Social Intelligence Approach for Service-Oriented Software

by

Faisal A. Binzagr

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer and Information Science) in The University of Michigan-Dearborn 2022

Doctoral Committee:

Professor Brahim Medjahed, Chair
Assistant Professor Mohamed Abouelenien
Assistant Professor Abdallah Chehade
Professor Qiang Zhu
Acknowledgements

First of all, I would like to acknowledge and give my warmest thanks to my advisor Professor Brahim Medjahed for his support and guidance from the moment I met him. This work could not have been possible without his expertise. He never hesitated to help and advise me to complete my thesis. I learned a lot from his great research experience during my journey.

I would also like to thank my parents and my siblings for their wise counsel and unwavering support. Also, a deepest gratitude owed to my wife. I will never forget her endless support and patience that cannot be underestimated. Thank you to my friends for supporting me as I worked through this. I am extremely thankful to my lab mates Hamza Labbaci and Khaled Alahmadi for providing me suggestions and inspirations for my thesis.

My heartfelt thanks also go to Dr. Qiang Zhu, Dr. Mohamed Abouelenien, and Dr. Abdallah Chehade, for their insightful comments and encouragement, but also for the interesting questions which incented me to widen my research from various perspectives.

Last but not least I would like to thank all the people who made this dissertation possible. Completion of this research has been a humbling experience and I will never forget the encouragement and enthusiasm from you all.

Faisal Binzagr
# Table of Contents

Acknowledgements ........................................................................................................ ii

Lists of Tables ....................................................................................................................... vi

List of Figures ........................................................................................................................ vii

Abstract ................................................................................................................................. ix

Chapter 1 Introduction .......................................................................................................... 11

1.1 Background and Motivation ......................................................................................... 11

1.2 Thesis Statement ........................................................................................................... 14

1.3 Research Issues ............................................................................................................. 16

1.4 Research Contributions ............................................................................................... 18
  1.4.1 Contribution 1: CrowdMashup: Recommending Crowdsourcing Teams for Mashup Development ............................................................................................................. 19
  1.4.2 Contribution 2: FAME: An Influencer Model for Service-Oriented Environments ................................................................................................................................. 20
  1.4.3 Contribution 3: TEAM: Leveraging Experiences from Developer Communities for API Recommendation .......................................................................................................... 21

1.5 Related Work .................................................................................................................. 22
  1.5.1 Crowdsourcing and Social Computing .................................................................... 23
  1.5.2 Influencer Models .................................................................................................... 25
  1.5.3 Quality of Service .................................................................................................. 27
  1.5.4 Service Recommendation ...................................................................................... 29

1.6 Dissertation Organization .............................................................................................. 30

Chapter 2 CrowdMashup: Recommending Crowdsourcing Teams for Mashup Development ... 31

2.1 Introduction .................................................................................................................... 31
2.2 Scenario .......................................................... 34
  2.2.1 Fixed Team Size ........................................... 34
  2.2.2 Flexible Team size ....................................... 35

2.3 CrowdMashup Approach ....................................... 37
  2.3.1 Social Analysis of Developers Community (SADC) .... 38
  2.3.2 Query Model ............................................. 43
  2.3.3 Team Finder (TF) ......................................... 46
    2.3.3.1 Graph-Based Techniques ............................ 47
    2.3.3.2 Cluster-Based Technique ............................ 50
    2.3.3.3 Search-Based Technique ............................ 52

2.4 Implementation and Performance ............................ 54
  2.4.1 CrowdMashup Prototype ................................. 54

2.5 Experiment .......................................................... 56

2.6 Conclusion .......................................................... 60

Chapter 3 FAME: An Influencer Model for Service-Oriented Environments .......................... 61

3.1 Introduction .......................................................... 61

3.2 The FAME Approach: An Overview .......................... 65
  3.2.1 Motivation ................................................ 65
  3.2.2 Architecture .............................................. 67

3.3 Influencer Identification and Prediction in FAME ........... 69
  3.3.1 Unstructured Data Extractor (UDE) ....................... 69
  3.3.2 Structured Data Extractor (SDE) ......................... 70
  3.3.3 API Influencer Score (AIS) ............................. 73
  3.3.4 Influence Score Prediction ............................... 74

3.4 Experiments .......................................................... 77

3.5 Conclusion .......................................................... 78

Chapter 4 TEAM: Leveraging Experiences from Developer Communities for API

Recommendation .......................................................... 80

4.1 Introduction .......................................................... 80

4.2 Motivation .......................................................... 83

4.3 TEAM Architecture ............................................... 84
4.3.1 Data Extraction Categorization (DEC) ............................................................................. 86
4.3.2 Features Engineering (FE) .............................................................................................. 89
4.4 QoE Dimensions ................................................................................................................. 90
4.5 TEAM QoE Model .............................................................................................................. 99
4.6 Implementation and Performance ....................................................................................... 103
  4.6.1 TEAM Tool ..................................................................................................................... 103
  4.6.2 Experiments .................................................................................................................. 105
4.7 Conclusion .......................................................................................................................... 111

Chapter 5 Conclusion and Future work .................................................................................... 112
  5.1 Conclusion ......................................................................................................................... 112
  5.2 Future Work ...................................................................................................................... 113
Bibliography ............................................................................................................................. 114
Lists of Tables

Table 2.1 Performance Evaluation of Recommender Systems.............................................. 40
Table 3.1 Motivating Scenarios.......................................................................................... 66
Table 3.2 API Features in Developer Communities.................................................................. 72
Table 3.3 Prediction Models.................................................................................................. 77
Table 4.1 Google Maps & Bing Maps Analysis ...................................................................... 83
Table 4.2 Descriptive Statistics of API Usage Metric.............................................................. 88
Table 4.3 Class Label Setting of API Usage Metric................................................................. 89
Table 4.4 QoE Features in Developer Communities.............................................................. 91
Table 4.5 Developer Features ............................................................................................... 93
Table 4.6 Quantitative Performance Metrics for of API Usage Metrics Classification .......... 108
List of Figures

Figure 2.1 CrowdMashup for Competition and Tournament .......................................................... 34
Figure 2.2 CrowdMashup for Business and Enterprise ................................................................. 36
Figure 2.3 CrowdMashup Architecture ......................................................................................... 37
Figure 2.4 Interests Table .............................................................................................................. 39
Figure 2.5 Sociometric Graph ........................................................................................................ 41
Figure 2.6 Sociometric Plus Graph Process .................................................................................. 42
Figure 2.7 Query Model Structure ............................................................................................... 44
Figure 2.8 Crowdsourcing Team Generation ................................................................................ 48
Figure 2.9 Clustering Process ....................................................................................................... 51
Figure 2.10 Cluster Based Algorithm ............................................................................................ 51
Figure 2.11 Genetic Algorithm Process ......................................................................................... 52
Figure 2.12 Solution Representation ............................................................................................. 53
Figure 2.13 Example of Team Sub-Graph ...................................................................................... 53
Figure 2.14 The CrowdMashup User Interface ............................................................................. 55
Figure 2.15 Single Query Team Performance for Non-CTG (Random, Skills Only, Sociometric
Only) and CTG (CTG-Skills, CTG-Sociometric) ................................................................. 56
Figure 2.16 Multiple Queries Team Performance for Different Team Sizes ............................... 57
Figure 2.17 Best Teams Performance ............................................................................................ 59
Figure 2.18 Time Analysis ............................................................................................................ 60
Figure 3.1 FAME Architecture .................................................................................................... 68
Figure 3.2 Influencer Tiers ................................................................. 74
Figure 3.3 Impact of API Features on the AIS Score .................................. 75
Figure 3.4 Recall and Precision .................................................................. 78
Figure 4.1 QoE & QoS Overview ................................................................. 82
Figure 4.2 QoE Architecture ...................................................................... 85
Figure 4.3 Distribution of API Usage Metrics .............................................. 88
Figure 4.4 StackOverflow and GitHub Users K-Mean Clustering .................. 93
Figure 4.5 Security-Related Posts Processing .............................................. 97
Figure 4.6 Structure of Random Forest classification model ....................... 100
Figure 4.7 Structure of SVM .................................................................... 101
Figure 4.8 Structure of Neural Networks classification model ..................... 102
Figure 4.9 The APIs QoE visualization page ............................................... 104
Figure 4.10 The APIs QoE prediction page .................................................. 105
Figure 4.11 Models Performance ............................................................... 109
Figure 4.12 Confusion Matrices ................................................................. 110
Figure 4.13 Variable Importance ................................................................ 110
Abstract

Service-oriented computing allows companies to break down functionalities into individual, autonomous services. The last decade has seen a surge of services in the form of APIs in a variety of domains. The API economy is growing rapidly, and companies are making APIs an integral part of their software development strategies. The reuse of existing APIs allows developers to build large-scale applications referred to as mashups or composite services. Service Composition or Mashup is a technique to construct value-added services by integrating pre-existing APIs. Recent technologies such as cloud computing, big data, and Internet of Things (IoT) boosted the popularity of service composition and mashups. This is because of the ability of services to interact and work together despite their heterogeneity and autonomy.

Integrating multiple APIs created by various third parties require a wide array of technical skills such as Web, data management, software engineering, programming, and security. To overcome these challenges, developers often turn to programming communities or crowd (e.g., StackOverflow, GitHub) to share practices, knowledge, experience, and brainpower in solving intricate problems. This growing trend adds a "social" dimension to software development allowing programmers to share experiences on a daily basis through blogs, wikis, tutorials, bug reporting, and discussion forums. As a result, the data related to APIs (both technical and non-technical) are scattered in different platforms and with various formats. Our vision in this research is to leverage artificial intelligence techniques (e.g., machine learning, natural language processing) in order to turn API-related "social" data into useful information to support the creation
of future mashups and service compositions. We use the term social intelligence (combination of social computing and artificial intelligence) to refer to the proposed approach.

In this dissertation, we propose a social intelligence-based approach for service composition. First, we introduce CrowdMashup, a crowdsourcing approach for mashup team recommendation. We analyze online developer communities and API directories to infer developers' interests in APIs through natural language processing. We propose three techniques to generate teams of developers that best fulfill mashup requirements: graph-based, clustering-based, and search-based. Second, we propose FAME (inFluencer Apis in developer coMmunitiEs), a multi-dimensional influencer model for APIs in service-oriented environments. The proposed model helps API providers to increase the visibility of their APIs and API consumers to select the best-in-class APIs. We introduce a linear regression technique to predict the evolution of influence scores and correlate API features to those scores. Third, we propose TEAM (qualiTy of Experience-based Api recoMmendation), an approach that leverages prior API development experiences to recommend APIs. We extract structured and unstructured information from various developer communities to build a Quality of Experience (QoE) model. We train three different classifiers to recommend APIs and predict their usage: random forest, support vector machine, and deep neural net-work. Finally, we conduct extensive experiments on real-world and large datasets extracted from developer communities to evaluate the proposed approaches.
CHAPTER 1

Introduction

1.1 Background and Motivation

Service-oriented computing allows companies to break down capabilities and business functionalities into individual, autonomous services [1]. The last decade has seen a surge of services in the form of Web APIs (simply APIs) in a variety of domains [2]. The API economy is growing rapidly and companies are making APIs an integral part of their software development strategies. For instance, the ProgrammableWeb¹ directory lists more than 24,339 APIs as of (January 2022). APIs enable developers to access hardware and software resources via the Internet and Web-specific protocols. According to the W3C (World Wide Web Consortium) ”A web service is a software system designed to support interoperable machine-to-machine interaction over a network”. These software systems are self-contained, distributed, interactive applications that can be invoked or published over the internet. Communication among Services/APIs is based on the principles of SOAP (Simple Object Access Protocol), REST (Representational State Transfer), or RSS/Atom feeds [3].

