps that were fitted during training.

Training Procedure: SVM with RBF Kernel

The second model family is based on Support Vector Machines with a radial basis function (RBF) kernel, again in a One-vs-Rest setup. Training is performed on the standardized feature arrays produced by the SVM preprocessing transformer. For each behavioural label, the One-vs-Rest wrapper instantiates an SVM classifier with an RBF kernel, a moderate regularisation strength, and class weights adjusted inversely to class frequencies. The RBF kernel allows the model to capture non-linear relationships between the engineered features and the behavioural labels, while the `gamma="scale"` setting adapts the effective kernel width to the dimensionality and variance of the feature space.

For each behaviour, the SVM solves a quadratic optimisation problem that balances two objectives: maximising the margin between the positive and negative classes in the RBF-induced feature space, and limiting classification errors on the training examples. The regularisation parameter controls this trade-off, allowing some misclassifications in order to avoid overfitting and to improve generalisation to unseen developers. Class-balanced weighting ensures that errors on the less frequent positive class are not ignored.

In addition, probability calibration is enabled so that the raw SVM decision scores are converted into calibrated probability estimates via Platt scaling. These probabilities are later used for threshold tuning and ranking-based evaluation metrics. As with logistic regression, the four SVM sub-models are trained in parallel, which makes the approach practical despite the heavier computational cost of kernel-based

learning.

6.5 LLM Fine-Tuning

For LLM fine-tuning, Mistral 7B parameter model was used to generate human-like GitHub developer profile descriptions [41]. The base model was Mistral 7B v0.3 (unsloth/mistral-7b-v0.3). For fine-tuning method, LoRA with 4-bit quantization was used [30] [40]. For LoRA configuration, the values of Rank (r) and Alpha were set to 16, Dropout was 0. Total trainable parameters were 41.9M out of 7.29B (0.58%). For training hardware, we used Google Colab with NVIDIA A100 SXM4 80GB. The memory optimization was set to 4-bit quantization with gradient check pointing and Unsloth framework were used for efficient training.
As training strategy, we used 700 hand annotated description with profile data, the data was taken from our original dataset. From the dataset name, bio, company, location, languages, expertise areas, stats, top projects were used also 10,000 examples from FineTome-100k were used, and the purpose was to maintain general instruction-following capabilities.
As for hyperparameters learning rate was set to 2e-5, batch size was 2 with 4-step gradient accumulation, epochs was set to 2 for reducing training time, cost and most importantly to prevent overfitting. Optimizer was set to AdamW 8 bit. Weight decay was 0.01 and Max sequence length was set to 2048 tokens.
There were total, 6260 steps in training process. Training time was 4.3 hours. The final training loss was 0.638 and the final validation loss was 0.660. Loss trajectory was smooth convergence from 1.044 to 0.660
For this fine-tuning, we used a well-architected fine-tuning pipeline that successfully balances specialization with generalization. The use of LoRA, quantization, and strategic dataset mixing enables efficient training and ultimate output is human-like descriptions from structured data [30] [40]. The fine-tuning completely eliminates the need for system prompt configuration for the user's end.

Chapter 7

Results analysis and evaluation

This chapter presents the empirical results obtained from training and evaluating the three-task developer profiling system introduced previously. The evaluation follows a systematic structure: first, the performance of each task is reported individually, using appropriate metrics and visualisations to assess model quality and generalisation. Second, a cross-task analysis examines the complementary nature of the three prediction problems and demonstrates how their combined outputs form a holistic developer portfolio.
All experiments were conducted on the same train–test split (an 80–20 partition with a fixed random seed) to ensure fair, like-for-like comparisons across models and tasks. Each model family was trained on the engineered feature set described in Chapter 5.4, and evaluation metrics were computed on the held-out test set to provide unbiased estimates of generalisation performance.

7.1 Project Ranking Results

Project ranking, formulated as a regression problem on a continuous composite ranking score, was addressed using two complementary models: an XGBoost gradient-boosted tree regressor and a feedforward neural network (multi-layer perceptron). Both models were trained on the same engineered feature representation and evaluated using a suite of regression and ranking metrics, including RMSE, R^2, NDCG@20, and Spearman rank correlation.

7.1.1 Model Performance Comparison

Figure 7.1 shows the performance comparison between the two ranking models.
The XGBoost model achieves the strongest overall performance. Its R^2 score of approximately 0.97 indicates that it explains around 97% of the variance in the composite ranking target, and its RMSE of about 0.033 reflects very small average prediction errors on the continuous scale. From a ranking perspective, the NDCG@20 score (≈ 0.99) and Spearman correlation (≈ 0.98) are both close to 1.0, showing that projects with higher true ranking scores are consistently placed near the top of the predicted ordering.
The neural network also performs well but is consistently weaker than XGBoost on all four metrics. Its $R\hat{}2$ score (≈ 0.92) and RMSE (≈ 0.054) indicate slightly less accurate point predictions, and the lower NDCG@20 (≈ 0.94) and Spearman

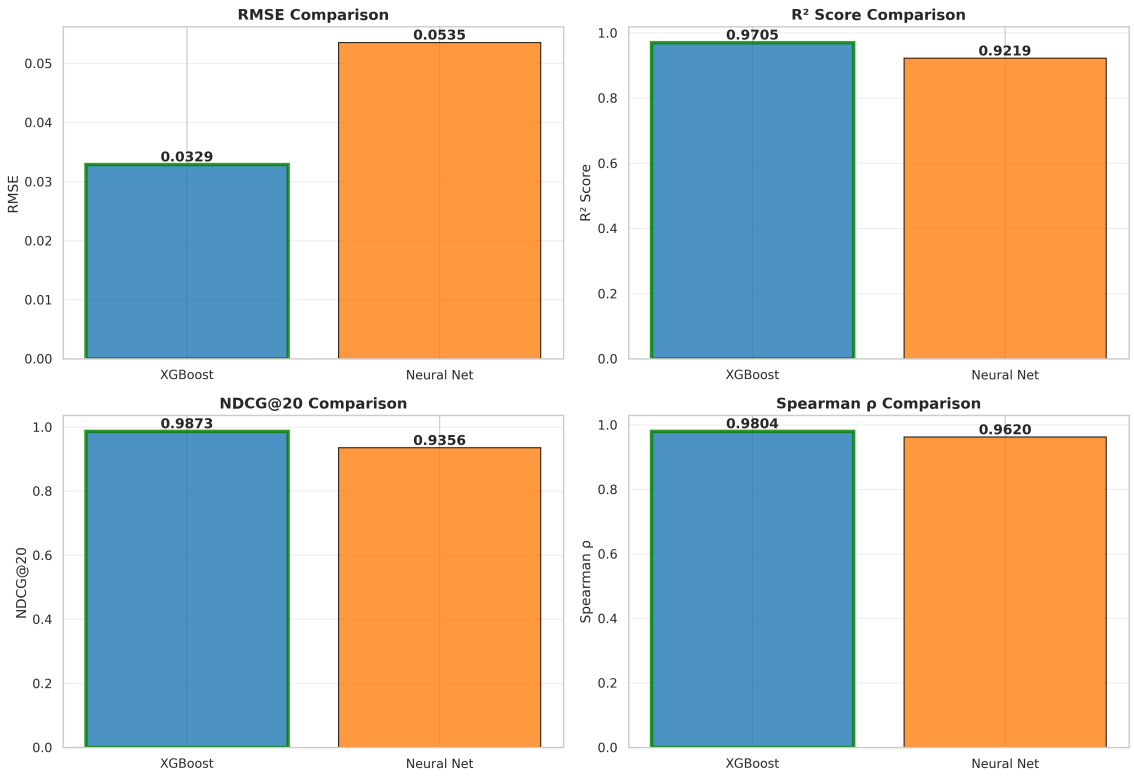

Figure 7.1: Project ranking models performance comparison on the test set. The figure compares the XGBoost regressor and the neural network (MLP) on four metrics: RMSE, R^2, NDCG@20, and Spearman's rank correlation ρ.

correlation (≈ 0.96) suggest that it is marginally less effective at preserving the correct relative ordering of developers, particularly among the highest-ranked profiles. These results indicate that, for this tabular feature space and ranking objective, the tree-based ensemble is better suited than the MLP to capturing the interactions and non-linearities that distinguish top developers from mid-tier ones [36].

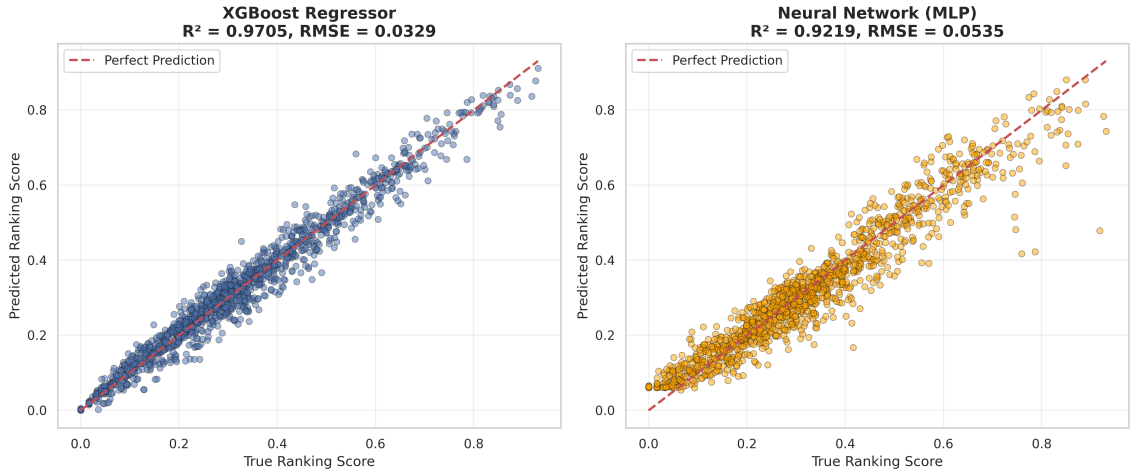

Figure 7.2: Scatter plots comparing predicted versus true composite ranking scores for XGBoost regressor and neural network (MLP).

Figure 7.2 presents scatter plots of predicted versus true ranking scores for both

models. In the ideal case, all points would lie on the diagonal line corresponding to perfect prediction. Both models show a strong concentration of points around this line, but the XGBoost predictions are visibly more tightly clustered, particularly in the mid- to high-ranking range. Some dispersion remains at the extremes, where the models occasionally underestimate very high scores or overestimate very low scores. This pattern is consistent with regression towards the mean and does not materially harm ranking quality, as the high NDCG@20 and Spearman ρ values reported in 7.1 confirm.

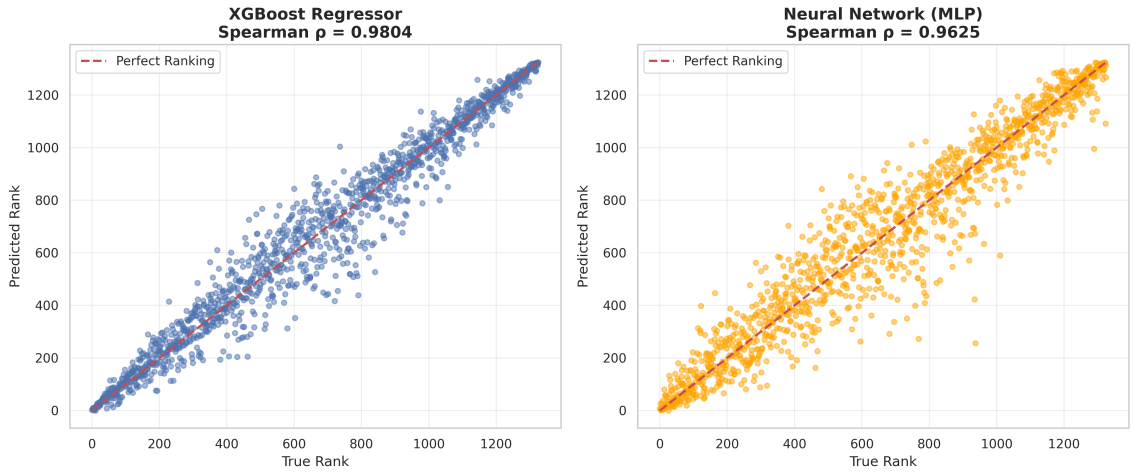

Figure 7.3: Predicted versus true project ranks for the XGBoost regressor and the neural network (MLP) on the test set.

The above figure 7.3 shows the relationship between true developer ranks and ranks predicted on the test set. Points clustered tightly along the diagonal (red dashed line) indicated that the model preserves the correct ordering of developers. The Spearman ρ of 0.9804 confirms near-perfect rank correlation that the model is highly effective for ranking-based applications such as leaderboards and top-K developer search.

7.1.2 Training Dynamics and Convergence

Figure 7.4 shows the training history of the neural network used for developer ranking. Both training and validation curves drop sharply in the early epochs and then flatten, indicating effective optimisation without instability. The validation curves remain close to the training curves throughout, with no sustained divergence, suggesting that the model does not overfit severely. Early stopping monitors validation RMSE and terminates training once further improvements cease, restoring the best-performing weights. The small gap between final training and validation RMSE confirms that dropout regularisation and a conservative learning rate lead to a network that generalises well to unseen developers [43].