The reuse of existing APIs allows developers to build large-scale applications referred to as Mashup or Composite services. Service Composition or Mashup² is a technique to combine many Web applications by consuming pre-existing APIs (or services) to construct value-added services [1]. It enables developers to build high-quality service-oriented applications with less effort. The

¹ https://programmableweb.com/
² The terms service composition and mashup are used interchangeably in this document.
vast number of public APIs leads to a growing of mashups. For instance, the Yahoo Weather API has been used in more than one thousand repositories in GitHub³. Besides, recent technologies such as cloud computing, big data, and Internet of Things (IoT) boosted the popularity of service composition and mashups. This is because of the ability of services to interact and work together despite their heterogeneity and autonomy.

The substantial number of available APIs creates many challenges to the area of developing mashups. First, there are so many APIs publicly available that provide similar functionalities. For instance, there are around one thousand APIs available in ProgrammableWeb as mapping services. Picking the best API to use is not a straightforward task. Second, the large amount of information available about APIs is scattered across multiple independent platforms and cannot be accurately obtained from one single source. Besides, those platforms return API-related data in heterogeneity formats. For example, posts in StackOverflow⁴ and commit comments in GitHub are textual format, and bug reports in Bugzilla⁵ are presented in proprietary format. Combining and analyzing that information is a tedious process to extract novel insights to service selection, recommendation, and composition. Ultimately, mashup development generally involves several APIs requiring a variety of technological skills such as REST, SOAP, JSON, XML, and security. This often calls for the collaboration of multiple developers to reduce the overall mashup cost (e.g., development time).

The difficult process now is how to recommend developers to be part of mashup development teams. Integrating multiple APIs created by diverse third parties require a wide array of technical skills such as Web (e.g., REST), data management (e.g., JSON), programming (e.g., SDKs), and

---
³ https://github.com
⁴ https://stackoverflow.com
⁵ https://bugzilla.mozilla.org
security (e.g., authentication). To overcome these challenges, developers often turn to programming communities or crowd (e.g., StackOverflow, GitHub, and Topcoder) to share practices, knowledge, experience, and brainpower in solving intricate problems. Crowdsourcing is a powerful sourcing model to perform a broad range of hard tasks by splitting the work between workers [4]. It has been used in software development to perform vital activities such as implementation, design, coding, or testing [5]. Besides, the software industry has recently seen a new trend where crowdsourcing companies such as Topcoder sell services to corporate, mid-size, and small-business clients, and pay community members (i.e., developers) for their work. Moreover, the growth of open source development accelerates mashup development by involving social computing techniques. Social computing reflects interactions that people experience on a daily basis, such as blogs, email, wikis, social bookmarking, across artificial systems known as social networks.

The data related to APIs are scattered in different platforms and with various formats. The main source of this data is the developer communities, this includes technical and non-technical information. For instance, tutorials, source code, reporting bugs, feedback, and opinions about APIs in various ways such as participating in discussion forums. Collecting and analyzing this historical information and then turn it into useful information by using an intelligent approach is called social intelligence. This is a tedious task, for that reason, we leverage artificial intelligence approaches to turn API-related data and social information into useful information. As an example of using artificial intelligence, we applied a natural language processing technique (sentiment analysis) to turn developers’ textual feedback from discussion forums into a polarity score such as (positive, neutral, and negative) which can be used to infer the developer’s interests in APIs. Leveraging the past experience of developers from communities and crowds will help future
mashup development by learning from their historical data such as (success, failure, bugs, and est.).

1.2 Thesis Statement

Service recommendation for building mashup has lately taken a fundamental stage as an emerging research area. Several techniques have been proposed for mashup recommendations [6] and [7]. However, these techniques and methods ignore feedback from community developers in their recommendations. For that reason, we envision research at the intersection of artificial intelligence, service-oriented computing (API and Mashup), and social computing (developer communities). These fields are fundamental to the study of how to facilitate mashup development, particularly driven by new sources of data, for example, developer communities and social media to learn from the past development experiences. Therefore, we propose several approaches to provide developers with a big picture of how to find the best service in the service-oriented environment. The scope of this thesis is at the intersection of three major research areas (Figure 1.1): social computing (i.e., leverage developer data from communities), artificial intelligence (i.e., machine learning and natural language processing), and service computing (i.e., APIs, service composition, and mashup). Programmers often turn to developers’ communities (e.g., GitHub and StackOverflow) to connect, collaborate and brainpower in solving intricate problems. Since those platforms have a large number of developers (by millions), massive data are generated. To understand this information, we need to investigate fundamental social computing principles such as the relation between developers. Moreover, the past decade has witnessed a growing interest in API and Mashup development [8]. For instance, the popular ProgrammableWeb API directory currently includes more than 24,339 APIs and 7,989 mashups as of (January 2022). Therefore,
data related to APIs are scattered in different platforms with different formats. Hence, there is a need to leverage artificial intelligence techniques to learn and study the past software development experiences from developer communities. To achieve this goal, we need to evaluate and analyze the data from those communities and API & Mashup repositories. For that, we applied artificial intelligence techniques such as collaborative filtering and natural language processing to assist developers to build their mashup.

Figure 1.1 Thesis Statement

Our research vision is inspired by earlier work done in psychology which aims to combine the social aspect and intelligence to see how social relationships can have an impact on intelligence. The concept of social intelligence (SI) was first introduced by Thorndike [9] as one
that” encompasses our abilities to interpret others’ behavior in terms of mental states (thoughts, intentions, desires, and beliefs), to interact both in complex social groups and in close relationships, to empathize with others’ states of mind, and to predict how others will feel, think and behave” [10]. In this thesis, we borrow this concept and use it in the area of API and mashup development. We look at social interactions among developers in the communities as well as their past development experiences (success or failure). Then we analyze those relationships and experiences using artificial intelligence techniques to learn how to build better mashups and service compositions in the future.

1.3 Research Issues

We identify the following challenges addressed in our research to recommend APIs for service composition and mashups:

- *Identifying teams for mashup development:* Mashup creation typically requires several APIs that include a range of integrating technology skills such as REST, SOAP, JSON, XML, and security. It also needs many developers to collaborate together to reduce the total cost of mashup (e.g., development time). What makes the problem more challenging is the substantial number of available APIs and programmers. For example, ProgrammableWeb publishes more than 24,339 APIs, StackOverflow, and GitHub developer community platforms report an estimated 16 and 56 millions subscribers respectively as of (January 2022). However, there are two important factors for finding and forming skilled mashup developers. First, [11] studies have shown that good social relationships improve team success among members. Hence, the problem is how to measure developer’s social relationships. Second, measuring developer’s skills-
programming experience and working reputation is not straightforward. To the best of our knowledge, this is the first work to resolve recommendations in mashups from the developer’s perspective.

- *Understanding API influence:* Service-oriented software suffers from a lack of visibility due to the huge number of available APIs. This issue will affect both API providers and consumers. API providers are the developers who created the APIs available in the community. They publish important information about their APIs such as tutorials, articles, SDKs, libraries, new releases, and source code. API consumers use existing APIs to build mashups and service-oriented applications. They share experiences, feedback, and opinions about APIs in various ways such as participating in discussion forums, reporting bugs, and following APIs. Therefore, API consumers may have different views on what makes an API relevant. Some consumers may value APIs with the least number of reported bugs. Others may consider the opinions expressed by peers toward the API as significant. However, API providers may not able to reach out larger audience of developers. What is needed is an influencer model for APIs in service-oriented environments similar to the concept of influences commonly used in social media. Such model helps API providers to increase the visibility of their APIs and API consumers to select the best-in-class APIs.

- *Leveraging Quality of Experience of APIs from developer’s communities:* The performance of the involved services determines the overall performance of the composed or mashup service. Hence, it is crucial to pick a high-quality service to form mashup service. A large body of research proposed quality of service (QoS) as a metric for composition selection [12]. However, very few contributions looked at the quality of experience (QoE) as a metric for selection. Understanding and measuring developers’ QoE
is a vital and challenging task, especially with the substantial number of available APIs that provide similar functionality, selecting the best service to use is not straightforward. Furthermore, obtaining the quality from the developers’ communities illustrates the real difficulty and experience that can evaluate the service accurately.

1.4 Research Contributions

Developers usually use a web search engine such as Google to find resources online to improve productivity. To be specific, in service-oriented software, developers often need knowledge beyond what they have already known such as how to use specific protocol or service documentation description. [13], [14]. A better understanding of what developers are searching for in the internet would help us to understand their behaviors and the problems that they face. Especially, when there is a lack of APIs documentation, developers share their experience in different ways such as using Q/A platform, share code fragments/ repositories and tutorials. The issue is how to decide which API to use and how to use those APIs and which programmers most suitable to work together to build your software. We propose novel techniques to assist developers in building mashups. Our techniques are based on the following two practices in software development. First, past software development experiences from these communities to assist programmers and software companies in building new software. Second, software development often requires knowledge beyond what developers already knew [13]. For instance, quality of experience information is usually not mentioned in software/APIs documentation. Figure 1.2 summarizes our current research contributions.
1.4.1 Contribution 1: CrowdMashup: Recommending Crowdsourcing Teams for Mashup Development

Mashups involve the collaboration of multiple developers to build Web applications out of pre-existing APIs. A large body of research focused on recommending APIs for mashups. However, very few contributions looked at recommending developers. We propose CrowdMashup [15], a crowdsourcing approach for mashup teams recommendation. We analyze online developer communities and API directories to infer developers’ interests in APIs through natural language processing. We predict missing interest values using the alternating least square method for collaborative filtering. We also model interactions (comments and replies) among developers as a weighted undirected graph and introduce a sociometric to identify socially related developers. We propose a graph-based algorithm, based on the concept of cliques in graph theory, that combines...
developers’ skills and sociometric to recommend efficient and balanced teams. Moreover, we extend CrowdMashup with a more comprehensive evaluation by developing three techniques (graph-based, cluster-based, and search-based) [16] to generate teams that best fulfill mashup requirements. We conduct experiments on real-world data and APIs from StackOverflow and ProgrammableWeb to evaluate our approach. Experiments show promising results in generating efficient teams using graph-based algorithms for mashup development.