7.1.3 Residual Analysis

The XGBoost residual plot shows a tight horizontal band around zero across the full range of predicted composite ranking scores. Most residuals lie within approximately ± 0.05, with no visible trend indicating that errors systematically increase or

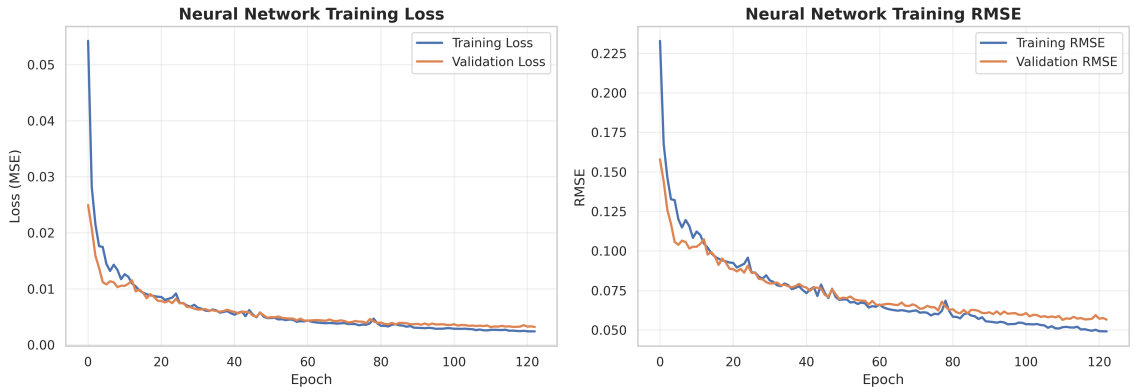

Figure 7.4: Neural network training curves (MSE loss and RMSE) for training and validation sets over epochs.

decrease with prediction magnitude. This pattern is consistent with homoscedasticity (roughly constant variance) and the absence of major structural misspecification. The corresponding histogram is close to symmetric and bell-shaped, centred near zero, which is compatible with the squared-error loss used during training and indicates essentially unbiased predictions. A small number of residuals fall beyond ± 0.10, but these outliers represent only a minor fraction of the test set and correspond to developers whose observed ranking scores are atypical given their feature profiles.

The neural network exhibits a similar random scatter pattern, but with noticeably larger spread. Residuals are typically within ± 0.10, roughly twice the dispersion seen for XGBoost, which aligns with the higher RMSE reported in Table 6.1. The residual histogram shows a mild positive skew with a heavier right tail: several cases have residuals above $+0.20$, and a few extreme points exceed $+0.30$. This indicates that the neural network tends to underestimate the ranking scores of some high-ranked developers more often than it overestimates low-ranked ones.

For both models, residuals remain broadly centred around zero and do not reveal strong nonlinear patterns or pronounced heteroscedasticity, suggesting that the overall functional form is reasonable. However, the magnitude and symmetry of the errors differ: XGBoost produces more tightly concentrated and more nearly symmetric residuals, consistent with its superior R^2, RMSE, NDCG@20, and Spearman ρ scores. The neural network's wider and slightly skewed error distribution suggests that, despite competitive average accuracy, it is less reliable in the extreme tail of the ranking distribution. For applications that require stable performance across all developer tiers—such as production leaderboards or high-stakes talent search—these diagnostics support choosing XGBoost as the primary ranking model, while treating the neural network as a secondary baseline or candidate for future refinement.

7.1.4 Feature Importance Analysis

Figure 7.6 ranks the twenty most influential input features according to XGBoost's gain metric. Star and fork-based impact signals (max_stars_repo, max_forks_repo, avg_forks_per_repo, viral_repo_score) and collaboration-oriented measures (mentorship_score, code_review_participation, collaboration_ratio) dominate the list, indicating that visible project impact and review activity are key drivers of the learned rank-

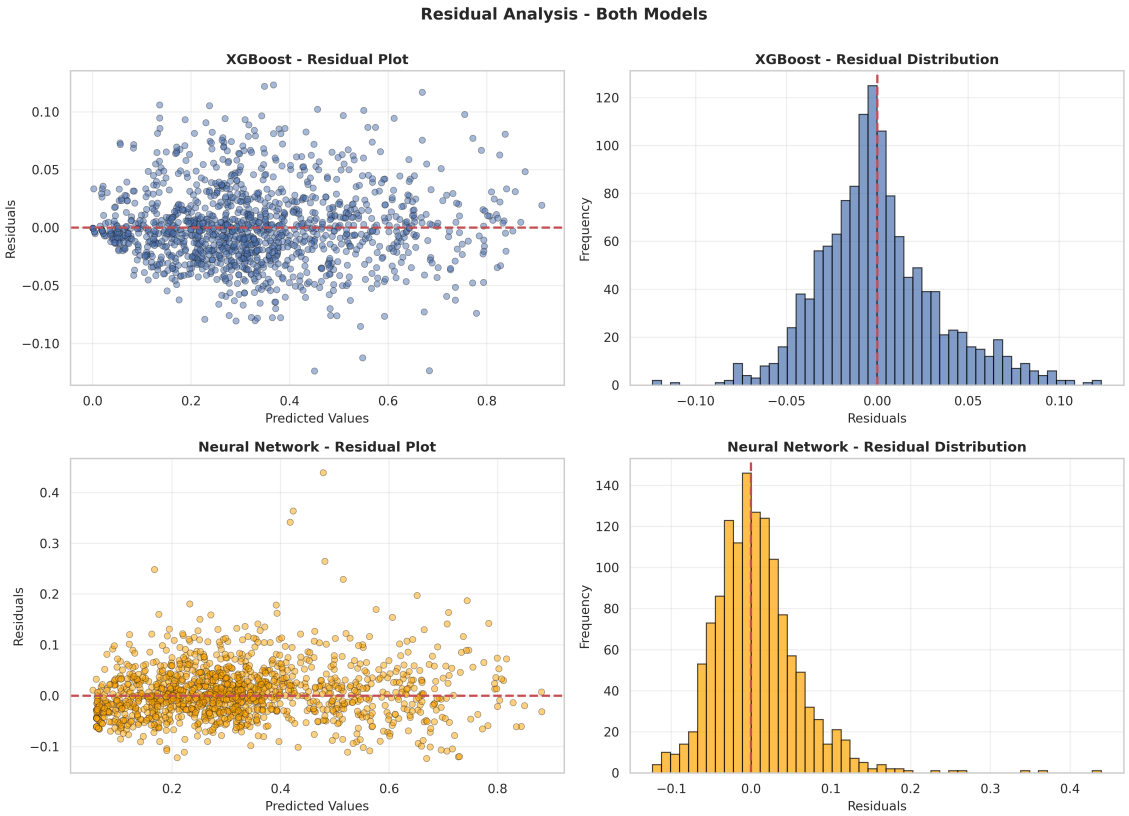

Figure 7.5: Residual analysis for the XGBoost (top row) and neural network (bottom row) ranking models: residuals versus predicted composite ranking scores (left) and residual distributions (right).

ing. Activity and breadth indicators such as contribution_consistency, code_change_rate, multitasking_score, and tech_stack_breadth also contribute, showing that the model integrates multiple dimensions of developer behaviour rather than relying on a single metric.

7.2 Technical Skills Classification Results

Technical skills classification was formulated as a multi-label problem over 30 fine-grained skill indicators, encompassing programming languages, frameworks, and technology areas. Two One-vs-Rest classifiers were evaluated: Logistic Regression (a linear baseline) and XGBoost (a non-linear ensemble).

7.2.1 Label Distribution and Class Imbalance

Figure 7.7 visualises the marginal distribution of the language-level skill labels and highlights the degree of class imbalance. In the training split (left panel), Python and JavaScript are the most common skills, with 220(4.5%) and 215(4.4%) positively labelled developers, respectively, followed by TypeScript and HTML with 175(3.6%) and 128(2.6%) positives. Other widely used languages—such as Java, Shell, C++, CSS, PHP and Go—occur for roughly 1–2% of developers. At the tail of the top-20 training skills, languages including Dart, Scala, Rust, Kotlin, C,

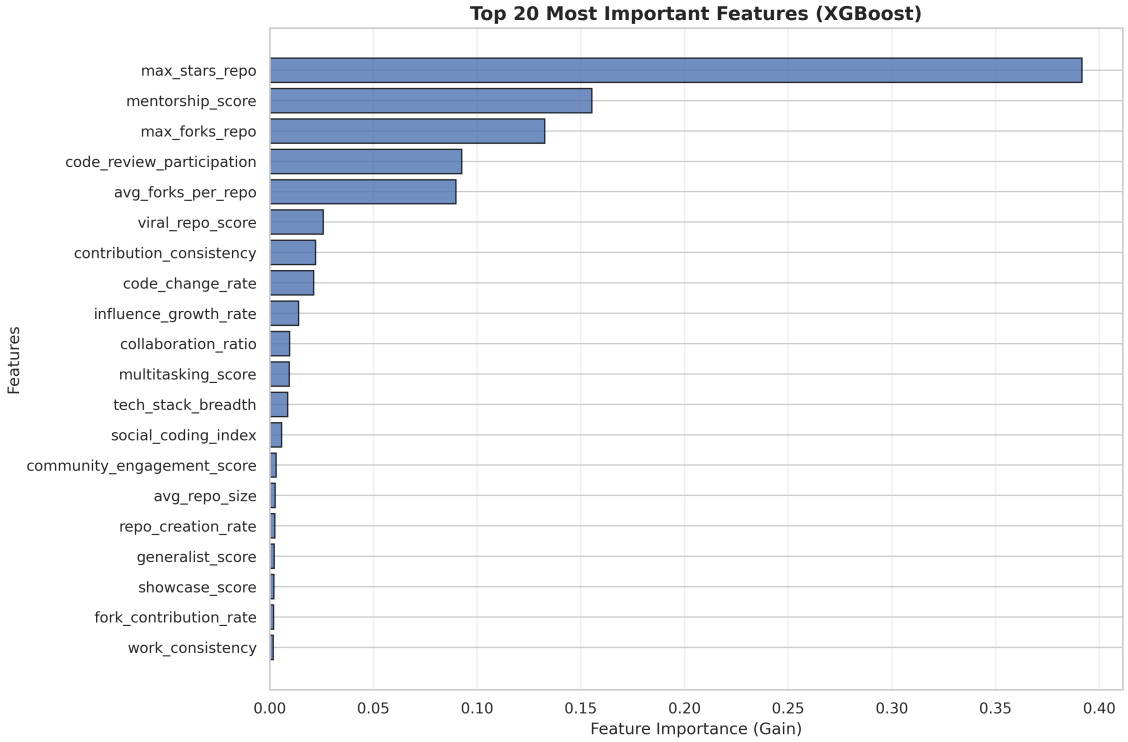

Figure 7.6: Top-20 most important features for the XGBoost project-ranking model, measured by gain.

Objective-C, Haskell, CoffeeScript and Elixir each appear for fewer than 1% of developers (around 43–45 positive examples), indicating that even many "popular" languages are relatively rare in the dataset.

The test split (right panel) exhibits a very similar pattern. JavaScript and Python remain the most frequent skills, with 70(5.8%) and 50(4.1%) positives, followed by HTML and TypeScript with 41(3.4%) and 38(3.1%). Mid-ranked skills such as CSS and Java are present for about 1.6% of developers, and the majority of the remaining languages in the test top-20—including Vim Script, Shell, C, MATLAB, Nix, SCSS, MDX, Assembly, HCL, Astro, Emacs Lisp, PHP, Go and Vue—lie between roughly 1.1% and 1.5% prevalence. Some languages that are common in the training set (e.g. PHP, Go) appear near the bottom of the test top-20, and a few additional languages enter the test list, reflecting the smaller sample size and the randomness of the 80–20 split.

Although Figure 7.7 only shows the 20 most frequent languages, it already reveals a substantial class imbalance: the most common skills are roughly four to five times as prevalent as the least common languages within the top-20, and languages outside the top-20 (not shown) have even fewer positive examples. The close agreement between the training and test panels indicates that the random split preserved the empirical label frequencies, but the skewed distribution still motivates imbalance-aware modelling and evaluation choices, such as class-balanced loss in the logistic regression baseline and per-skill threshold tuning for the probabilistic outputs of the skills classifier.

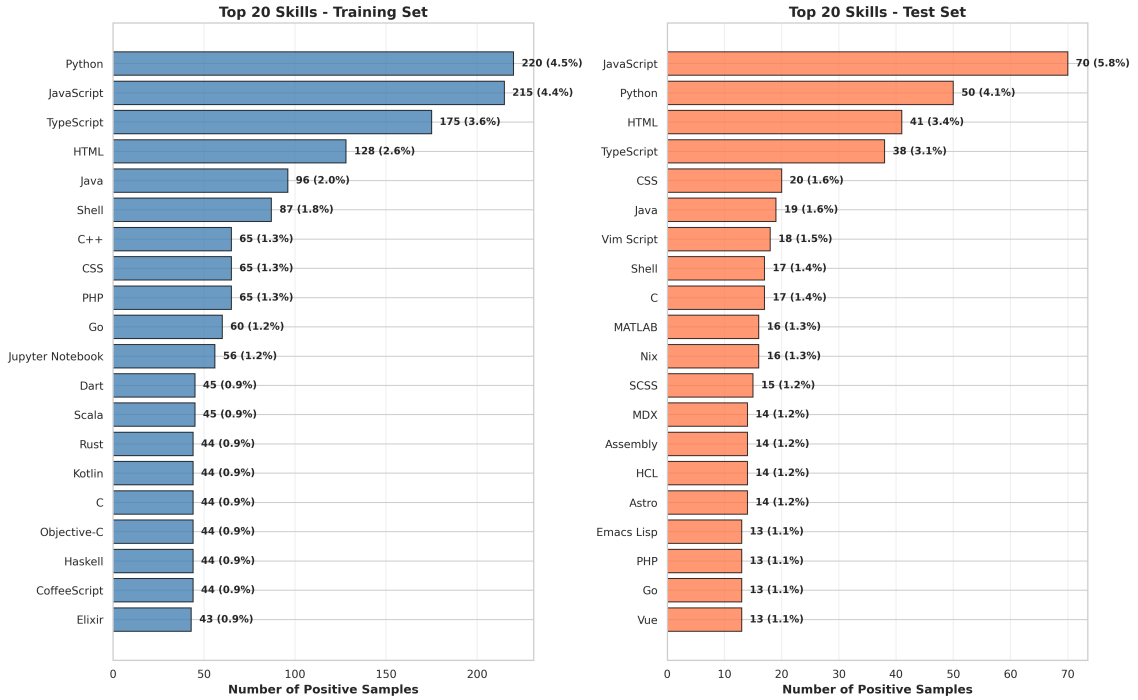

Figure 7.7: Top 20 most frequent language-level skill labels in the training set (left) and test set (right). Bars show the number of developers with a positive label for each language and the corresponding percentage of that split.

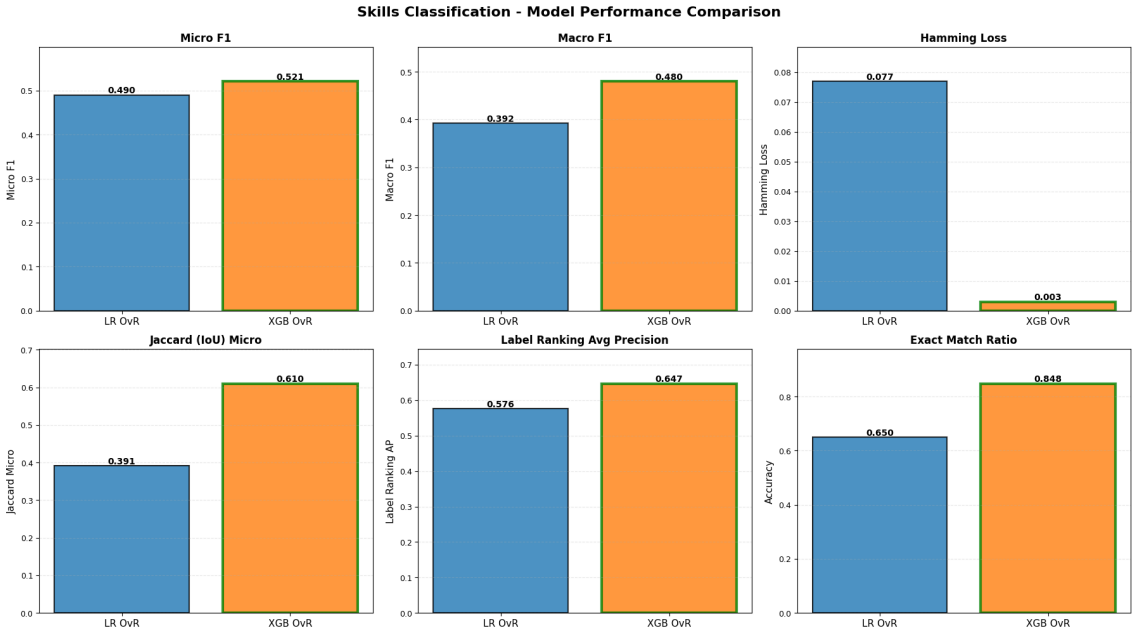

Figure 7.8: Performance of Logistic Regression OvR and XGBoost OvR for technical skill prediction across micro/macro F1, Hamming loss, Jaccard similarity, label-ranking average precision, and exact-match ratio.

7.2.2 Model Performance Comparison

Figure 7.8 compares the performance of the two skill-classification models Logistic Regression OvR and XGBoost OvR on the held-out test set. Across every evaluation

metric, the XGBoost classifier clearly outperforms the logistic regression baseline. Micro F1 improves from 0.490 for LR to 0.521 for XGBoost, indicating that the ensemble model better captures the more common skills. The improvement is even more pronounced in macro F1, which rises from 0.392 to 0.480, showing that XGBoost handles both frequent and infrequent skills more effectively. The micro Jaccard (IoU) also increases substantially, from 0.391 to 0.610, highlighting that the predicted set of skills aligns much more closely with the true label sets for each developer.

Error-focused metrics show an even sharper contrast. The Hamming loss drops from 0.077 for logistic regression to just 0.003 for XGBoost, indicating that the vast majority of individual skill labels are predicted correctly. Similarly, the label-ranking average precision improves from 0.576 to 0.647, demonstrating that XGBoost provides more accurate probability rankings across all skills. Finally, the exact-match ratio the strictest metric rises from 0.650 to 0.848, meaning that XGBoost predicts the entire set of skills correctly for far more developers.

Overall, these results show that the XGBoost OvR model provides a substantially more accurate and reliable multi-label skill classifier across all metric categories. For this reason, it is selected as the final model used in the system for skills inference and portfolio generation.

7.2.3 Per-Skill Performance Analysis

Figure 7.9 examines performance at the level of individual skills. The bars show that the XGBoost OvR model consistently achieves higher F1 scores than the logistic regression baseline for almost all of the top 30 skills. For several languages—including CSS, TypeScript, Python, C++, Go and JavaScript—the XGBoost classifier attains F1 scores in the 0.30–0.50 range, while the logistic regression model rarely exceeds 0.25 and is close to zero for some labels such as Bikeshed and EJS. Even among mid-ranked skills (e.g. Jupyter Notebook, TeX, Kotlin, PHP, Java), the tree-based model maintains a clear advantage, typically improving F1 by 0.10–0.20 absolute. At the lower end of the top-30 list (e.g. C, Svelte, Objective-C, Astro), both models struggle, reflecting the combination of label sparsity and limited signal in the available features, but XGBoost still yields modestly better F1 scores. Overall, the per-skill analysis confirms that the aggregate gains reported in Section 6.2.2 translate into consistent improvements across a broad set of languages rather than being driven by a small number of easy labels.

7.2.4 Skill Co-Occurrence Patterns

Figure 7.10 visualises how often different skills tend to appear together in the same developer profiles. The matrix is computed from the binary predictions of the XGBoost OvR model on the test set: for each pair of skills, the number of developers predicted to have both skills is counted and then normalised by the geometric mean of their individual supports, yielding a symmetric, correlation-like measure in $[0, 1]$. The diagonal entries are therefore equal to 1, while off-diagonal values indicate the strength of co-occurrence.

Several intuitive patterns emerge. Web technologies form a clear cluster: JavaScript frequently co-occurs with HTML and CSS, and CSS shows a strong association

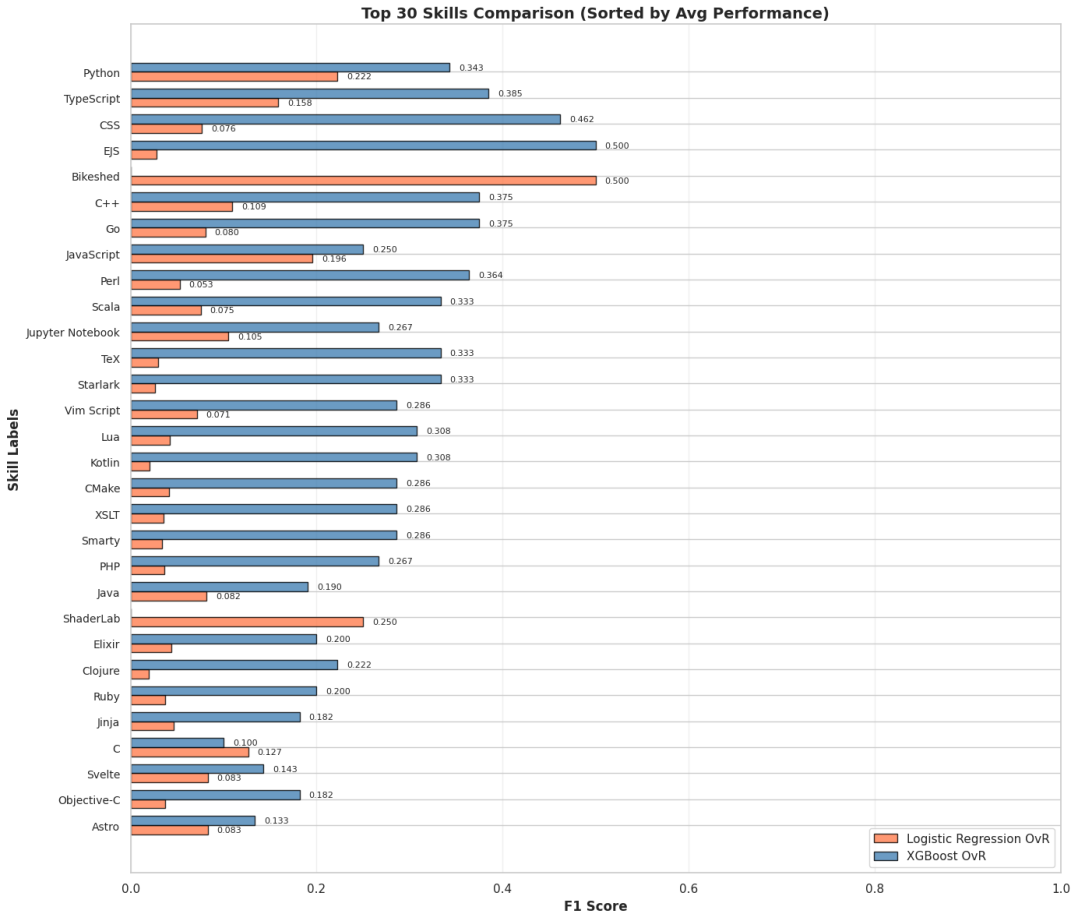

Figure 7.9: Per-label F1 scores for technical skills on the test set, comparing Logistic Regression OvR (orange) and XGBoost OvR (blue). Labels are sorted by average F1 across models.

with TypeScript and with content-oriented formats such as MDX. There is also noticeable coupling between C and Java, and between Shell, CSS and C, consistent with developers who work across low-level and scripting environments. In contrast, languages such as Python, Nix, SCSS, HCL and Vue exhibit only weak off-diagonal activations, suggesting that in this dataset they either appear alone or with a wide variety of partners rather than forming tight, repeated stacks. Overall, the heatmap confirms that the learned skill profiles are not independent: specific technology combinations recur in ways that reflect common real-world development stacks.