1.4.2 Contribution 2: FAME: An Influencer Model for Service-Oriented Environments

Service-oriented software suffers from a lack of visibility due to the huge number of available APIs. This issue will affect both API providers and consumers. API providers are the developers who created the APIs available in the community. They publish important information about their APIs such as tutorials, articles, SDKs, libraries, new releases, and source code. API consumers use existing APIs to build mashups and service-oriented applications. They share experiences, feedback, and opinions about APIs in various ways such as participating in discussion forums, reporting bugs, and following APIs. Therefore, API consumers may have different views on what makes an API relevant. Some consumers may value APIs with the least number of reported bugs. Others may consider the opinions expressed by peers toward the API as significant. However, API providers may not be able to reach out larger audience of developers. To that end, we propose FAME (inFluencer Apis in developer coMmunitiEs) [17], a multi-dimensional influencer model for APIs in service-oriented environments. The proposed model helps API providers to increase the visibility of their APIs and API consumers to select the best-in-class APIs. We introduce a cumulative API influence score to measure the influence of APIs across communities and categorize APIs into tiers based on their influence. We introduce a linear regression technique to
predict the evolution of influence scores and correlate API features to those scores. We conduct experiments on large and real-world datasets from programming communities to illustrate the viability of our approach.

1.4.3 Contribution 3: TEAM: Leveraging Experiences from Developer Communities for API Recommendation

Service-oriented software systems require the integration of various APIs provided by third parties and accessible via Web technologies. However, the sheer number and wide variety of available APIs makes it challenging for developers to identify the right APIs to use in their projects. For instance, the popular ProgrammableWeb API directory currently lists more than one thousand payment APIs (as of October 2021). Existing techniques for API recommendation mostly rely on user-centric QoS (Quality of Service) metrics such as cost, availability, and response time. While QoS may be suitable for API consumers and end-users, they may not be as helpful for developers. Developers are generally interested in synthesizing past experiences from their peers to identify the APIs that are most suitable for their software project and development needs. To that end, we propose TEAM (quality of Experience-based API recommendation) [18], an approach and tool that leverages prior API development experiences to recommend APIs. We extract structured and unstructured information from various developer communities such as GitHub, StackOverflow, ProgrammableWeb, HackerNews, and YouTube to build a Quality of Experience (QoE) model. We train three different classifiers (random forest (RF), support vector machine (SVM), and neural network (NN)), to recommend APIs and predict their usage. We conduct extensive experiments on real-world and large datasets extracted from developer communities to evaluate the proposed approach.
1.5 Related Work

We investigate the state of the arts related to our contributions to understand the research problem that has been studied and the proposed solutions. Figure 1.3 illustrates the related work taxonomy to categorize our contributions and to map them with existing work. Therefore, we classify our contributions under two main areas API selection and developers’ recommendation. We borrow the classification of Web service selection methodologies [19] and we mapped the FAME contribution under the content-based and TEAM contribution under user-based. However, regarding the developers’ recommendation area. We classified the state of arts related to developers’ recommendations into three categories: communities-based, economic- factor, and algorithm complexity. We consider CrowdMashup contribution under the communities-based area. We will describe each contribution related work as follow:
1.5.1 Crowdsourcing and Social Computing

The growth and popularity of crowdsourcing has led to significant research on forming teams to facilitate collaborative software development [20]. Part of this research has focused on team structure, while other contributions focused on the complexity of the algorithm and economic factors for team building. [21] shows that network structure between members has a vital effect on team formation. It uses four different network structures to model team formation and compares the performance of each structure. [22] takes advantage of social network information and uses hierarchical structures (e.g., using “report to”) between team members. [23] defines a self-organized team formation technique by allowing members to rate each other and use other information such as demographics (e.g., age, gender). [24] proposes a framework that recommends teams based on the skills and connection among members. It uses co-authorship in DBLP and clustering algorithms to find expert teams (sub-graph). [4] employs a dynamic programming technique in crowdsourcing based on the prior familiarity of members to generate target teams. It considers the availability (response time) of the members to find the most familiar alternative members. [25] defines heuristic algorithms based on notions such as weak and strong ties in social networks. It utilizes two metrics to find social connection from an undirected weighted graph. [26] investigates software development team productivity based on an anonymous survey at a large software company. They built a regression model to identify important team culture factors for modeling team productivity. The model found that developers report their team’s productivity based on communication and satisfaction with each other. [27] explores the network structure on crowdsourcing to determine how open-source project performance varies as the share of crowdsourced requirements increases. They used six types of measurements to test the effectiveness such as requirement close-out time, requirement response time. However, they
offered regression models to analyze the impact of network structure and requirements crowdsourcing on the performance. [28] aims to improve the developer team formation experience by interviewing 32 developers working on Microsoft and joining new teams. They focus on three factors in the team structure such as learning, confidence building, and socialization.

Several techniques dealt with the issue of improving the efficiency of the team formation process. [29] proposes a genetic algorithm with the goal of finding the best groups that can meet the defined tasks based on members availability, skills, and price. [30] introduces an approach for forming teams with specific skills from a vast professional community using network communication costs to optimize team formation. It calculates communication costs by using minimum spanning tree and the largest shortest path from the graph. [31] describes a greedy approach for better performance considering team size and workload such as the number of tasks allocated to each member.

[32] and [33] propose a team formation technique based on pricing to find cost effective teams. [34] studies task coordination cost in crowdsourcing teams. It aims to facilitate self-coordination and communication among teams by distributing and synchronizing the project tasks. [11] introduces a technique for forming multiple teams to maximize the global efficiency of the teams considering skills, availability, sociometric (relationship), and allowed time (part-full time) members. [35] proposes a negotiation-based team formation technique where the deal to join the team is used as a formation factor. [36] investigates how personality affects team performance by applying the DISC (dominance, inducement, submission, compliance) personality test. [37] discusses team elasticity in software development such as the skills, experiences, response time, and reliability of the workers. [38] proposes a data leak-aware system in crowdsourcing team by applying clustering algorithms that detect social interactions between members to avoid data
leakage. [39] conducts statistical analysis to investigate how to extract influence factors from successful teams.

CrowdMashup differs from existing approaches in multiple ways. First, to the best of our knowledge, our contribution is the first to look at team recommendation related to mashups. Second, we define a two-level approach to analyze developers’ communities. At the individual developer’s level, we infer developer’s interests in APIs through natural language processing and collaborative filtering. At the community level, we consider social relationships among developers as an important factor to recommend team members. We model interactions among developers as a weighted undirected graph and find cliques to identify strongly related developers. Note that our approach is different from the one introduced in [25] where members of the same team are selected from different cliques to ensure the impartiality of the execution result of a task. We use cliques to recommend teams composed of (socially) strongly connected members to improve productivity.

1.5.2 Influencer Models

The identification of influential nodes in distributed environments such as social networks and forums has been the subject of many research efforts [40][41][42][43]. Few research proposals [44][45] study influencers in software development. However, existing research considers developers as influencers not APIs. [42] proposes a methodology to identify influencer nodes that are likely to affect other nodes in social networks. It computes the centrality degree of nodes and analyzes node activities. [42] focuses on the position of nodes in the network. Our approach instead leverages both structured (e.g., number of mashups, number of articles) and unstructured (e.g., user feedback) across multiple developer platforms to identify influencers. We also show that influencer identification precision gets better as we leverage a larger number of features. [43]
presents a study for finding influential authors on Twitter forums. It combines both user profile information and user interaction features with decision tree to identify influencer authors. [46] introduces a study for finding developers API-related opinions. It surveys software developers and finds that developers prefer to see opinions about the following API aspects in developer forums, such as StackOverflow. They surveyed 178 software developers. They found that developers need a tool to efficiently analyze the opinions, such as an opinion summarizer or API sentiment trend analyzer.

Our approach identifies APIs as influencers not users opinions. Moreover, we use a multi-objective function that combines multiple attributes collected from various sources. [41] proposes a study to understand influencers who lead development and dictate how projects evolve. It shows that analyzing influencer behaviors allows understanding the evolution of software ecosystem and even predict future evolution. The main focus of our approach is to identify influencer APIs and the attributes that contribute to their emergence, rather than assuming the existence of those influencers and studying their behavior. [47] shows that influence score depends on engagement, sentiment, and growth. [40] shows that originality and uniqueness of user content are crucial factors to identify influencers in Instagram. [47] and [40] rely mainly on social metrics to determine influencers. Our approach extends the analysis to encompass attributes from various sources. Besides, it considers APIs as influencers in programming platforms instead of users in social networks. [48] computes influence score for users across several social networks. It evaluates the quantity and quality of reactions a user action prompted to assess the extent to which the user is influential. [44] and [45] identify the most influential developers, repositories, technologies, and programming languages in GitHub. [44] shows that the analysis of social networks, particularly the relations among developers, developers and repositories, and developers
and followers help identify developers’ influencer index. [45] proposes an approach to measure user influence in GitHub. It analyzes relationships between users, as well as between users and projects. In contrast to our approach, [45] and [44] are restricted to GitHub and StackOverflow data. In our approach, we show that using a multi-objective function that combines both structured and unstructured features from diverse platforms substantially enhances the precision of the influencer identification process. We also introduce models to predict the evolution of influencer scores for newly developers and existing APIs.

1.5.3 Quality of Service

The problem of evaluating the quality of Web service for optimal selection and composition has been the subject of many research efforts. Part of this research has focused on objective quality criteria, while other contributions focused on the subjective quality criteria and others combined both of them.

Objective Quality Criteria: [12] proposes WSRec which predicats the QoS properties by using historical QoS data from similar users and services. It employs Pearson Correlation Coefficient (PCC) collaborative filtering technique. They used real world services to evaluate their approach. [49] employs a QoS-aware Web service selection algorithm based on the clustering of QoS properties. The proposed algorithm is able to decrease the execution time to find the optimal service. [50] introduces WSExpress which is a web service search engine. This engine ranked the result based on the QoS and the functionality description of the service for example if the user search for CarPrice service the result could be AutomobileInformation or VehicleRecommend. It utilizes the experiment on real-world Web services. This approach misses the QoE for optimize recommendation result.
Subjective Quality Criteria: [51] applies an automated approach to aggregate the QoE attributes and investigates the relationship between QoE and QoS. This study use user perception about web service based on 58 different platforms. It gains the overall end-user view about services. In contrast, our study investigate the QoE attributes based on developers’ views and attitudes not for all regular users. [52] leverages QoE by applied Mean Opinion Score (MOS) test and correlate QoS parameters to subjective QoE. Then it estimates the QoE based on a fuzzy-rough hybrid model for web service selection. [53] proposes QoE-aware cloud service ranking framework based on Markov chain model. They collect QoE from user rating to optimize the ranking result. [54] defines the Web-QoE as the “Quality of experience of interactive services that are based on the HTTP protocol and accessed via a browser”. It captures the MOS rating score from a two-dimensional hidden memory Markov model. [55] studies the impact of a user’s task based on the browsing behavior such as page load time. The goal of Web-QoE is to optimized traffic management based on QoE for browser-based services load time. [56] focuses on Web quality of experience (QoE) in mobile devices app and their corresponding web services. It applies feasibility study on Android apps such as app startup, clicking, scrolling, and searching. They found a strong difference in traffic patterns between using a web service from an app or from a browser, highlighting the importance of apps for a holistic QoE assessment in mobile networks. [57] focuses on a QoE framework that integrates the Key Performance Indicators (KPI) and Key Quality Indicators (KQI) using the correlation model. The model determines the QoE concerning the quality requirements from the web architecture. Moreover, they build a classification model to predict QoE and applied 10-fold cross-validation to evaluate their model.