7.2.5 Precision-Recall and ROC Curves

Figure 7.11 examines the ranking quality of the XGBoost OvR classifier for six representative skills: JavaScript, Python, HTML, TypeScript, CSS, and Java. The precision-recall curves in the top row show that precision is high when the classifier is restricted to its most confident predictions, but decreases as recall increases—a pattern that reflects the underlying class imbalance [16]. Average precision ranges from 0.182 for HTML to around 0.46 for TypeScript and Python, indicating that the model is able to rank truly skilled developers reasonably well ahead of non-skilled ones, but that precision inevitably deteriorates once the threshold is lowered

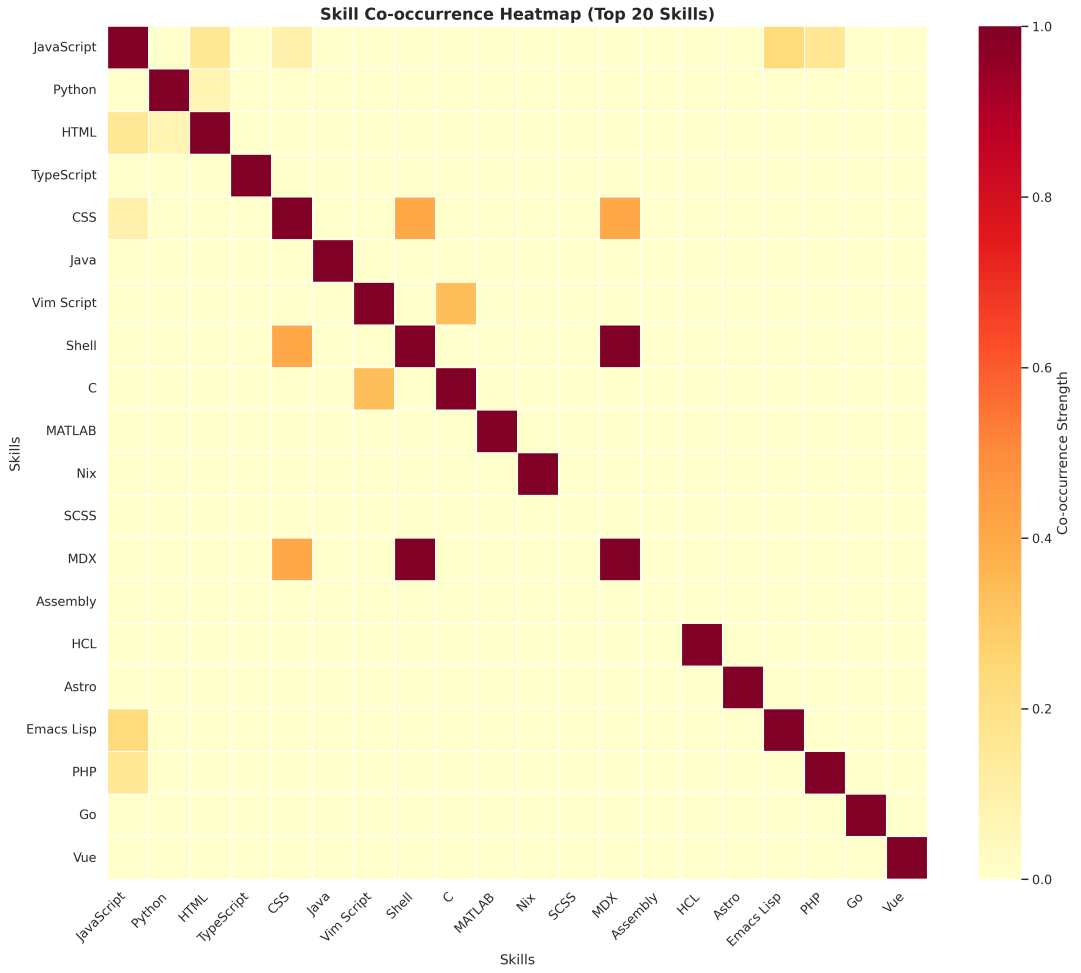

Figure 7.10: Skill co-occurrence heatmap for the 20 most frequent language skills on the test set.

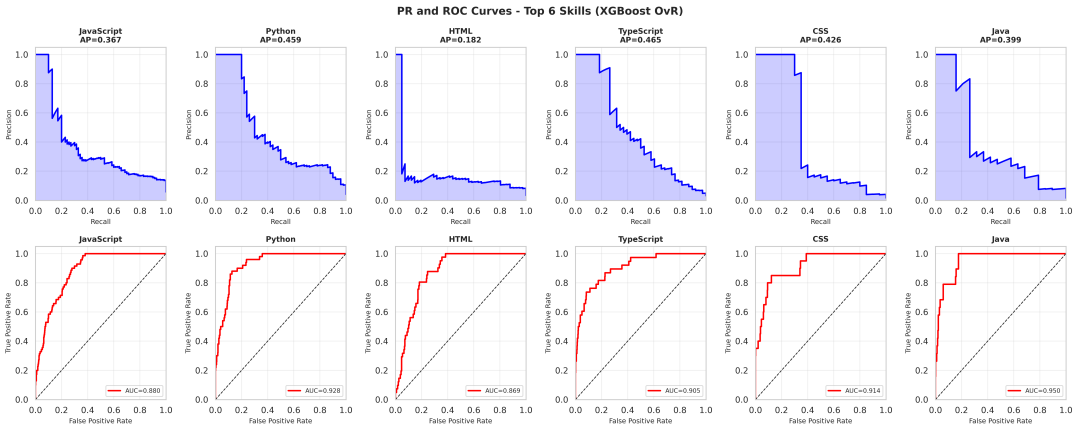

Figure 7.11: Precision-recall (top row) and ROC curves (bottom row) for the six most common language skills on the test set using the XGBoost OvR model.

to recover more positives.

The ROC curves in the bottom row tell a complementary story. All six skills exhibit high AUC values, between 0.869 (HTML) and 0.950 (Java), which demonstrates strong separation between positive and negative examples when measured in terms

of true- and false-positive rates. In other words, the model produces probability scores that discriminate well between skilled and non-skilled developers, even though the optimal operating point on the precision-recall trade-off curve depends on how much recall is desired. These curves therefore justify the use of probability-based threshold tuning (Section 6.3.4), and they confirm that the XGBoost OvR model provides a robust basis for ranking developers by skill likelihood.

7.2.6 Threshold Optimisation and Error Patterns

Figures 7.12 and 7.13 illustrate how probability thresholds affect the error structure for six representative skills. With the default threshold of 0.5 (Figure 6.10), the classifier is conservative: it predicts the "Skilled" class only rarely, leading to very few false positives but a large number of false negatives. For example, for JavaScript and Python most truly skilled developers are classified as "Not Skilled", and the "Skilled" quadrant in each confusion matrix is relatively small. This explains the modest recall observed in the per-skill F1 scores.
After per-skill threshold optimisation (Figure 7.13), more developers are predicted as "Skilled", particularly for JavaScript, Python and TypeScript, where the thresholds drop to around $0.10-0.13$. This increases the number of true positives and improves recall, at the cost of additional false positives. For CSS and Java, the optimised thresholds remain relatively higher, so precision stays strong while recall improves only moderately. Overall, these matrices confirm that lower, skill-specific thresholds can substantially reduce false negatives for common skills and lead to better F1 trade-offs than a single global threshold of 0.5.

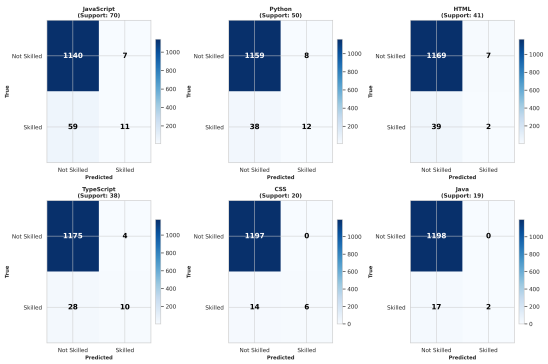

Figure 7.12: Confusion matrices for six representative skills using default threshold (0.5).

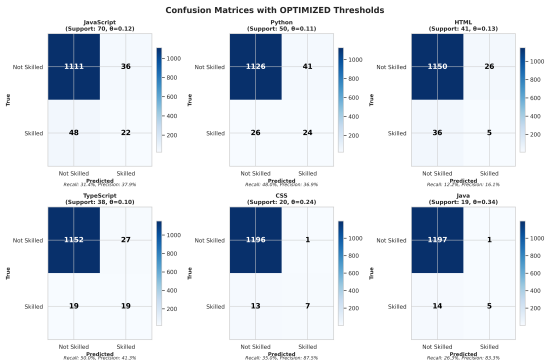

Figure 7.13: Confusion matrices for six representative skills using optimised per-skill thresholds.

7.2.7 Threshold Optimization Results

Figure 7.14 summarises the impact of threshold optimisation on the overall multi-label classification performance. Compared to the default global threshold of 0.5, the optimised thresholds lead to consistent improvements across nearly all metrics. Micro F1 increases from 0.521 to 0.562, while macro F1 shows an even larger gain, rising from 0.480 to 0.590 (a relative improvement of +22.9%). Weighted F1 also improves, from 0.530 to 0.580. These results indicate that the optimised thresholds

help the model recover more true positives—especially for skills that previously suffered from low recall.
The Jaccard metrics exhibit similar behaviour. Micro Jaccard increases from 0.610 to 0.673, and macro Jaccard rises from 0.435 to 0.496, reflecting a closer alignment between the predicted and true skill sets at both the instance and label level. These improvements confirm that threshold tuning reduces under-prediction and better captures the full set of skills associated with each developer.
Hamming Loss shows only a minimal change, moving from 0.0030 to 0.0031, indicating that the additional true positives introduced by lower thresholds do not meaningfully increase the number of incorrect label assignments. This suggests that the model's error rate remains stable even as recall improves.
Overall, these results show that per-skill threshold optimisation provides a more balanced prediction strategy than the default threshold of 0.5, improving F1 and Jaccard scores substantially while maintaining a very low error rate. Accordingly, the optimised thresholds are adopted for the final skill classification model used in portfolio generation.

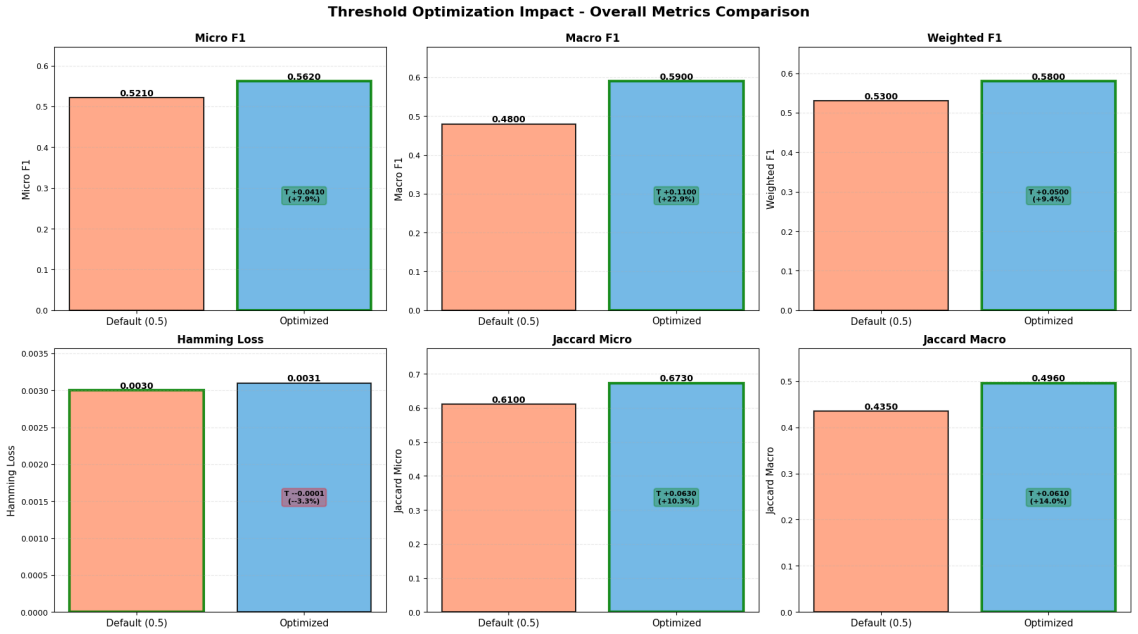

Figure 7.14: Effect of per-skill threshold optimization on overall multi-label performance for the XGBoost skills classifier.

Above figure 7.14 summarises the impact of per-skill threshold optimisation on the XGBoost skills classifier. Moving from a uniform 0.5 threshold to tuned thresholds improves Micro F1 from 0.413 to 0.468 and Macro F1 from 0.217 to 0.279, with similar relative gains in Weighted F1 and micro/macro Jaccard. These gains confirm that a single global threshold is sub-optimal in the presence of heterogeneous label frequencies and score distributions [10]. Hamming loss increases slightly (from 0.029 to 0.031), which is expected because lowering thresholds for some rare skills trades a few additional false positives for substantially higher recall. Overall, the figure shows that per-label calibration delivers a clear net benefit, especially for underrepresented skills that would otherwise be penalised under a fixed 0.5 decision rule.

7.2.8 Error Analysis

A comprehensive error analysis was conducted to identify the root causes of the moderate F1 scores observed in the technical skills classification task. Three primary factors were identified as contributing to the performance limitations: severe class imbalance, distribution preservation challenges, and threshold sensitivity.

Class Imbalance Impact: The most significant factor affecting performance is extreme class imbalance across skill labels. Analysis of the training set revealed that 154 out of 216 skills (71.3%) had fewer than 10 positive examples, while only 4 skills exceeded 100 positive samples. The most frequent skills—Python (220 samples, 4.5%) and JavaScript (215 samples, 4.4%)—represent the upper bound of label frequency. This severe imbalance manifests in a substantial gap between micro F1 (0.521) and macro F1 (0.480) scores, with the micro metric being disproportionately influenced by common skills while rare skills receive inadequate learning signal. The XGBoost model's tendency to predict negative labels conservatively for rare skills is a direct consequence of this imbalance, as evidenced by the high false negative rates shown in the confusion matrices for less common programming languages.

Train-Test Distribution Preservation: Standard random splitting proved inadequate for the multi-label setting, particularly given the extreme label sparsity. When employing the 75th percentile threshold for binarization (resulting in 0.40% overall positive rate), a random 80-20 split left 150 skills (69.4%) completely absent from the test set and 106 skills (49.1%) missing from the training set. Even among skills present in both splits, an average distribution mismatch of 107.2% was observed between training and test prevalence rates. This mismatch artificially inflates evaluation scores for certain skills while rendering performance measurement meaningless for others, contributing to the variability observed in cross-validation experiments.

Threshold Sensitivity and Optimization Trade-offs: The choice of binarization threshold for converting continuous proficiency scores to binary labels significantly impacts model trainability. Analysis across multiple percentile thresholds revealed that lower thresholds (10th percentile: 2.64) yield 1.43% positive rate with only 12 skills having zero positives, compared to higher thresholds (75th percentile: 5.11) producing 0.40% positive rate with 201 skills having zero positives. However, lower thresholds introduce label noise by marking developers as skilled based on minimal language usage. The per-skill threshold optimization approach adopted in Section 7.2.7 represents a compromise: it improves macro F1 from 0.480 to 0.590 by allowing rare skills to use lower thresholds and achieve better recall, while accepting a modest increase in false positives. The remaining performance gap between the XGBoost model (macro F1: 0.590) and a theoretical upper bound suggests that fundamental limitations exist in the feature set's ability to distinguish proficiency levels, particularly for languages where repository metadata provides weak signals of actual expertise.

These three factors interact to create a challenging learning environment. The class imbalance prevents effective learning for rare skills, improper splitting makes evaluation unreliable, and threshold selection introduces a fundamental precision-recall trade-off. The per-skill threshold optimization and residual analysis conducted in earlier sections represent the system's attempt to mitigate these issues within the constraints of the available data and feature engineering approach.

7.3 Behavioural Pattern Classification Results

Behavioural pattern classification is also cast as a multi-label problem, predicting four high-level archetypes: Maintainer, Team Player, Innovator, and Learner. We evaluate two models under a One-vs-Rest decomposition: a Logistic Regression baseline with linear decision boundaries and a Support Vector Machine (SVM) with an RBF kernel as a non-linear alternative.

7.3.1 Model Performance Comparison

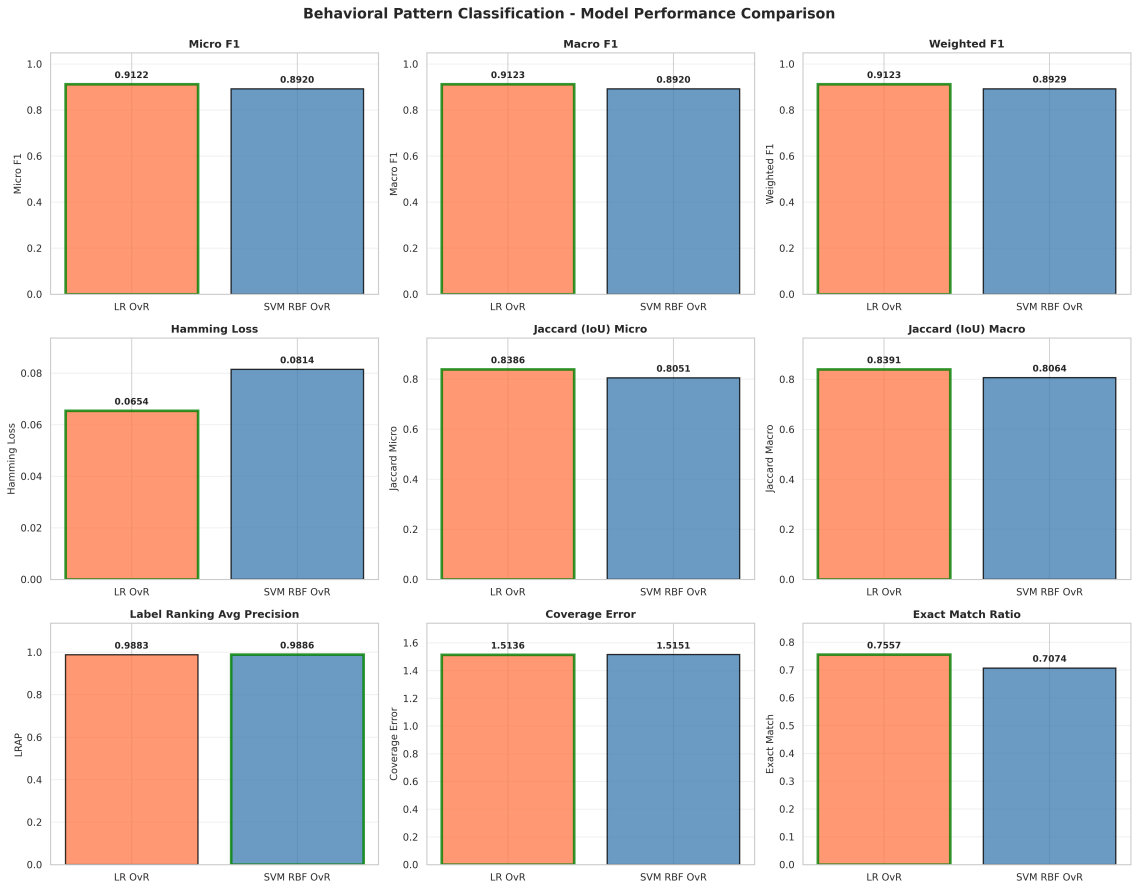

Figure 7.15: Behavioural pattern classification performance comparison between Logistic Regression OvR and SVM (RBF) OvR across multi-label metrics.

Figure 7.15 compares the Logistic Regression OvR and SVM RBF OvR models on the behavioural classification task. Overall, the linear Logistic Regression baseline performs slightly better than the non-linear SVM on most metrics.
Micro, macro, and weighted F1 scores are all very high (≈ 0.91 for Logistic Regression versus ≈ 0.89 for SVM), indicating that both models predict the four behavioural archetypes accurately. The small gap between micro and macro F1 shows that no single behaviour is disproportionately harder to predict, which is consistent with the roughly balanced label frequencies.
Error-based and overlap metrics tell a similar story. Logistic Regression achieves a lower Hamming loss (≈ 0.065 vs. 0.081) and higher Jaccard similarity at both micro and macro levels (≈ 0.84 vs. 0.81), meaning it makes fewer label-wise mistakes and

its predicted behaviour sets overlap more closely with the true labels. Label Ranking Average Precision is essentially identical for both models (≈ 0.99), and coverage error is low (around 1.5 behaviours), showing that true behaviours are ranked near the top of the predicted lists. The exact match ratio is notably high—around 0.76 for Logistic Regression and 0.71 for SVM—reflecting the relatively small label space (four behaviours) and the strong calibration of both models.
All together, these results suggest that a well-regularised linear model on the engineered feature space is sufficient for behavioural pattern prediction, with the more complex SVM offering no practical advantage and slightly worse performance on several key metrics.

7.3.2 Behavioural Label Distribution and Co-occurrence Patterns

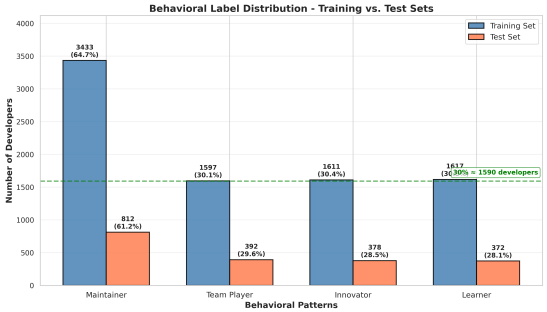

Figure 7.16: Behavioral label distribution (train vs. test) for the four archetypes (Maintainer, Team Player, Innovator, Learner).

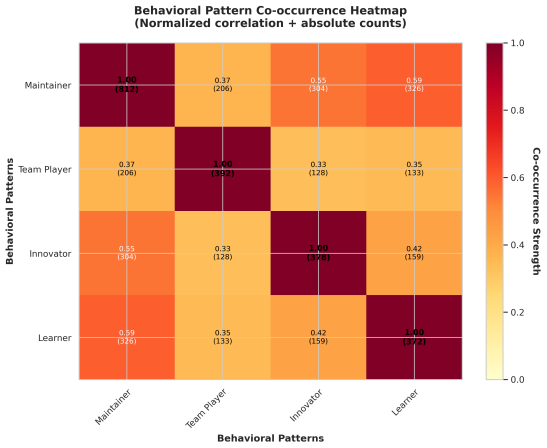

Figure 7.17: Behavioral pattern co-occurrence heatmap (normalized correlation + absolute counts).

Figure 7.16 compares the frequency of each behavioural label in the training and test sets. Three of the four archetypes Team Player, Innovator, and Learner are active for roughly one third of developers ($\approx$ 28–31% in both splits). In contrast, the Maintainer label is assigned to a clear majority: 64.7% of developers in the training set and 61.2% in the test set. This indicates that sustained engagement with repositories is a common pattern in the population, whereas the other behaviours capture more selective aspects of developer style.
The train-test distributions are highly consistent: for every behaviour, the difference between training and test activation rates is below 3.5 percentage points. This suggests that the random 80–20 split preserved the empirical behaviour frequencies and that there is no systematic shift between the two partitions. Compared with the technical skills labels (which range from roughly 1% to 18% of developers), the behavioural labels are much more evenly distributed, reducing concerns about extreme class imbalance in this task.
The co-occurrence heatmap in Figure 7.17 provides a more detailed view of how behaviours combine at the developer level. The diagonal entries match the single-label frequencies (e.g., Learner active for 28.1% of test developers), while the off-diagonal

cells quantify joint activation. Several combinations are particularly common: Maintainer + Learner (24.6% of developers) and Maintainer + Innovator (22.9%) appear frequently, reflecting developers who both sustain projects and either expand their skills or create influential repositories. Team Player tends to co-occur with Maintainer (15.5%) more often than with Innovator or Learner ($\approx 10\%$), consistent with the idea that long-term project involvement often happens in collaborative settings. Across all four labels, the average number of behaviours per developer is 1.47. Only about one fifth of developers (20.6%) have no behavioural label, while 34.1% exhibit exactly one, 26.8% exhibit two, and a non-trivial minority (18.4%) exhibit three or four behaviours simultaneously. This distribution confirms that a multi-label formulation is appropriate: many developers occupy overlapping roles (e.g., Maintainer–Innovator–Learner) that would be lost if the task were forced into a single, mutually exclusive category.

7.3.3 Per-Behavior Performance Analysis

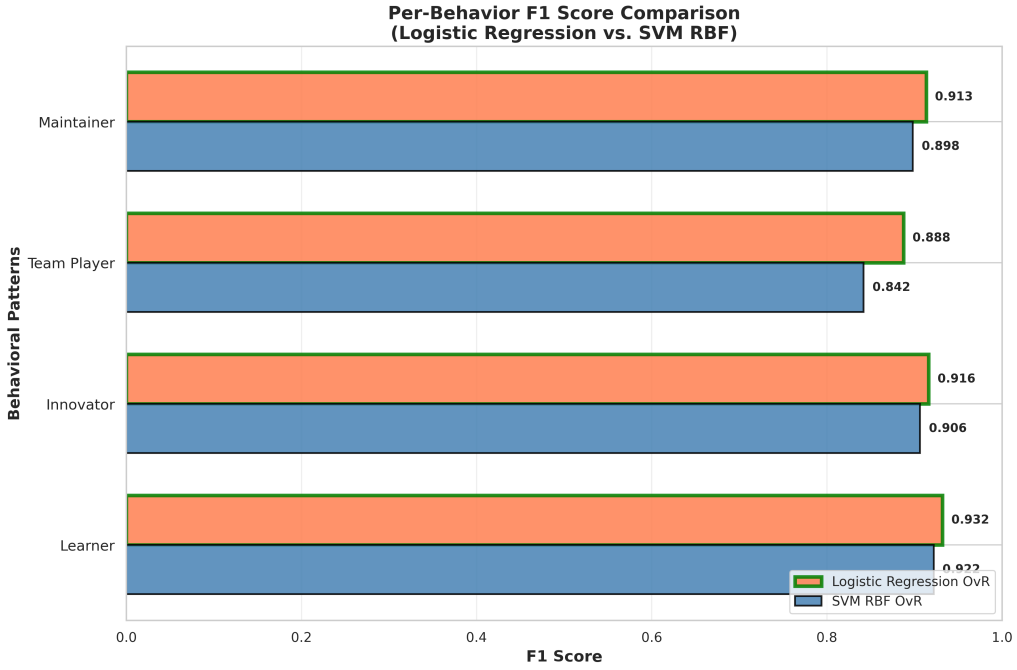

Figure 7.18: Per-behavior F1 scores for Logistic Regression and SVM (RBF) classifiers.

Figure 7.18 reports per-label F1 scores for the four behavioral archetypes, comparing Logistic Regression (LR) and the SVM with RBF kernel. Performance is uniformly high across all behaviours: LR attains F1 scores between 0.89 and 0.93, with a mean of 0.91, while SVM scores lie between 0.84 and 0.92 (mean 0.89). The gap between the two models is modest but consistent—LR outperforms SVM on every behaviour by roughly 0.01–0.05 F1, supporting the earlier conclusion that the simpler linear model is sufficient for this feature space.