Objective and Subjective Quality Criteria: [58] defines an integrated model that use both QoE and QoS for service selection. It uses QoE as historical quality and the current QoS as current
quality of the service. This approach applied multicriteria decision making (MCDM) to rank best service. [59] introduces two algorithms to predict the QoS and QoE based on multiple linear regression and neural networks. The study analyzed the factors that increase the accuracy of the prediction such as distance and occupations of the users. [60] proposes a model that considers the QoE and QoS attributes as criteria for ranking web services. It categorizes quantitative QoS attributes to (technical and non-technical) based on their characteristics to increase the accuracy of the model. The main difference with our work is that we focus on the developers experience and attitude to quantifying the QoE from the developer perspective. We analyze the developers’ communities to build our QoE model.

1.5.4 Service Recommendation

The recent growth of the published Web service on the internet makes Web service recommendation a real challenging and time-consuming task due to the large search space. The recommendation techniques aim to help the users to pick the suitable services for their use. We categorized the proposed solutions based on the techniques used by the researchers. Part of this research has focused on a content-based approach, while other contributions focused on collaborative filtering-based and hybrid recommendation systems. The content-based approach relies on the content similarities of Web services. [6] proposed a cluster-based technique by adopting semantic similarity and association rules between services to cluster services. Then, the generated clusters are filtered by the QoS values to return the recommended services. Several techniques proposed a collaborative filtering strategy, which leverages the historical usage information of the Web services. [7] proposed an approach to recommend services based on usage history by employing the Hierarchical Dirichlet Process (HDP) and Probabilistic Matrix
Factorization (PMF). The HDP discovers the functionally of the services based on their specifications while PMF recommends services based on their usage. Then the approach uses Bayesian theorem to combine the two recommendation methods to obtain the final recommendation. [61] improves the quality of service (QoS) prediction as a recommendation technique to pick high-quality services by using geographical information. It utilizes both locations of users and Web services when picking similar neighbors for the users and service. The experiment applies to the WSDream dataset which includes the IP addresses of all users and the URLs of all Web services to identifying their locations. The hybrid filtering technique combines content-based and collaborative filtering to take advantage of both techniques. [62] proposed a hybrid technique by combining collaborative filtering and textual content. The textual descriptions and tags are extracted from ProgrammableWeb which is the largest Web service and mashup registry. It utilizes a deep neural network to characterize complex relations between mashups and services using a multilayer perceptron network.

1.6 Dissertation Organization

The remainder of this thesis is structured as follows: Chapter 2 introduces our contributions on recommend crowdsourcing team for mashup development. Chapter 3 reports our contribution related to identifying influencers APIs. Chapter 4 reports our work on the quality of experience: developer’s attitude about Web Services. Chapter 5 presents the conclusion and future work.
CHAPTER 2
CrowdMashup: Recommending Crowdsourcing Teams for Mashup Development

2.1 Introduction

The past decade has witnessed an increasing interest in mashup development [8]. For instance, the popular ProgrammableWeb API directory currently includes about 8,000 mashups as of (January 2022). Mashups are Web applications that aggregate pre-existing APIs (or services) to create valuable services with added functionality [63]. Mashup development generally involves several APIs requiring a variety of technological skills such as REST, SOAP, JSON, XML, and security. This often calls for the collaboration of multiple developers to reduce the overall mashup cost (e.g., development time). A large body of research focused on recommending APIs for mashups [64]. However, very few contributions looked at recommending developers to be part of mashup development teams. With the substantial number of available APIs and programmers, finding skilled mashup developers is not straightforward. For instance, ProgrammableWeb lists more than 24,339 APIs. The StackOverflow and GitHub developer community platforms report estimated 16 and 56 millions subscribers as of (January 2022), respectively. Besides, the software industry has recently seen a new trend where crowdsourcing companies (e.g., Topcoder) sell services to corporate, mid-size, and small-business clients, and pay community members (i.e., developers) for their work. These companies also organize open tournaments and programming challenges in which programmers are organized in teams to compete against each other. Therefore,
it is important to form balanced teams with skilled developers. Selecting appropriate developers should be performed carefully to improve productivity [32]. In the context of mashups, two factors contribute to successful developer recommendation. First, mashups involve various APIs that require a large array of skills. A recent study shows that the interest of project members toward specific tasks leads to better outcomes [36]. Hence, it is vital to pick developers that possess the right skills, demonstrate significant interest in the mashup, and have a good reputation among their peers. Second, it is necessary to form teams with members that can get along with each other. Studies have confirmed that strong social relationships among members increase team performance [65]. Most interactions among mashup and API developers take place via online communities such as StackOverflow and GitHub. Positive discussions between developers, through questions and answers, tend to increase their social ability and productivity.

Moreover, using pre-existing services to design new applications has various advantages [66] to speed up the development such as: (I) it is easier to reuse software that have been established previously than trying to recreate them. (II) Providing consumers with the ability to utilize cloud resources (e.g., memory and CPU). (II) APIs providers are in charge of maintaining local infrastructures for example networking and energy. (III) Reducing interdependencies between application parts so that updates or failures in one do not affect other services. In this chapter, we propose CrowdMashup, a crowdsourcing-based approach for recommending teams of developers for mashups. We analyze StackOverflow and ProgrammableWeb to generate teams that best satisfy mashup requirements. To the best of our knowledge, this is the first work to address recommendation in mashups from developer’s perspective. The main contributions of this chapter are summarized below:
• We use natural language processing (NLP) [67] to assign interest scores to developers in using APIs. We utilize two NLP models Stanford CoreNLP and Spacy to perform this task. As developers may omit to comment on certain APIs, we predict missing scores using the alternating least square method for collaborative filtering [68] and parallel matrix factorization [69]. We combine the computed interest scores and reputation values of developers in the community to quantify their skills.

• We define a sociometric to assess social relationships among developers in the community. Sociometry is a quantitative method in psychology for measuring social relationships [25]. We model interactions (comments and replies) among developers as a weighted undirected graph. The weight of each edge represents the number of interactions between developers modeled as nodes.

• We propose three techniques to generate teams from mashup queries (i.e., specification of the mashup requirements). We compare the proposed techniques to analyze team performance by conducting experiments on real-world data and APIs.

  – The first technique is graph-based and adopts the concept of cliques from graph theory to identify strongly related developers [70]. A clique is a subset of vertices from the sociometric graph where every two distinct vertices are adjacent. We compare the skills of the developers in the clique along with their sociometric scores to recommend top-t teams.
– The second technique is clustering-based and utilizes Louvain model [71] to detect communities in the graph. Since the result of the Louvain model is a set of communities that have strong connectivity, we adapt the detection hierarchical clustering algorithm in order to partition the community into teams that best fulfill mashup requirements.

– The third technique is search-based and uses a genetic algorithm [72] to generate teams from mashup queries. We adopt the algorithm to ensure the connectivity between suggested developers and guarantee the size of the team based on mashup requirements.

2.2 Scenario

2.2.1 Fixed Team Size

![Diagram](image)

Figure 2.1 *CrowdMashup* for Competition and Tournament
Let us consider a crowdsourcing company, such as TopCoder, with an open global community of designers, developers, data scientists, and competitive programmers. The company holds many programming and design tournaments (e.g., Mashathon by TopCoder) where developers are asked to build complete products using sponsor APIs. To increase competition between teams and deal with all teams fairly, the Mashathon committee looks to form teams with balanced performance and fix team size Figure (2.1). For example, if the APIs sponsors are GoogleMap API, Tunein Radio API, and Yahoo Weather API, then developer teams may develop a mashup and call it World Graph Broadcast which employs mapping and radio service. However, other teams may develop a mashup that includes all offered APIs to create World Graph Radio mashup. The aim of this mashup is to provide real-time weather and radio on the world map. Since the skills of the team members are different, the level of consuming the services is different. Therefore, recommending developers to be part of mashup development teams is not straightforward. Accordingly, it is important to form balanced teams with skilled developers. In the context of the mashups development team, the CrowdMashup focus on two factors to create balanced teams. The first factor is picking developers that possess the right skills, demonstrate significant interest in the mashup, and have a good reputation among their peers. The second factor, it is necessary to build teams with members that can get along with each other (strong social relationships). Recommending based on those factors leads to increase teams’ performance and reduces the overall mashup cost (e.g., development time).

2.2.2 Flexible Team size

Suppose that a real estate company would like to build a mashup service that offers advanced information related to the property list. The aim of using this information is to advise the seller
about the area such as comparing the crime rate of different areas and calling it Safe Real Estate Mashup. The company has a limited budget and time for this project.

Figure 2.2 CrowdMashup for Business and Enterprise

Therefore, they want to use crowdsourcing for building this service by assigning a team that can handle the project within the budget and time. The team can be formed of 2 to 3 members, this depends on the member’s skills such as overlapping skills, time, and budget. The needed APIs in the real estate mashup are Zillow API (for real estate list), FBI crime rate API (for crime rate information), and Google Maps API (for mapping service). In that regard, those APIs need a variety of technical skills such as REST, SOAP, JSON, XML, and security. For that reason, let’s consider three individuals, Alex, Bob, and Joe. Alex and Bob have a background using Zillow and Google Maps APIs and Joe has sufficient experience with all listed APIs. In that concern, the team can be designed differently with two or three members. In the case of a team with two members,
it would seem best to put Joe and Alex or Joe and Bob together because Joe has sufficient information about all APIs. However, a team of Alex and Bob has a considerable overlap of information so there is no benefit of this overlapping knowledge. Moreover, in the case of three members as a team of Alex, Bob, and Joe may accelerate the project but should satisfy the project budget. Therefore, recommending developers to be part of mashup development teams is not straightforward. The CrowdMashup recommend teams based on analyzing developer community to return the best team performance Figure 2.2. It applies suggestion features such as team size (Minimum/Maximum member) to consider the overlapping skills, budget, and time limitation.

2.3 CrowdMashup Approach

The CrowdMashup architecture (Figure 2.3) is composed of two major components: Social Analysis of Developers Community (SADC) and Team Finder (TF).
Regardless of whether or not there is a request to form mashup development teams. It uses data from the StackOverflow community to quantify and predict developer interest in adopting and using APIs. Developer forums have evolved into a troubleshooting handbook, with many developers sharing their experiences, concerns, and answers [73]. For example, StackOverflow has more than 22 million questions and 33 million answers in 2022. The LinkedIn API uses StackOverflow as a reference, at their official page, to support programmers in technical issues. The amount of affinity among developers is also displayed in developer communities. Because of their involvement in online groups, many programmers may find up working together on projects [65].

TF is activated when the mashup administrator sends a mashup query to it. It returns the most effective teams that meet the mashup query’s specifications. A mashup administrator is a user or entity who is searching for developers to work on a mashup with them. A possible mashup administrator is Topcoder. It hires individual community programmers to work on specific projects and provides software development services to third-party clients. It also hosts design competitions and consequently provides clients with design services.

2.3.1 Social Analysis of Developers Community (SADC)

SADC analyzes StackOverflow to create three data structures (Figure 2.3): interests table ($U_I$), reputation table ($\bar{U}_R$), sociometric graph (SG), and sociometric graph Plus (SG –Plus).