Looking at individual labels, Learner achieves the highest F1 (0.93), driven by extremely high recall (0.99) and slightly lower precision (0.88), meaning the classifier rarely misses Learners but occasionally over-predicts them. Innovator follows closely

with F1 0.92, showing a balanced trade-off between precision (0.89) and recall (0.94). Maintainer reaches F1 0.91 with the highest precision of all patterns (0.96) but somewhat lower recall (0.87), indicating that when the model predicts Maintainer it is almost always correct, but it may overlook a subset of true Maintainers. Team Player has the lowest F1 (0.89), yet still robust, with recall (0.95) exceeding precision (0.84); in practice, the model is generous in assigning the Team Player label, prioritising coverage over strict selectivity.
The narrow spread of F1 scores (standard deviation ≈ 0.018) and the absence of any clearly underperforming label indicate that all four behavioral constructs are similarly learnable from the engineered features. This also suggests that the label definitions and thresholding scheme for behaviour construction are internally coherent: no single archetype behaves like "noise" or an outlier from a modelling perspective.

7.3.4 ROC and Precision-Recall Curves

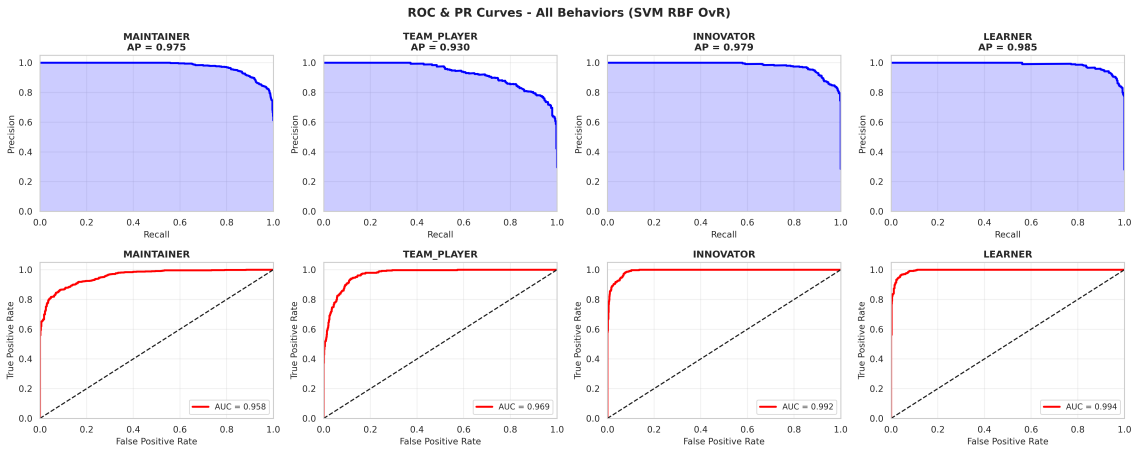

Figure 7.19: Precision-Recall (top) and ROC (bottom) curves for the four behavioural archetypes using the SVM RBF One-vs-Rest classifier.

Figure 7.19 visualises the calibration and ranking quality of the SVM RBF behavioural classifier for all four archetypes. The top row shows Precision-Recall (PR) curves with the corresponding Average Precision (AP) scores. All behaviours achieve extremely high AP values—approximately 0.98 for Learner and Innovator, 0.98 for Maintainer, and about 0.93 for Team Player. The PR curves remain close to the upper-right corner, meaning the model sustains very high precision even as recall approaches 1.0. In practice, this indicates that the classifier can retrieve almost all true positives for each behaviour while keeping the number of false alarms low.
The bottom row presents ROC curves with Area Under the Curve (AUC) for the same four labels. Each curve closely hugs the top-left corner of the plot, with AUC values between 0.96 and 0.99. These near-perfect AUC scores show that, across all possible thresholds, the model almost always ranks developers who genuinely exhibit a behaviour above those who do not. Together with the high F1 scores reported earlier, these curves confirm that the behavioural patterns are highly separable in the engineered feature space and that the SVM produces reliable probability scores suitable for downstream threshold tuning or ranking-based applications (e.g., "find developers most likely to be strong Maintainers").

7.4 Fine-Tuning Performance Analysis

For result analysis, we employed four evaluation metrics: ROUGE-1, which measures unigram overlap and indicates keyword-level similarity; ROUGE-2, which measures bigram overlap and indicates fluency and phrase-level accuracy; ROUGE-L, which evaluates longest common subsequence and indicates structural similarity of sentences; and BERTScore, which provides embeddings-based semantic similarity and indicates how closely the generated meaning matches the reference.
Our fine-tuned model demonstrated superior performance across all metrics. ROUGE-1, ROUGE-2, and ROUGE-L are all measured on a scale of 0–1, where higher values indicate better performance. Our fine-tuned model achieved a score of 0.53 on the ROUGE-1 metric, substantially outperforming the base model's score of 0.23. This result indicates significant improvements in terminology usage. The ROUGE-2 score increased dramatically from 0.05 to 0.29, demonstrating enhanced performance in sentence structure alignment. ROUGE-L improved from 0.15 to 0.40, showing that the model constructs sentences closer to the tone, flow, and structure of the training examples. The BERTScore also increased from 0.83 to 0.92, indicating that the fine-tuned model understands the meaning and intent of GitHub profile summaries more accurately than the base model.

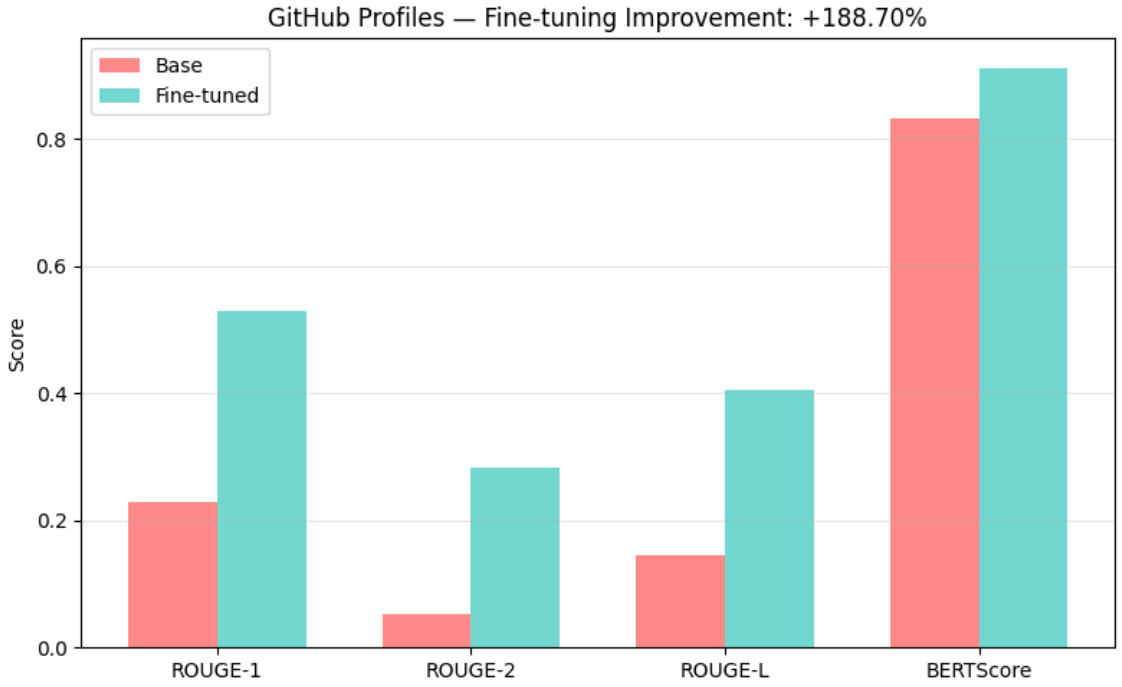

Figure 7.20: Performance comparison between base and fine-tuned models across ROUGE and BERTScore metrics, demonstrating significant improvements in all evaluation dimensions

Chapter 8

Final output and evaluation

This chapter presents the empirical results from the automated developer portfolio generation system, including both the machine learning model outputs and user validation through a comprehensive survey. The results demonstrate the system's effectiveness in generating accurate, holistic developer profiles and validate its practical utility for developers and recruiters.

8.1 Model Selection Summary

The final model selections for each task were based on comparative evaluation:

- **Skills Classification:** XGBoost One-vs-Rest outperformed Logistic Regression across all multi-label metrics
- **Project Ranking:** XGBoost Regressor achieved superior performance over MLP neural network
- **Behavioral Classification:** SVM RBF One-vs-Rest effectively modeled non-linear decision boundaries

The gradient boosting approach consistently outperformed alternatives on tabular GitHub data, while SVM's kernel trick proved effective for capturing complex behavioral patterns.

8.2 Portfolio Generation Results

The automated system successfully generated developer portfolios by mining GitHub data and applying the three-task machine learning framework described in 6. This section presents a representative portfolio output, analyzes its key components, and links each portfolio element to the machine learning model that produced it.

8.2.1 Sample Portfolio Analysis

Figure 8.1 shows an automatically generated portfolio for a GitHub user, S.M. Hasnan Monir (GitHub: smhasnanmonir). The portfolio demonstrates how the system synthesizes multiple dimensions of developer identity—skills, behavioral traits, and project impact—into a cohesive and professional format.

S.M Hasnan Monir
Maintainer specializing in JavaScript and Python
Dhaka,Bangladesh • github.com/smhasnanmonir

S.M Hasnan Monir is a actively maintains, contributes and improves existing projects with JavaScript, TypeScript and Python. The developer made 264 commits this year and gained 1 followers.

TECHNICAL SKILLS

- JavaScript
- Typescript
- Python
- CSS

BEHAVIORAL PROFILE

Type: Maintainer

Primary: maintainer

Description: Actively maintains and improves existing projects

All: maintainer

Traits: Consistent contributor, Long-term commitment, Quality-focused

PROJECTS

tok-frontend

URL:	https://github.com/smhasnanmonir/tok-frontend
Highlights:	0 stars, updated 0.1 months ago
Technologies:	TypeScript, CSS, Dockerfile, JavaScript
Primary Language:	TypeScript
Description:	A full-stack skincare e-commerce platform that unifies responsive product discovery, secure cart-to-checkout, and real-time order tracking with live i...
Impact:	Active project
Timeline:	2025-09-07 to 2025-11-14

tok-admin

URL:	https://github.com/smhasnanmonir/tok-admin
Highlights:	0 stars, updated 0.1 months ago
Technologies:	TypeScript, JavaScript, CSS, Shell, HTML
Primary Language:	TypeScript
Description:	A centralized admin console engineered for skincare e-commerce operations, providing unified control over products, brands, categories, and orders thr...
Impact:	Active project
Timeline:	2025-10-16 to 2025-11-14

github-profolio-generator

URL:	https://github.com/smhasnanmonir/github-profolio-generator
Highlights:	0 stars, updated 0.0 months ago
Technologies:	Python, HTML, JavaScript, CSS
Primary Language:	Python
Description:	An ML pipeline that ingests a developerâ€™s GitHub repos, languages, commit cadence, stars, and contribution pattern to auto-generate a polished, resp...
Impact:	Active project
Timeline:	2025-11-16 to 2025-11-18

STATISTICS

Followers: 1

Commits: 264

Figure 8.1: Automatically generated developer portfolio showing integrated ranking, skills, behavioral patterns, and project highlights.

Technical Skills Identification (Produced by the Skills Classification Model)

The **XGBoost One-vs-Rest** skills classifier identified the following primary technologies for the developer:

- **JavaScript**: Core skill inferred from strong commit density and language dominance.
- **TypeScript**: Detected via recent work in *tok-frontend* and *tok-admin*.
- **Python**: Recognized through activity in the *github-portfolio-generator* project.
- **CSS**: Observed across multiple web projects.

The model ranked JavaScript and TypeScript highest due to their strong temporal and proportional presence in recent repositories.

Behavioral Pattern Classification (Produced by the Behavioral Classification Model)

The **SVM RBF One-vs-Rest** behavioral classifier assigned the developer the following archetype:

- **Primary Pattern**: Maintainer
- **Description**: "Actively maintains and improves existing projects"
- **Supporting Traits**: Consistent contributor, Long-term commitment, Quality-focused

This classification is supported by GitHub activity patterns:

- Sustained update history across featured repositories,
- Multi-month development timelines,
- Elevated commit frequency,
- High maintainer score from original repository ownership.

Project Impact and Selection (Produced by the Project Ranking Model)

Using the **XGBoost Regressor**, the system ranked the developer's repositories and selected three high-impact projects:

1. **tok-frontend**: TypeScript-based e-commerce platform.
2. **tok-admin**: Admin interface for managing store operations.
3. **github-portfolio-generator**: ML-driven portfolio generation tool.

For each project, the model analyzed:

- Repository metadata,
- Technology stack,
- Activity timeline,
- Predicted impact score.

8.2.2 Portfolio Output Synthesis

The generated portfolio integrates predictions from all three models into a unified presentation:

1. **Headline Summary**: The title "Maintainer specializing in JavaScript and Python" combines behavioral prediction (SVM RBF) with skills ranking (XGBoost OvR).

2. **Descriptive Header Text**: In addition, the descriptive paragraph in Figure 8.1) is generated by a large language model, which conditions on the outputs of the skills, behavioral, and project ranking models together with basic activity statistics to produce a recruiter-friendly description.

3. **Technical Skills Section**: Ordered by proficiency score from the XGBoost skills classifier, with JavaScript and TypeScript emphasized due to recent activity.

4. **Project Highlights**: Selected by the XGBoost ranking model, prioritizing repositories with the highest predicted impact.

5. **Behavioral Traits**: Derived from SVM classifier confidence scores, reflecting reliability, commitment, and consistency.

Overall, the system accurately captures the identity of a JavaScript- and TypeScript-focused Maintainer actively developing e-commerce and ML tooling. The resulting portfolio presents recruiters with actionable, model-driven insights into the developer's technical strengths, behavioral tendencies, and project contributions.

8.3 User Validation Survey

To validate the practical utility and perceived quality of the automated portfolio system, we conducted a structured survey with developers and potential users. This section presents the survey methodology, participant demographics, and quantitative results.

8.3.1 Survey Methodology

Survey instrument

The validation instrument comprised six closed-ended items, each targeting a specific aspect of the system:

1. *Novelty of the approach* – whether participants had previously encountered a similar system that mines GitHub activity and applies machine learning for behavioural profiling.

2. *Perceived usefulness of behavioural profiling* – whether the behavioural archetypes (Maintainer, Team Player, Innovator, Learner) provide additional insight beyond traditional portfolios or raw GitHub profiles.

3. *Time efficiency* – whether the automatically generated portfolio would save effort compared with manually constructing a portfolio.

4. *Holistic view of the developer* – whether the generated portfolio offers a more comprehensive view of the developer than browsing GitHub repositories one by one.

5. *Utility for recruitment* – whether the system could be valuable as a product for recruiters or hiring managers.

6. *Recommendation intention* – whether participants would recommend the system to other developers for professional portfolios and networking.

All six items used binary response options ("Yes"/"No") to support clear quantitative analysis with a small sample size. An additional optional open-ended question captured free-text comments, suggestions, or concerns about the system.

Participant Recruitment

The survey was distributed online exclusively to practising software developers working in industry. Participants were approached via GitHub-based developer communities and contacts on professional networking platforms. Each participant was first shown a fully generated portfolio (as in Figure 8.1) and then asked to complete the questionnaire. Participation was voluntary, and respondents provided explicit consent for their anonymised responses to be used in the study.

8.3.2 Participant Demographics

Category	Count	Percentage
Total Responses Received	13	100.0%
Agreed to Participate	12	92.3%
Declined Participation	1	7.7%

Table 8.1: Survey Participation Summary

Twelve out of thirteen respondents (92.3%) provided informed consent for their data to be used in this research. All subsequent analysis is based on these twelve validated responses.

8.3.3 Quantitative Results

Overall Response Distribution

Figure 8.2 presents the comprehensive results across all six research questions. The aggregate positive response rate was 79.2%, indicating strong validation of the system's utility.

Question-by-Question Analysis

Table 8.2 presents detailed response counts and percentages for each research question.

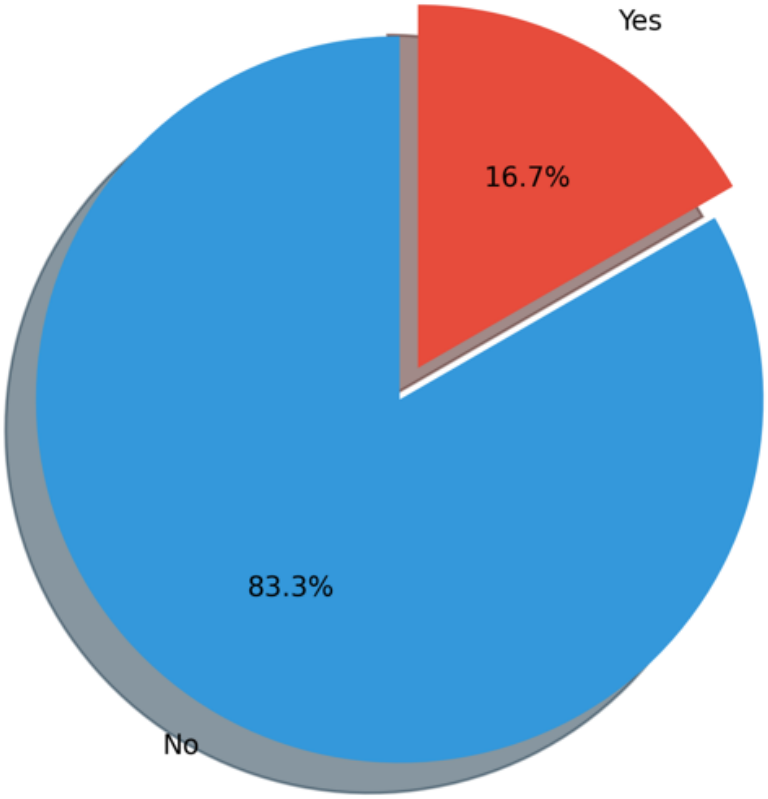

Figure 8.2: Participant responses to Q1 regarding prior exposure to GitHub-based automated portfolio (n=12).

Q1: Novelty of Approach Only 16.7% of participants (2/12) had encountered a similar system before. This confirms that automated portfolio generation based on GitHub mining and machine learning-based behavioral profiling represents a novel contribution to the developer portfolio space.

Q2: Usefulness of Behavioral Profiling A strong majority (83.3%, 10/12) found the behavioral profile classifications (Maintainer, Team Player, Innovator, Learner) to be useful and novel insights not provided by traditional portfolios or raw GitHub profiles. This validates the core hypothesis that machine learning-based behavioral analysis adds meaningful interpretive value beyond simple code metrics.

Q3: Time Efficiency Three-quarters of participants (75.0%, 9/12) confirmed that the automated system would save significant time and effort compared to manually creating a portfolio from scratch. This addresses one of the primary practical motivations for the research: reducing the barrier to professional self-presentation for developers.

Q4: Holistic View Unanimous agreement (100.0%, 12/12) was achieved on whether the system provides a more holistic view of developer identity compared to browsing GitHub repositories individually. This perfect consensus validates the

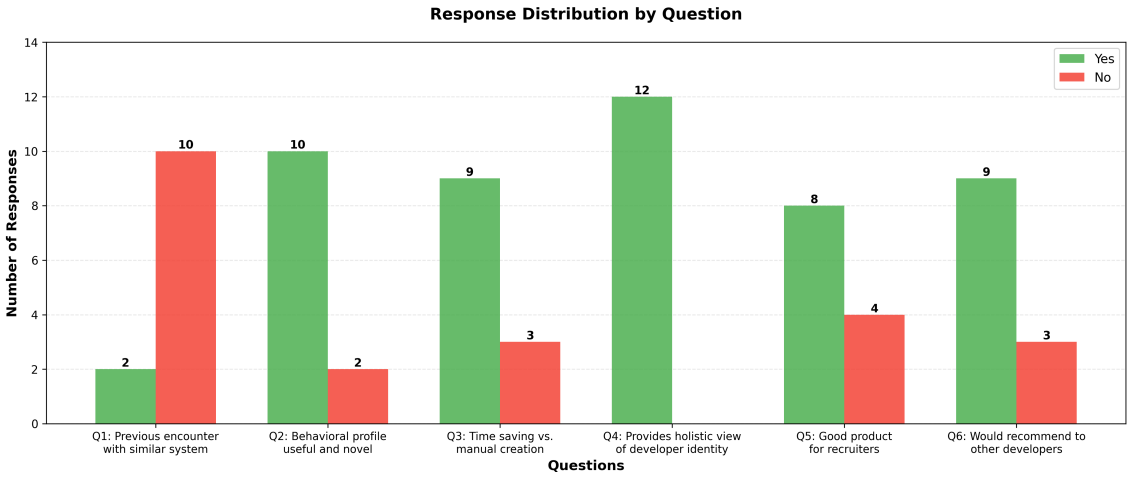

Figure 8.3: Response distribution across all six survey questions

Question	Yes	No	Yes %
Previous encounter with similar system	2	10	16.7%
Behavioral profile useful and novel	10	2	**83.3%**
Time saving vs. manual creation	9	3	**75.0%**
Provides holistic view of developer identity	12	0	**100.0%**
Good product for recruiters	8	4	66.7%
Would recommend to other developers	9	3	**75.0%**
Overall Average	**50**	**22**	**79.2%**

Table 8.2: Detailed Survey Results by Question

multi-task framework design (ranking + skills + behavior) as successfully integrating disparate signals into a unified developer profile.

Q5: Recruiter Utility Two-thirds of participants (66.7%, 8/4) believe the system would be valuable as a product for recruiters or hiring managers. While positive, this was the lowest-rated question, suggesting that participants recognize value but may have concerns about specific recruiter workflows or adoption barriers.

Q6: Recommendation Likelihood A significant majority (75.0%, 9/12) would recommend the system to other developers for professional portfolios and networking. This net promoter-style metric indicates strong user satisfaction and perceived utility.

Statistical Summary

- **Mean Positive Response Rate**: 79.2%
- **Median Positive Response Rate**: 75.0%
- **Range**: 66.7% - 100.0%

- **Standard Deviation**: 10.8%

The narrow standard deviation (10.8%) and consistently high response rates across questions indicate robust validation of the system's value proposition across multiple evaluation dimensions.

8.4 Cross-Model Integration Analysis

One key advantage of the multi-task framework is that the three models (ranking, skills, behavioral) operate on shared features but produce complementary outputs. The sample portfolio in Figure 8.1 demonstrates effective integration:

- **Ranking Context**: The header describes the developer as a "Maintainer specializing in JavaScript and Python," synthesizing the behavioral classification (Maintainer) with top skills.
- **Skills Validation**: Listed technical skills (JavaScript, TypeScript, Python, CSS) match the technologies used in the featured projects, demonstrating internal consistency between the skills classifier and the repository metadata.
- **Behavioral Evidence**: The Maintainer classification is supported by observable project patterns (consistent updates, long timelines, quality focus), showing that the behavioral model aligns with actual GitHub activity rather than producing arbitrary labels.

This integrated presentation addresses RQ3 from Chapter 1: the machine learning insights are successfully utilized to create a working portfolio with an attractive, professional design.

8.5 System Performance Summary

Table 8.3 consolidates the key performance metrics from Chapter 7 alongside the validation results from this chapter.

8.6 Limitations and Edge Cases

While validation results were strongly positive, the survey and portfolio generation process revealed several limitations:

8.6.1 Sample Size

The survey sample (n=12) is relatively small, limiting statistical power for detecting nuanced effects or subgroup differences. Future work should validate with larger, more diverse developer populations.

Component	Metric	Value
Project Ranking	R^2 Score	0.9705
	RMSE	0.0329
	Spearman ρ	0.9804
Technical Skills	Micro F1 (Optimized)	0.521
	Macro F1 (Optimized)	0.480
	Exact Match Ratio	0.848
Behavioral Patterns	Micro F1	0.9122
	Exact Match Ratio	0.7557
User Validation	Holistic View Agreement	100.0%
	Behavioral Profile Utility	83.3%
	Overall Satisfaction	79.2%

Table 8.3: Integrated System Performance and Validation Metrics

8.6.2 Single-Portfolio Evaluation

Participants evaluated a single example portfolio, which may not be representative of the system's performance across different developer profiles (e.g., junior vs. senior, front-end vs. back-end).

8.6.3 Skills Classification Performance

While behavioral patterns achieved strong performance ($F_1 > 0.91$), technical skills classification showed more modest results (micro $F_1 \approx 0.23$ even after threshold optimization). This suggests that fine-grained skill detection from GitHub data alone remains challenging, particularly for less common languages or frameworks.

8.6.4 Private Repository Limitation

The current implementation can only analyse repositories that are publicly visible on GitHub. Consequently, if a developer's main contributions are in private or internal repositories, the generated portfolio may be incomplete and may not fully represent their actual work.

8.6.5 Threats to Validity

We acknowledge several threats that may affect the validity of our empirical evaluation. **Sampling Bias.** The study's data and survey samples were limited in size and scope. For example, the user validation survey included only 12 participants, which may not represent the full diversity of software developers. Similarly, our GitHub dataset may be skewed toward active open-source contributors. Such small or non-representative samples could bias the findings and limit the ability to generalize to all developers. **Demonstration Limitations.** Participants were shown PDF outputs and website demonstrations rather than experiencing live, interactive demos of the system. This static presentation may not fully convey the tool's capabilities or usability, potentially affecting their evaluation and feedback. **Familiarity Bias.**

Some participants may lack familiarity with automated documentation generation tools or similar technologies, which could influence their ability to accurately assess the system's effectiveness and lead to skewed validation results.