**User Interests Table** ($U_I$) - The first step in analyzing the developer community is to identify a set of APIs that they utilize. To that end, we crawled all APIs from ProgrammableWeb and
extracted the name and primary category of each service using the Scrapy framework. Since StackOverflow has about 55 millions comments (questions and answers), we focused on the ones that are related to APIs. We filtered StackOverflow comments using the API names retrieved from ProgrammableWeb. The next step is to analyze developers’ comments and assign scores of interests in using APIs. For that purpose, we applied sentiment analysis to get the interest score \( U \) for each user. We parsed developer’s comments using SpaCy, a high-performance natural language processing (NLP) framework that includes robust pre-trained linguistic models for sentiment analysis.

For example, the comment “... Google Visualization API has several ways to do each task so it’s important to know what you have already done, and we could start there...” returns a positive interest value about Google Visualization API. An example of negative interest about Google Maps API is: “I simply have no experience with the Google Maps API ...”.

![Interests Table](image)

**Figure 2.4 Interests Table**

Since certain APIs are not discussed by some developers, we ended-up with missing interest scores (Table 2.4a). To solve this problem, we utilized alternating least squares algorithm for implicit data based on latent factors (ALS implicit) [75]. Moreover, we employed a parallel Matrix

---

6 https://scrapy.org/
7 https://spacy.io/
factorization algorithm called (LIBMF) [69] Accordingly, we completed interest scores for all developers and APIs as shown in Table 2.4b. If an API is listed on ProgrammableWeb but unknown (i.e., not discussed) on StackOverflow, then ALS cannot complete the missing interest scores for this API. To deal with this issue, we average the interest scores of $u_i$ for all APIs on StackOverflow that have the same category as the unknown API. Then, we assign the average score as $u_i$’s interest score for this API. If no API with similar category is commented by $u_i$, we average $u_i$’s interest scores for all APIs discussed by $u_i$.

<table>
<thead>
<tr>
<th>Recommendation Algorithm</th>
<th>RMSE</th>
<th>MSE</th>
<th>MAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALS implicit</td>
<td>1.997</td>
<td>3.988</td>
<td>1.976</td>
</tr>
<tr>
<td>LIBMF</td>
<td>0.212</td>
<td>0.045</td>
<td>0.149</td>
</tr>
</tbody>
</table>

Table 2.1 Performance Evaluation of Recommender Systems

Moreover, we compare the performance of ALS implicit and LIBMF using recommenderlab library in r. Table 2.1 shows the error of the recommended algorithms. Root mean squared error (RMSE), mean squared error (MSE), and mean absolute error (MAE) are reported. RMSE measures the standard deviation of residuals, MSE measures the variance of the residuals and MAE measures the average of the residuals in the dataset. LIBMF shows lower error rate, therefore, we consider it as in algorithm to generate user interests table.

User Reputation Table ($U_R$) - StackOverflow has a reputation system which provides the level of expertise $U_R (u_i)$ for each user $u_i$. Since the extracted reputation has highly distributed values, we applied the z-score normalization to write reputation values into a standardized structure. The
following formula shows the final reputation $\overline{U_R}$ for $u_i$, where $\mu$ and $\sigma$ represent the mean and standard deviation of all reputation values, respectively:

$$\overline{U_R}(u_i) = \frac{U_R(u_i) - \mu}{\sigma}$$  \hspace{1cm} (1)

**Sociometric Graph (SG)** - Another major aspect in teams formation is the social ability, or sociometry, among developers [25]. The idea is to make sure that members of the same team can actually work together. Studies showed that social relationships among members of the same team have a positive impact on improving the team productivity [76]. In our approach, we use interactions among developers via questions and replies in StackOverflow as a mean to estimate their social relationships. Developers that engage in more conversations with each other in online communities have more chances to successfully collaborate.

We scanned the history of interactions among developers in StackOverflow regardless if the questions/replies are related to APIs or not. Then, we modeled those interactions as an undirected weighted graph, called sociometric graph (SG). Each node in the graph represents a user (Figure 2.5).
2.5a). An edge \((u_i, u_j)\) states an existing interaction (question or reply) between users \(u_i\) and \(u_j\).

Developers may interact at various levels, from few questions/replies to thousands. To capture this aspect, we label each edge \((u_i, u_j)\) with a weight \(W_e(u_i, u_j)\) that gives the number of interactions between users \(u_i\) and \(u_j\):

\[
W_e = (u_i, u_j) = \# \text{interactions between} \ (u_i, u_j)
\]  

Figure 2.6 Sociometric Plus Graph Process

Sociometric Graph Plus (SG-Plus) - We presented a sociometric graph (SG) which captures the interaction of the developers only. Moreover, we proposed the Sociometric Graph Plus (SG-Plus) which aims to provide valuable information to the graph. Figure 2.6 shows the process of creating the SG-Plus. We converter the SG graph into an adjacency matrix and combine it with developers' expertise to reflect the sociometric information with the skills of the developers in the graph. The SG-Plus relied on the mashup query to identify the interest of the developers for specific API. For that reason, a new SG-Plus will be created when the mashup administrator submits a new query.
Therefore, developers may interact at various levels, from few questions/replies to thousands. To capture this aspect, we label each edge \((u_i, u_j)\) with a weight \(W_e (u_i, u_j)\) that gives the number of interactions between users \(u_i\) and \(u_j\). In addition, we evaluate the skills of each user (i.e., developer) \(u_{i,j}\) in the community based on their interest and reputation to calculate the weight for each edge:

\[
W_e = (u_i, u_j) = \sum_{u_i, u_j \in SG-Plus} \# interactions between (u_i, u_j) + \hat{U}_R(u_i, u_j) + User_{skills}(u_i, u_j) \tag{3}
\]

### 2.3.2 Query Model

Mashup administrators interact with CrowdMashup through mashup queries. A mashup query \(Q\) defines the mashup requirements through \(Q = (t, [\text{min, max}], A, \delta)\) where:

1. \(t\): is the number of required teams.

2. \(\text{min}\): is the number of members within each team.

3. \(\text{max}\): is the number of members within each team.

4. \(A\): is a list of APIs that compose the mashup.

5. \(\delta\): The range to be considered as overlapping skills between the team members.
Each element in the list A is defined as \(< API_{ID}, API_w >\). APIID is an ID that uniquely identifies the API. \(API_w\) is the weight (in the range 0 to 1) of the API. It represents the level of importance of the corresponding API in the mashup. For instance, a location-based mashup (e.g., transportation) may rely on a mapping API; the mapping API should be given a significant weight value to make sure the most skilled developers are recommended for this API. A small \(API_w\) implies that the API may not be mastered by all teams members; a big \(API_w\) indicates that the API should be mastered by most teams members.

**Example 1.** Assume we want to build 5 teams of minimum 2 and maximum 3 developers for a mashup that composes GoogleMaps (with ID 1 and weight 0.6), Foursquare (with ID 4 and weight 0.4) and Last.fm (with ID 5 and weight 0.1). The mashup query is specified by \(t=5\), \(min=2\), \(max=3\), \(A=[< 1; 0.6 >; < 3; 0.4 >; < 5; 0.1 >]\), and \(\delta=0.5\). Mashup administrators may not want to limit
mashups to specific APIs by providing the list of API categories instead of APIs. For instance, they may refer to ‘Social” as a required category instead of Facebook or Twitter. In this case, we automatically fetch all APIs that belong to the categories listed by the administrator from ProgrammableWeb and replace each category by the matching APIs.

Example 2. Assume we want to build 5 teams of minimum 2 and maximum 3 developers for a mashup that composes APIs from the Mapping and Social categories with 0.6 and 0.4 weights, respectively. Assume that APIs with IDs 1, 30, and 47 belong to Mapping and APIs with IDs 3, 17, and 22 relate to Social. The query is specified by \( t=5, \min=2, \max=3, \text{ and } A= [< 1, 0.6 >, < 30, 0.6 >, < 47; 0.6 >, < 3, 0.4 >, < 17, 0.4 >, < 22, 0.4 >] \) and \( \delta =0.1 \).

In addition, the query model interacts with different algorithms to generate mashup teams based on the mashup requirements using the same parameters. However, we describe how the query model interacts with the proposed algorithms. As shown in Fig 2.7., the pseudo-code of query model, where the empty list \( L \) is defined to hold all generated teams. Since we have different algorithms to generate teams, the query model supports the selected algorithms option which is based on the algorithm ID. Then the mashup query \( Q \) is passed to generate the targeted teams. Those teams will be added to \( L \) to hold the generated teams from the selected algorithm. Moreover, in order to increase the overlap skills between the members, we need to reduce the size of the team and \( \delta \) which leads to an increase in the team performance. Finally, the list \( L \) will be sorted by team performance and return the top suggested teams to the mashup administrator.
2.3.3 Team Finder (TF)

TF generates teams that best satisfy the mashup query requirements. It uses three different techniques to generate teams (graph-based, cluster-based, and search-based). For the graph-based technique, we proposed two algorithms (*Crowdsourcing Team Generation* (CTG) and *Crowdsourcing Team Generation Plus* (CTG-Plus)). The CTG algorithm uses SG as input while all other algorithms used SG-Plus as input. Before describing how TF generates teams, we introduce the metrics to calculate the performance of a team. We evaluate the skills of each user (i.e., developer) \( u_i \) in the community based on \( u_i \)'s reputation and interest in each \( API_j \in A \) specified in the mashup. The user's interest in \( API_j \) is multiplied by \( API_w^j \) to take into account the weight (i.e., importance) assigned by the administrator to each API:

\[
User_{skills}(u_i) = \hat{R}(u_i) \times \sum_{API_j \in A} U_I(u_i, API_j) \times API_w^j \quad (4)
\]

Based on the skills of each user \( u_i \) given in formula (4), we define the skills of a team \( T \) composed of \( m \) members as the sum of the skills of all members:

\[
Team_{skills}(T) = \sum_{u_i \in T} User_{skills}(u_i) \quad (5)
\]

Using the sociometric graph \( SG \), we also introduce the *sociometric score* of \( T \) to quantify the level of collaboration between members. The sociometric score \( Team_{socimetric}(T) \) of \( T \) accumulates the weights of all edges that connect members in \( T \) and divide it by the number \( m \) of team members:

\[
Team_{socimetric}(T) = \frac{\sum_{u_i \in T, u_j \in T} W_e(u_i, u_j) \in SG}{m} \quad (6)
\]
From formulas (4) and (5), we define the overall performance of \( T \) by summing the skills and sociometric of the team:

\[
Team_{Performance}(T) = Team_{skills}(T) + Team_{sociometric}(T)
\]  

(7)

### 2.3.3.1 Graph-Based Techniques

We proposed two graph-based algorithms which generate teams that best satisfy the mashup query requirements. The *Crowdsourcing Team Generation* (CTG) and *Crowdsourcing Team Generation Plus* (CTG-Plus) utilized a graph theory methodology to build the targeted team.

**Crowdsourcing Team Generation (CTG)** - the algorithm uses as input the sociometric graph \( SG \) as well as interests and reputation tables, \( U_i \) and \( \hat{R} \). Before describing the CTG algorithm, we introduce the metrics to calculate the performance of a team based on \( SG, U_i \), and \( \hat{R} \).