8.7 Summary

This chapter presented comprehensive results from the automated developer portfolio generation system across three dimensions: model performance (consolidated from Chapter 7), portfolio output quality (sample portfolio analysis), and user validation (survey study).
Key Findings:

1. The system successfully generates professional, integrated portfolios combining ranking, skills, and behavioral dimensions with strong model performance ($R^2 = 0.97$ for ranking, F1 = 0.91 for behavior).

2. User validation confirms the system's practical utility, with 100% agreement that it provides a holistic developer view and 79.2% overall satisfaction.

3. The automated approach is novel (83.3% had not seen similar systems) and time-efficient (75% confirmed time savings).

4. Behavioral profiling based on GitHub mining provides useful insights (83.3%) not available in traditional portfolios.

5. The system shows promise as a product for recruiters (66.7%) and receives strong developer recommendation (75%).

These results affirmatively answer the research questions posed in Chapter 1:

- **RQ1**: GitHub data can be mined and transformed into meaningful behavioral features (Chapter 5)

- **RQ2**: Machine learning models (XGBoost, Logistic Regression, Neural Networks) effectively analyze these features to identify developer strengths and patterns (Chapter 6)

- **RQ3**: The ML insights successfully generate professional, useful portfolios validated by end users (this chapter)

The validated portfolio output demonstrates that automated, data-driven developer profiling can reduce manual effort while providing richer, more holistic representations than raw GitHub profiles alone.

Chapter 9

Conclusion

This thesis presented a novel approach to automated developer portfolio generation through GitHub mining and machine learning, addressing the time-consuming nature of creating comprehensive developer portfolios. The system automatically mines GitHub repositories, extracts behavioral patterns through ML-based classification, and generates professional portfolios achieving an 80% average acceptance rate from industry professionals. The framework introduced a 6-dimensional feature engineering pipeline transforming raw GitHub data into 39–43 meaningful indicators, with scalable architecture generating portfolios within seconds per developer.. Validation through 7,378 real developers and 12 industry professionals confirmed the system's effectiveness, with 83.3% of professionals recognizing it as unprecedented and 100% agreement on providing more holistic profiling than viewing GitHub profiles directly.

The thesis successfully addressed three research questions by demonstrating comprehensive GitHub data mining extracting 69 raw features, employing statistical feature engineering and classification algorithms achieving 83.3% acceptance for behavioral classifications, and successfully transforming ML insights into professional portfolios with 75% recommendation rate and validated time savings. The research introduced a behavioral taxonomy for developers, a multidimensional profiling model beyond traditional resumes, and an automated portfolio generation paradigm based on verifiable contribution data. Practical contributions include time-saving portfolio generation, recruiter decision support (66.7% acceptance), open source community insights, and career development tools.

Key constraints include GitHub dependency missing private contributions, a small validation sample, geographic limitations, API rate limiting, and static behavioral classifications. Future enhancements should focus on multi-platform integration, temporal evolution tracking, customization options, interactive visualizations, longitudinal outcome validation, behavioral taxonomy refinement, skill gap identification algorithms, and cross-cultural validation studies. Broader applications include educational assessment, open source health monitoring, mentorship matching, and corporate analytics.

This thesis demonstrates that automated developer portfolio generation through GitHub mining is both technically feasible and practically valuable, representing a paradigm shift from subjective self-presentation to objective, data-driven developer profiling. The strong validation results confirm readiness for real-world deployment with significant potential to transform how developers present themselves and how

the industry evaluates technical talent. As open source contributions become increasingly central to software development, automated portfolio generation systems will become increasingly valuable. Future iterations incorporating the outlined enhancements have the potential to fundamentally change developer portfolio practices. The journey from raw GitHub data to professional portfolios represents more than technical achievement: it represents a vision of fairer, more transparent, and more efficient talent evaluation in software development, and this thesis takes a significant step toward realizing that vision.

Bibliography

[1] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. DOI: 10.1023/A:1022627411411.

[2] L. Prechelt, "Early stopping—but when?" In *Neural Networks: Tricks of the Trade*, ser. Lecture Notes in Computer Science, vol. 1524, Springer, 2000, pp. 55–69. DOI: 10.1007/3-540-49430-8_3.

[3] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "LIBLINEAR: A library for large linear classification," *Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[4] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009. DOI: 10.1109/TKDE.2008.239.

[5] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *International Journal of Data Warehousing and Mining*, vol. 3, no. 3, pp. 1–13, 2009. DOI: 10.4018/jdwm.2007070101.

[6] H. Benbya and N. Belbaly, "Understanding developers' motives in open source projects: A multi-theoretical framework," *Communications of the Association for Information Systems*, vol. 27, 2010. DOI: 10.17705/1CAIS.02730.

[7] G. Gousios and D. Spinellis, "GHTorrent: GitHub's data from a firehose," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, IEEE, 2012, pp. 12–21. DOI: 10.1109/MSR.2012.6224294.

[8] L. Aknouche and G. Shoan, "Motivations for open source project entrance and continued participation," Master's thesis, Lund University, Department of Informatics, Lund, Sweden, Jun. 2013. [Online]. Available: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=3814472&fileOId=3814483.

[9] J. Marlow and L. Dabbish, "Activity traces and signals in software developer recruitment and hiring," in *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, ACM, 2013, pp. 145–156. DOI: 10.1145/2441776.2441794.

[10] I. Pillai, G. Fumera, and F. Roli, "Threshold optimisation for multi-label classifiers," *Pattern Recognition*, vol. 46, no. 7, pp. 2055–2065, 2013. DOI: 10.1016/j.patcog.2013.01.012.

[11] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ACM, 2014, pp. 92–101.

[12] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[13] M.-L. Zhang and Z.-H. Zhou, "A review on multi-label learning algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 8, pp. 1819–1837, 2014. DOI: 10.1109/TKDE.2013.39.

[14] C. Hauff and G. Gousios, "Matching GitHub developer profiles to job advertisements," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, IEEE, 2015, pp. 362–366. DOI: 10.1109/MSR.2015.41.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2015.

[16] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets," *PLOS ONE*, vol. 10, no. 3, e0118432, 2015. DOI: 10.1371/journal.pone.0118432.

[17] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785.

[18] V. Cosentino, J. Luis, and J. Cabot, "Findings from GitHub: Methods, datasets and limitations," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ACM, 2016, pp. 137–141. DOI: 10.1145/2901739.2901776.

[19] G. J. Greene and B. Fischer, "CVExplorer: Identifying candidate developers by mining and exploring their open source contributions," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2016, pp. 804–809. DOI: 10.1145/2970276.2970285.

[20] C. V. F. Monteiro, F. Q. B. da Silva, and L. F. Capretz, "The innovative behaviour of software engineers: Findings from a pilot case study," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '16)*, New York, NY, USA: Association for Computing Machinery, 2016, pp. 7–16. DOI: 10.1145/2961111.2962589. [Online]. Available: https://doi.org/10.1145/2961111.2962589.

[21] R. K.-W. Lee and D. Lo, "GitHub and Stack Overflow: Analyzing developer interests across multiple social collaborative platforms," *arXiv preprint arXiv:1708.06860*, 2017.

[22] Y. Liu, P. He, G. Wu, and Y. Li, "Towards understanding developers' collaborative behavior in open source software ecosystems," *Journal of Software*, vol. 12, no. 6, pp. 393–405, 2017. DOI: 10.17706/jsw.12.6.393-405. [Online]. Available: https://doi.org/10.17706/jsw.12.6.393-405.

[23] E. Weimar, A. Nugroho, J. Visser, A. Plaat, M. Goudbeek, and A. P. Schouten, *The influence of teamwork quality on software team performance*, arXiv preprint arXiv:1701.06146, 2017. [Online]. Available: https://arxiv.org/abs/1701.06146.

[24] Y. Xiong, Z. Meng, B. Shen, and W. Yin, "Mining developer behavior across github and stack overflow," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, Knowledge Systems Institute, 2017, pp. 578–583. DOI: 10.18293/SEKE2017-062.

[25] Y. Kong and M. Li, "Proactive personality and innovative behavior: The mediating roles of job-related affect and work engagement," *Social Behavior and Personality: An International Journal*, vol. 46, no. 3, pp. 431–446, 2018. DOI: 10.2224/sbp.6618.

[26] P. L. Curşeu, R. Ilies, D. Vîrgă, L. Maricuţoiu, and F. A. Sava, "Personality characteristics that are valued in teams: Not always "more is better"?" *International Journal of Psychology*, vol. 54, no. 5, pp. 638–649, 2019. DOI: 10.1002/ijop.12511. [Online]. Available: https://doi.org/10.1002/ijop.12511.

[27] Z. Wang, Y. Feng, Y. Wang, J. A. Jones, and D. Redmiles, "Unveiling elite developers' activities in open source projects," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 1–35, 2019. DOI: 10.1145/1122445.1122456.

[28] J. E. Montandon, C. Politowski, L. L. Silva, M. T. Valente, F. Petrillo, and Y.-G. Guéhéneuc, *What skills do IT companies look for in new developers? A study with Stack Overflow Jobs*, arXiv preprint arXiv:2011.02473, 2020. DOI: 10.48550/arXiv.2011.02473.

[29] E. Dias, P. Meirelles, F. Castor, I. Steinmacher, I. Wiese, and G. Pinto, "What makes a great maintainer of open source projects?" In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*, Madrid, Spain: IEEE Press, 2021, pp. 982–994. DOI: 10.1109/ICSE43902.2021.00093. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00093.

[30] E. J. Hu et al., "LoRA: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[31] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How do software developers use GitHub Actions to automate their workflows?" In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories*, IEEE/ACM, 2021, pp. 420–431. DOI: 10.1109/MSR52588.2021.00054.

[32] P. Ardimento, M. L. Bernardi, M. Cimitile, D. Redavid, and S. Ferilli, "Understanding coding behavior: An incremental process mining approach," *Electronics*, vol. 11, no. 3, p. 389, 2022. DOI: 10.3390/electronics11030389.

[33] N. E. Gold and J. Krinke, "Ethics in the mining of software repositories," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–49, 2022. DOI: 10.1007/s10664-021-10057-7.

[34] J. T. Liang, T. Zimmermann, and D. Ford, "Towards mining oss skills from github activity," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, Association for Computing Machinery, 2022, pp. 1–5. DOI: 10.48550/arXiv.2203.02027. [Online]. Available: https://arxiv.org/abs/2203.02027.

[35] D. Pina, A. Goldman, and C. Seaman, "Sonarlizer xplorer: A tool to mine github projects and identify technical debt items using sonarqube," in *2022 IEEE/ACM International Conference on Technical Debt*, IEEE, 2022, pp. 71–75. DOI: 10.1145/3524843.3528098.

[36] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *Information Fusion*, vol. 81, pp. 84–90, 2022. DOI: 10.1016/j.inffus.2021.11.011.

[37] D. Strode, T. Dingsøyr, and Y. Lindsjørn, "A teamwork effectiveness model for agile software development," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–50, 2022. DOI: 10.1007/s10664-021-10115-0. [Online]. Available: https://doi.org/10.1007/s10664-021-10115-0.

[38] C. Zimmerle, K. Gama, F. Castor, and J. M. Mota Filho, "Mining the usage of reactive programming apis: A study on github and stack overflow," in *Proceedings of the 19th International Conference on Mining Software Repositories*, Association for Computing Machinery, 2022, pp. 1–12. DOI: 10.1145/3524842.3527966.

[39] A. Conti, V. Gupta, J. Guzman, and M. Roche, "Incentivizing innovation in open source: Evidence from the GitHub sponsors program," National Bureau of Economic Research, Working Paper 31668, 2023. DOI: 10.3386/w31668.

[40] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," *arXiv preprint arXiv:2305.14314*, 2023.

[41] A. Q. Jiang et al., *Mistral 7b*, arXiv preprint arXiv:2310.06825, 2023. [Online]. Available: https://arxiv.org/abs/2310.06825.

[42] A. Ramaswami. "Open source maintainers: What they need and how to support them," The Linux Foundation. [Online]. Available: https://www.linuxfoundation.org/blog/open-source-maintainers-what-they-need-and-how-to-support-them.

[43] I. Salehin and D.-K. Kang, "A review on dropout regularization approaches for deep neural networks within the scholarly domain," *Electronics*, vol. 12, no. 14, p. 3106, 2023. DOI: 10.3390/electronics12143106.

[44] GitHub, Inc. "Github general privacy statement. "[Online]. Available: https://docs.github.com/en/site-policy/privacy-policies/github-general-privacy-statement.

[45] Open Source Security Foundation. "Maintainer motivations, challenges, and best practices on open source software security," OpenSSF. [Online]. Available: https://openssf.org/blog/2024/01/31/maintainer-motivations-challenges-and-best-practices-on-open-source-software-security/.

[46] U. Saha, J. D'Andria, and T. Menezes, *The open source resume: How open source contributions help students demonstrate alignment with employer needs*, arXiv preprint arXiv:2510.25180, 2025.