The CTG algorithm (Algorithm 1) identifies strongly connected members in the sociometric graph \( SG \) using the concept of *cliques* in graph theory. A *clique* \( C \) is a subset of vertices of an undirected graph such that every two distinct vertices in \( C \) are adjacent [70]. We use the *Bron Kerbosch* algorithm [70] to return cliques in the AllCliques list (line 3). Another important data structure is *SharedCliques* (lines 1 and 16). Each element \( SC \) in this list contains *common vertices* between cliques as well as the remaining vertices (called *potential vertices*) in the cliques. For example, Figure 2.5b depicts two adjacent cliques \( C_1 = \{u_1, u_2, u_4\} \) and \( C_2 = \{u_2, u_3, u_4\} \). The common and potential vertices are defined by \( SC.common=\{u_3, u_4\} \) and \( SC.potential=\{u_1, u_2\} \), respectively. Due to space limitation, we omit the algorithm for the *GetSharedCliques*() function.
CTG uses AllCliques and SharedCliques to recommend the top-t (t is the number of required teams). Each element in the returned TeamsList is composed of the team’s members and performance of the team as defined in formula (7). The algorithm first looks for cliques of size m (i.e., cliques with required number of members). If more teams still need to be generated (TeamsList.size() < t), then CTG explores the shared cliques.

Algorithm 1: Crowdsourcing Team Generation (CTG)

```
Input : $U, N$ Table, $U, I$ Table, Sociometric Graph $SG$, Mashup Query $Q$
Output: TeamsList (recommended teams)
1 SharedCliques ← null;
2 TeamsList ← null;
3 AllCliques ← BronKerboschAlgorithm($SG$);
4 foreach $C$ ∈ AllCliques do
5    if (C.size() == m) then
6       $T$ = All users from $C$;
7       Calculate Performance$_{Team}$ ($T$) of team $T$;
8       TeamsList.add($T$, Performance($T$));
9       AllCliques = AllCliques − $C$;
10   end
11 end
12 if (TeamsList.size() >= t) then
13   TeamsList ← sort(); //By team performance
14   Return Top-t teams from TeamsList;
15 end
16 SharedCliques = GetSharedCliques(AllCliques);
17 foreach $SC$ ∈ SharedCliques do
18    if (SC.common.size() == m) then
19       $T$ = Top users from SC.common; //By skills or sociometric
20       Calculate Performance$_{Team}$ ($T$) of team $T$;
21       TeamsList.add($T$, Performance($T$));
22       SharedCliques = SharedCliques − $SC$;
23    end
24 end
25 if (TeamsList.size() >= t) then
26   TeamsList ← sort(); //By team performance
27   Return Top-t teams from TeamsList;
28 end
29 foreach $SC$ ∈ SharedCliques do
30    if (SC.common.size() < m) then
31       $T$ = Users from SC.common + remaining top from SC.potential; //By skills or sociometric
32       Calculate Performance$_{Team}$ ($T$) of team $T$;
33       TeamsList.add($T$, Performance($T$));
34       SharedCliques = SharedCliques − $SC$;
35    end
36 end
37 TeamsList ← sort(); //By team performance
38 Return Top-t teams from TeamsList;
```

Figure 2.8 Crowdsourcing Team Generation
We identify the following three cases during team recommendation:

**Case 1: Cliques have \( m \) members** (lines 4-15) - CTG first parses cliques with the exact number of members. If the size of a clique \( C \) is \( m \), then all members of \( C \) are used to form a team. We calculate the performance of \( T \), insert \( T \) and its performance to \( TeamsList \), and remove \( C \) from \( AllCliques \). If \( TeamsList \) reaches the desired number \( t \) of teams (lines 12-15), \( TeamList \) is sorted based on performance and the top-\( t \) teams are returned, hence ending the algorithm. Otherwise, we process shared cliques (Case 2).

**Case 2: Shared cliques have at least \( m \) members** (lines 16-28) - CTG processes shared cliques that have enough members in their common vertices. It picks the top-\( m \) members from common vertices using one of two selection options (line 19). (i) **CTG by Skills**: \( m \) members with the highest skills are selected; and (ii) **CTG by sociometric**: \( m \) members with the highest sociometric scores are selected. The corresponding teams are inserted into \( TeamsList \) as described in Case 1; the shared cliques used to build the teams are removed from \( SharedCliques \). If \( TeamsList \) reaches the desired number \( t \) of teams (lines 25-28), \( TeamList \) is sorted based on performance and the top-\( t \) teams are returned, hence ending the algorithm. Otherwise, we proceed to Case 3.

**Case 3: Shared cliques have less than \( m \) members** (lines 29-38) - CTG handles the shared cliques that do not have enough members in their common vertices. It picks the remaining members from the potential vertices in the shared cliques. The remaining members are selected using CTG by **Skills** or CTG by **Sociometric** as described in Case 2 (line 31). Teams along with their calculated performance are added to \( TeamsList \) and the top-\( t \) teams are returned.

**Crowdsourcing Team Generation Plus (CTG-Plus)** - the algorithm identifies strongly connected members in the graph as CTG, but it uses a sociometric graph Plus SG - *Plus*. It utilizes
the concept of *cliques* in graph theory. A *clique* $C$ is a subset of vertices of an undirected graph such that every two distinct vertices in $C$ are adjacent [70]. We use the *Bron Kerbosch* algorithm [70] to return cliques in the *AllCliques* list.

CTG-Plus and *AllCliques* to recommend teams. If the team size is more than the existing clique’s size, then the CTG-Plus will explore the connected vertices with the highest weight. Therefore, the performance of the teams will be the sum of weighted edges of the SG - *Plus*:

$$Team_{\text{Performance}}(T) = \sum_{u_i \in T} SG - Plus_{\text{edge}}(u_i) \quad (8)$$

Moreover, the CTG-Plus will recommend teams that have connected vertices. This will ensure the social relation between the team members.

### 2.3.3.2 Cluster-Based Technique

The concept of community detection and graph clustering has been used as a method for finding groups within a complex graph in social platforms. We proposed a Cluster Based CB algorithm. Algorithm 3 adapts two pre-existing clustering algorithms to find the best team performance. We applied Louvain algorithm [71] which is consider as community detection algorithm. This algorithm based on the modularity measure and a hierarchical approach to detect communities in the network. The result of this cluster can not specify the size of the cluster. Therefore, we employ a hierarchical cluster to divide the detect community (sub-graph) into the target cluster size. For the hierarchical cluster, we utilize *Canberra* distance method to find the similarities between each cluster. we used an algorithm implemented using the *igraph* package in R.
Figure 2.9 Clustering Process

Figure 2.9 illustrates the cluster-based algorithm process. The algorithm begins with identifying strong communities in the SG - Plus. The extracted communities (sub-graphs) convert into adjacent matrices to be used as input for the hierarchical cluster. Afterward, the clustering algorithm starts to identify a team of developers with the target size. The algorithm will the cluster which have connected vertices only in the graph.

```
Algorithm 2: Cluster Based (CB)

Input : SG - Plus, mashup - query
Output: top - k - teams
1 find connected graph and list it in S sub-graph list.
2 foreach s ∈ S do
3     Louvain algorithm for each i ∈ S.
4     if i ≤ max then
5         calculate the graph weight of i;
6         store i in list l;
7     else
8         if l > max then
9             foreach h ∈ i do
10                hierarchical clustering for i;
11                calculate the graph weight of h;
12                store h in list l;
13          end
14      end
15 end
16 Sort i list by performance
17 return top-k from l
```

Figure 2.10 Cluster Based Algorithm
2.3.3.3 Search-Based Technique

Genetic algorithms are one of the famous algorithms applied in search and optimization problem efficiently [77]. The algorithm based on the mechanics of natural selection. 2.9. illustrates the steps of finding the final solution. The first step is to generate the initial population of individuals. The individual represented as chromosome where the chromosome composed of a sequence of genes. Next, individuals are judged using a fitness function, and several genetic operators are involved to create a new population. Those operators represent better solutions based on the selection principles. This process is repeated until reaching the best result.

![Genetic Algorithm Process Diagram](image-url)
The chromosome C represent as solution for recommended team. The C composed of a sequence of developers as genes. We denoted C as a string of selected developers IDs. For sake of simplicity, let's assume that C has only five developers as a recommended solution (Alexis, Sarah, Jordan, Emma and Lily). Fig2.10 encodes a candidate solution for C.

![Figure 2.12 Solution Representation](image1)

![Figure 2.13 Example of Team Sub-Graph](image2)

we configured the fitness function to maximize the team performance to find the best team performance using formula (8). The selection technique is based on the fitness function and ensure the next generated team chromosome will not include duplication. We implement genetic algorithm using GA library in R [72].
2.4 Implementation and Performance

In this section, we describe the CrowdMashup prototype implementation. Then, we evaluate the performance of our approach using real-world data and APIs.

2.4.1 CrowdMashup Prototype

We implemented a CrowdMashup prototype in Java using CTG algorithm. We used Google BigQuery\(^8\) to retrieve comments about APIs from StackOverflow. We collected 8,617 comments related to 583 APIs. We used the Jgrapht library\(^9\) to handle graphs and identify cliques. We utilized Stanford Natural Language Processing library to calculate developers' attitude (interest) toward APIs. We used Apache Spark's scalable machine learning (MLlib) library\(^{10}\) to deal with missing developers' interest values.

---

8 https://cloud.google.com/bigquery/public-data/stackoverow
9 http://jgrapht.org/
10 https://spark.apache.org/
Figure 2.14 The *CrowdMashup* User Interface

Figure 2.14 shows *CrowdMashup*’s graphical interface. Mashup administrators specify their queries through the *Mashup Query pane* (top left). They assign the number of required teams and members in each team. Administrators enter either a list of specific APIs or generic API categories along with their weights. They also pick the algorithm to be used to generate teams: (1) *Skills Only*: members are selected based on skills only. (2) *Sociometric Only*: members are selected based on sociometric only. (3) *CTG-Skills*: uses both skills and sociometric but gives priority to skills in dealing with shared cliques (lines 19 and 31 in Algorithm 1). (4) *CTG-Sociometric*: uses both skills and sociometric but gives priority to sociometric in dealing with shared cliques (lines 19 and 31 in Algorithm 1). The generated teams are shown in the *Recommended Teams* pane (bottom left). The pane shows each recommended team as a list of
developer IDs. It also displays the calculated performance of each team and orders the generated teams based on their performance. The Team Analysis pane (right) displays the two metrics for team recommendation: sociometric sub-graph and team performances illustrated in a bar graph to visualize the performance of different teams. The time to generate teams is also shown in this pane.

2.5 Experiment

The aim of the experiment is to assess the ability of proposed algorithms to select teams with the best performance. We begin with the CTG algorithm, we ran our experiments on a 64-bit Windows 10 environment, in a machine equipped with an Intel i7-7700HQ and 16 GB RAM. We measured the performance of the generated teams using three non-CTG algorithms: Random (members are randomly selected), Skills Only, Sociometric Only; and two CTG algorithms: CTG-Skills and CTG-Sociometric.

![Figure 2.15 Single Query Team Performance for Non-CTG (Random, Skills Only, Sociometric Only) and CTG (CTG-Skills, CTG-Sociometric) Algorithms](image-url)
Figure 2.15 compares the five algorithms using the same mashup query to generate four teams with seven members per team. First, we compare CTG vs. non-CTG algorithm in terms of team performance. CTG algorithms perform better than non-CTG algorithms due to combining sociometric and skills. Besides, CTG-Skills generates better teams than CTG-Sociometric. This is because vertices that are outside cliques are unlikely to return high sociometric values. Then, we compare the distribution of the performance of the four teams recommended by each algorithm. Figure 2.15 shows that team performance decreases steadily from the first to the last team in both CTG algorithms. Hence, CTG shows more balanced teams than non-CTG algorithms. For instance, there is significant difference (more than double) between the performance of the first and second teams in the Sociometric-Only algorithm.

![Figure 2.15](image)

**Figure 2.16 Multiple Queries Team Performance for Different Team Sizes**

We also conduct experiments to explore how forming teams with various sizes is handled by CTG. We randomly generated queries with sizes 5, 10, 15, 20, 25, and 30. We had 5 queries
for each time size (for total of 30). As shown in Figure 2.16, CTG algorithms always show better team performances than the non-CTG algorithms regardless of the team size. This is because non-CTG algorithms ignore sociometric, skills, or both (in the case of random). Overall, generating teams with bigger sizes (more than 10 members) leads to lower performance, as it is harder to find a large number of developers with the right skills and social relationships. Studies have shown that 3-7 developer teams are key to successful software projects (3-5 person teams would be the best)\(^{11}\). Hence, this makes CTG a suitable technique for team recommendation. For large team sizes (e.g., 25), CTG-Skills shows better team performance than CTG-Sociometric as finding cliques or shared cliques with larger sizes becomes challenging. For teams of size 2-5, CTG-Skills and CTG-Sociometric are comparable, and they largely outperform the three other algorithms: Random, Skills Only, and Sociometric Only. In teams of size 6-10, CTG-Sociometric shows better team than CTG-Skills as finding cliques or shared cliques with size 10 is still possible and improves the overall team performance.

However, we compare the \textbf{search-based}, \textbf{cluster-based}, and \textbf{CTG-Plus} algorithms. These algorithms employ the SG-Plus graph as an input. We ran our experiments on a 64-bit Windows 10 environment, in a machine equipped with an Intel i9-9900K and 32 GB RAM. We ran all experiments on real-world data and APIs from StackOverflow and programmableWeb.

\(^{11}\) http://www.qsm.com/process improvement 01.html
Figure 2.17 Best Teams Performance

Figure 2.17 compares the three algorithms using the same mashup query to generate different team sizes. CTG Plus algorithm performs better than search-based and cluster-based algorithms due to finding all candidate teams in the graph. For this reason, the algorithm can find the maximum team performance. This is because all potential cliques are considered as teams from the SG-Plus graph. However, the search-based (genetic algorithm) shows better performance than the cluster-based because it used an optimization technique based on the principles of genetics and natural selection. The cluster-based algorithm shows a lack of electing members on forming teams due to the difficulty of finding high talents with a strong relationship.

Then, we compare the running time of the proposed algorithms using different team sizes. Figure 2.18 shows that the CTG-Plus and cluster-based outperform the genetic algorithm in computation time. The CTG-Plus and cluster-based took less than 7 seconds to recommend teams with 10 developers however genetic algorithm took around 22 minutes to generate the
same team size. Because finding connected team member constraint is added in the genetic algorithms process which leads to an increase in the computation time to find optimal teams.

Figure 2.18 Time Analysis

2.6 Conclusion

We propose the CrowdMashup approach to recommend teams for mashup development. The approach analyzes the StackOverflow developer community to infer developers’ skills in using APIs. The CrowdMashup models the ability of developers 49 to collaborate with each other via a sociometric graph. The second phase recommends crowdsourcing teams that best satisfy the requirements of a mashup query. We proposed four algorithms using three different techniques (cluster-based, search-based, and graph-based) to generate teams that best fulfill mashup requirements. We provide a prototype implementation and conduct experiments on real-world data and APIs from StackOverflow and programmableWeb to evaluate our approach. Experiments show promising results in generating efficient teams using the graph-based algorithm for mashup development.
CHAPTER 3

FAME: An Influencer Model for Service-Oriented Environments

3.1 Introduction

Service-oriented computing (SOC) has become the forefront of modern software development and is gradually being implemented in the software industry. In general, organizations and businesses are exporting their applications as Web services or APIs (Application Programming Interface) [64]. According to the W3C (World Wide Web Consortium) "A web service is a software system designed to support interoperable machine-to-machine interaction over a network". These software systems are self-contained, distributed, interactive applications that can be invoked or published over the internet. Communication among Services/APIs is based on the principles of SOAP (Simple Object Access Protocol), REST (Representational State Transfer) or RSS/Atom feeds [3].

However, integrating multiple APIs created by diverse third parties requires a wide array of technical skills such as Web (e.g., REST), data management (e.g., JSON), programming (e.g., SDKs), and security (e.g., authentication) [2][78]. To overcome these challenges, developers often turn to programming communities (e.g., GitHub) to share practices, knowledge, experience, and brainpower in solving intricate problems. For instance, GitHub reports (as of May 2019) having over 37 million users and 100 million repositories. Gathering and analyzing content about API usage and activities in existing communities (e.g., number of bugs, developer feedback,
number of mashups) provides opportunities to better understand developers’ interactions with APIs and detect relationships between APIs and mashups [1][79].

Combining social network analysis with service-oriented computing could bring novel insights to service selection, recommendation, and composition [1][15]. One particular research area that received significant attention in social computing is social influence [80]. Influencers have the power to impact the way others in the entire network behave or think [41]. Considerable work has been conducted to model influence or identify influencers in social media [40][41][42][43]. The research proposed in this chapter approaches the concept of influencers from an API perspective. We define an API as influential if it is widely adopted in mashup and service-oriented application development. The more influence an API has, the more interest that API sparks to developers. We perceive API developers (both consumers and providers) in programming communities as social actors. API consumers use existing APIs to build mashups and service-oriented applications. They share experiences, feedback, and opinions about APIs in various ways such as participating in discussion forums, reporting bugs, and following APIs. API Providers are the developers that created the APIs available in the community. They publish important information about their APIs such as tutorials, articles, SDKs, libraries, new releases, and source code.

Analyzing social content to identify influencer APIs has several advantages. First, API consumers will be able to integrate the best-in-class APIs. For instance, the ProgrammableWeb directory lists more than fifty mapping APIs. Selecting the right API is time-consuming and may have an impact on future mashup maintenance and development [1]. Second, consumers may have different views on what makes an API relevant. Some consumers may value APIs with the least number of reported bugs.
Others may consider the opinions expressed by peers toward the API as significant. Measuring influence based on various API features assists consumers in selecting APIs that are most suitable to their development style, needs, and requirements. Third, identifying influencer APIs enables providers to increase the visibility of their APIs and set up a strategy to reach out to a larger audience of developers. Providers will be able to compare their API's influence with the influence of a competitor's API and pinpoint a plan of actions to promote their APIs. Some providers may, for example, decide to enhance their involvement in discussion forums, while others may choose to increase the number of articles and tutorials published about their APIs. Developing an influencer model for service-oriented environments poses several research challenges. First, social data are scattered across multiple independent platforms and cannot be accurately obtained from one single source. For instance, the number of applications that use a given API is determined by looking at mashups listed in ProgrammableWeb and repositories hosted in GitHub. Besides, current platforms return API-related data in heterogeneous formats. For example, posts in StackOverflow and commit comments in GitHub are textual and require natural language processing techniques. News articles in HackerNews and bug reports in Bugzilla are presented in proprietary formats. Other data such as the number of issues in GitHub, number of posts in StackOverflow, and number of followers on ProgrammableWeb are returned as atomic values on different scales. Second, the social content collected from existing communities deals with various aspects of the APIs. It covers both technical (e.g., number of SDKs) and non-technical (e.g., number of API followers) information. This includes information about the API itself (e.g., number of change logs representing the API's evolution), API consumption (e.g., number of projects that use the API), and API social activities (e.g., number of posts and articles related the API). A multi-dimensional influencer model that captures various API features is needed. Once API features are gathered and
cleaned, it is important to determine the extent to which an API is influential and the features that have or do not have an impact on the API influence and, if so, to what degree. Third, newly developed APIs lack the social content necessary to assess their overall influence. Therefore, recommender systems based on API influence scores may omit to return such APIs. This may lead to the starvation of newly deployed APIs as they lack the required social presence. The proposed influencer model should allow bootstrapping the influence score of newly created APIs, hence overcoming the traditional cold start problem.

The identification of influential nodes in social networks has been the subject of many research efforts [40][41][42][43]. Existing research devoted to influencers in software ecosystems emphasizes on developers as influencers not APIs [44][45]. In this chapter, we propose FAME (inFluencer Apis in developer coMmunitiEs), an influencer model for APIs in service-oriented environments. To the best of our knowledge, FAME is the first approach to consider APIs (instead of developers) as influencers in building mashups and service-oriented applications. The main contributions of this chapter are summarized below:

- We propose an influencer model that extracts more than eighteen API features from multiple programming communities. The extracted features capture non-technical and technical information about APIs in various formats such as text, atomic values, and other proprietary structures. We perform sentiment analysis to quantify developers' opinions towards using APIs. We introduce a cumulative API Influence Score (AIS) to assess the influence of APIs in mashups and service-oriented applications. We also categorize APIs into tiers based on their influence scores.
- We predict the evolution of the influence scores of newly deployed and existing APIs using Non-Negative Least Square (NLS) linear regression technique. We conduct an
analytical study to determine the degree to which each extracted API feature impacts the influence score.

- We conduct experiments on four real-world programming platforms: GitHub, StackOverflow, HackerNews, and ProgrammableWeb. We categorize the extracted social content in five datasets depending on the deployment dates of the corresponding APIs (between 2005 and 2019). We compute the recall and precision of each dataset. Experiments reveal that the proposed approach can predict up to 87% influencer APIs with 71% precision.

3.2 The FAME Approach: An Overview

In this section, we give an overview of the proposed approach. We first introduce two scenarios to motivate our approach. Then, we describe FAME architecture for identifying and predicting influencer APIs.

3.2.1 Motivation

We describe two scenarios that illustrate the benefits and challenges of API influencer identification for consumers and providers. In both scenarios, we refer to two weather APIs: Aeris Weather (API\textsubscript{W}) and World Weather Online (API\textsubscript{WWO}).

Scenario 1 (API Consumers) - Let us consider a developer, Mary, looking for a weather API to use in a mashup. A search on ProgrammableWeb returns API\textsubscript{W} and API\textsubscript{WWO}. Since Mary has no prior experience programming with those APIs, she turns to developers in various programming communities to help her select the right one. Mary first looks at the features of API\textsubscript{W} and
Mary performs an extensive analysis of two APIs, \textit{API}_W and \textit{API}_{WWO}, on ProgrammableWeb (Table 3.1). Below is a summary of her findings. \textit{API}_W has more SDKs than \textit{API}_{WWO}. The two APIs have approximately the same number of articles published on the platform. \textit{API}_{WWO} has much more followers than \textit{API}_W. Because Mary is interested in mashup development. She learns that \textit{API}_{WWO} is used in more mashups than \textit{API}_W. However, she finds only 2 and 11 mashups for \textit{API}_W and \textit{API}_{WWO}, respectively. Mary then looks at the number of projects on GitHub that are relevant to the APIs. She noticed a much larger number of repositories related to \textit{API}_{WWO} than \textit{API}_W (39 vs. 6). Similarly, \textit{API}_{WWO} outperforms \textit{API}_W in terms of number of Wikis (428 vs. 3). However, less issues are reported by developers about \textit{API}_W than \textit{API}_{WWO} (29 vs. 310). Mary is overwhelmed by the number of API features published on each platform. She is confused about the features to consider in order to decide about the API to use. She becomes even more frustrated when she parses the long texts posted under the reported issues and commit comments on GitHub to get a better idea about her peers' opinions about the APIs.

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Features</th>
<th>\textit{AerisWeather API}</th>
<th>\textit{World Weather Online API}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProgramableWeb</td>
<td># of Articles</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td># of SDKs</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td># of Library</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td># of Mashup</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td># of Followers</td>
<td>126</td>
<td>271</td>
</tr>
<tr>
<td></td>
<td># of Changelog</td>
<td>21</td>
<td>19</td>
</tr>
<tr>
<td>GitHub</td>
<td># of Repositories</td>
<td>6</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td># of Commit</td>
<td>9</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td># of Issues</td>
<td>29</td>
<td>310</td>
</tr>
<tr>
<td></td>
<td># of Wikis</td>
<td>3</td>
<td>428</td>
</tr>
<tr>
<td>Google</td>
<td>By API name</td>
<td>14,000</td>
<td>42,400,000</td>
</tr>
<tr>
<td>Search Engine Index</td>
<td>By API EndPoint</td>
<td>1,750</td>
<td>569,000</td>
</tr>
<tr>
<td>Stack Overflow</td>
<td># of Posts</td>
<td>4</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 3.1 Motivating Scenarios
Scenario 2 (API Providers) - The provider of $API_W$, John, performs a Google search on $API_W$ and a competitor’s API, namely $API_{WWW}$. A search by API end-points returns more than 500,000 additional results for $API_{WWW}$. A second search by API names returned more than 42 million additional hits for $API_{WWW}$. John is concerned about the significant lack of popularity of his API compared to $API_{WWW}$. To increase the visibility of $API_W$ and promote its adoption, he looks at some of the features of $API_W$ and $API_{WWW}$ on ProgrammableWeb, GitHub, and StackOverflow (Table 3.1). The aim is to come up with an action plan to increase the adoption of $API_W$ by developers. The following questions need to be answered as part of John's action plan: how does he measure the influence of his API? which features are likely to have a higher impact on developers across programming communities to adopt $API_W$? how are the different API features related to each other? which features does he need to improve in order to enhance $API_W$ influence?

3.2.2 Architecture

The FAME architecture is composed of three modules (Figure 4.2): Unstructured Data Extractor (UDE), Structured Data Extractor (SDE), and the FAME Model. UDE extracts and analyzes unstructured (i.e., textual) API features from developer communities. Such features include commit comments in GitHub, posts in StackOverflow, and articles in ProgrammableWeb. UDE conducts sentiment analysis to quantify textual features as positive, neutral, or negative sentiment scores. Since the extraction and analysis of textual features is time consuming, UDE tasks are executed periodically and offline. SDE collects quantitative API features (e.g., number of repositories) from ProgrammableWeb, GitHub, and HackerNews. These features are extracted online (i.e., on demand) during the execution of an API influencer
identification or prediction request. Since SDE-extracted features are measured on different scales, data normalization techniques are applied to adjust those features to a common scale. Once all features are extracted, cleaned, and normalized, the FAME model aggregates those features to determine the API Influence Score (AIS) of each API. The calculated scores are used to cluster APIs into tiers: nano (least influential), micro, mid-tier, mega, and celebrity (most influential). The FAME model also uses non-negative-least-square regression to figure out significant features and associate weights to those features. Such weights are used to predict the evolution of AIS scores and tweak API features in order to enhance API influence across communities.
3.3 Influencer Identification and Prediction in FAME

In this section, we give details about the FAME approach for identifying and predicting influencer APIs. We first introduce the methods to extract both unstructured (i.e., textual) and structured features from programming community platforms (Section 3.3.1 and 3.3.2). Then, we describe the techniques for calculating API Influence Scores (AIS) and organizing APIs into influencer tiers based on AIS scores (Section 3.3.3). Finally, we present a linear regression-based model for predicting AIS scores (Section 3.3.4).

3.3.1 Unstructured Data Extractor (UDE)

UDE evaluates developers' sentiments toward APIs. It analyzes developers' feedback and computes scores of interest in APIs. UDE performs three major tasks: data collection, data pre-processing, and sentiment analysis. The data collection task crawls and collects developers' textual data from GitHub (commit comments), StackOverflow (posts), and ProgrammableWeb (articles). As each one of these platforms exports large volumes of data, we define data extraction patterns based on API names, endpoints, and topics to sort out API-related content and speed-up data collection. The data pre-processing task cleans textual data from irrelevant information such as code snippets and hyperlinks.

Sentiment analysis is the main UDE task. It evaluates developers' opinions toward APIs. We use the Stanford NLP (Natural Language Processing) Parser\(^\text{12}\). The parser adopts recursive neural nets, a deep learning technique, to figure out text polarity (positive, neutral, negative). It returns a sentiment score, \(sscore \in [-1, 1]\), along with text polarity. An \(sscore\) closer to -1 denotes a negative

\(^{12}\text{https://nlp.stanford.edu/}\)
sentiment. The sentiment is considered as positive if \( sscore \) is closer to 1. An \( sscore \) closer to 0 represents a neutral sentiment.

Some textual features may have higher user views than others. For instance, the sentiment of a post on an API with a large number of views should be given more importance than the sentiment of another post (on the same API) with a smaller number of views. Therefore, we associate a weight \( w_i \) for each textual feature \( f_i \). Each weight value \( w_i \) correlates to the number of views on \( f_i \). We normalize weights according to the following formula (1), where \( \text{view}_i \) and \( \text{Max}(\text{views}_\text{number}) \) represent the number of \( f_i \) views and maximum number of views in all features:

\[
\begin{align*}
  w_i &= \frac{\text{view}_i}{\text{Max}(\text{views}_\text{number})} \\
  (1)
\end{align*}
\]

Using the weight \( W_i \) and sentiment score \( sscore_i \) of a feature \( f_i \), we define the weighted sentiment as follows:

\[
\text{WeightedSentiment}_i = w_i \times sscore_i
\]

Finally, we define the overall sentiment for a given API \( j \) as the sum of the weighted sentiment of each extracted feature \( f_i \) divided by the total number of such features:

\[
\text{sentiment}(API_j) = \frac{\sum \text{WeightedSentiment}_i}{\text{TotalNumberOfExtractedFeatures}} \quad (2)
\]

3.3.2 Structured Data Extractor (SDE)

SDE collects structured API features from four programming community platforms: ProgrammableWeb, GitHub, StackOverflow, and HackerNews. By structured, we mean that the
features are well formatted in the platforms and ready to extract/use. We rely on Selenium\textsuperscript{13} WebDriver, a framework that automates Web data extraction. The framework allows feature extraction using predefined parsing rules. A rule contains the URLs to load data from and keywords describing APIs to filter data with. Data is parsed to extract features using DOM (Document Object Model)\textsuperscript{14}. Table 3.2 summarizes all API features extracted by UDE and SDE. Another important factor that helps assess the influence of an API is its spread over the internet. A well distributed/spread API is usually indexed on many search engines, which increases its visibility and eases its access. For example, Twitter API is accessible via multiple resources such as tutorials, documentations, and videos. This makes the API more likely to attract developers. We run two kinds of queries on Google search engine to measure the level of spread of an API. The first query counts the number of entries in the index that contain a given API name; the second query counts the number of entries containing a given API endpoint (Table 3.2).

\begin{footnotesize}
\begin{itemize}
\item https://www.seleniumhq.org/projects/webdriver/
\item https://www.w3.org/TR/WD-DOM/
\end{itemize}
\end{footnotesize}
Table 3.2 API Features in Developer Communities

The features retrieved by SDE are returned on different scales. For example, the number of issues reported for an API on GitHub may reach several hundreds; the number of SDKs available for an API on ProgrammableWeb is typically a one or two-digit value; the number of users interested in an API on GitHub may go beyond thousands. To normalize features on a common scale, we use the following formula:

$$\hat{x} = \frac{(x_{max} - x_{min}) * (r_{max} - r_{min})}{(x_{max} - x_{min})} + r_{min}$$  \hspace{1cm} (3)
Where: \( \hat{x} \) refers to the normalized value; \( x_{\text{max}} \) and \( x_{\text{min}} \) refer to the feature maximum and minimum values, respectively; \( r_{\text{max}} \) and \( r_{\text{min}} \) refer to the maximum and minimum new range values, 1 and 0 for our case.

### 3.3.3 API Influencer Score (AIS)

We define a metric, called API Influence Score (AIS), to model the degree to which community members use an API to develop mashups and service-oriented applications. For that purpose, we use the number of mashups and repositories that adopt the API on ProgrammableWeb and GitHub, respectively. However, some developers may display negative experiences using an API. To capture developers' opinions, the AIS score includes the average weighted sentiment. As shown in the formula (4), the AIS score is calculated using three of the API features extracted from community platforms (Table 3.2). The remaining extracted features are used to predict the AIS score as shown in Section 3.3.4. The AIS score is formally defined as follows:

\[
AIS_{(i)} = \sum \#M_i + \#R_i + \text{Sentiment}(API_i) \quad (4)
\]

Where: \( \#M_i \) is the number of mashups that use \( API_i \) on ProgrammableWeb; \( \#R_i \) is the number of repositories that use \( API_i \) on GitHub; Sentiment \((API_i)\) is the overall sentiment on \( API_i \) as defined in Section 3.3.1.

Using the computed AIS scores, we define influencer tiers to categorize APIs according to their influence level. Figure 3.2.b shows the five tiers: Nano, Micro, Mid-Tier, Mega, and Celebrity. Figure 3.2.a depicts the distribution of all APIs across the five influence tiers. The Nano tier regroups the least influential APIs. APIs in this tier have a score below 0.015. This category has the highest proportion of APIs, with about 600 identified APIs. Examples of Nano APIs are
Blinksale, Plunker and MyWot. The Micro tier contains APIs with a score between 0.015 and 0.15. Hoiio Voice, Kiva and Songkick are examples of APIs in this category. Mid- Tier refers to APIs with a score in the [0:15; 0:5] range such as LinkedIn, Zillow and Evernote. Mega regroups APIs with a significant influence score (AIS ∈ [0:5; 1:5]), such as Flickr, Last.fm, and Reddit.

Celebrity represents APIs with the highest influence (AIS > 1.5). Celebrities appear in the highest number of mashups and repositories. They also subject to positive sentiments among developers. Examples of celebrities are Google Maps, Twitter and YouTube. Figure 3.2.a shows that this tier has the lowest proportion of APIs, with about 20 identified APIs.

3.3.4 Influence Score Prediction

We compute the AIS score of an API using three features: number of mashups, number of repositories, and overall developers’ sentiment. However, it is difficult for API providers to have direct control on those features to improve the adoption of their APIs by developers. To help API
providers enhance the influence of their APIs, we conduct a statistical study to identify the most relevant API features that correlate the most to AIS scores. Once API providers understand which of the remaining features (other than number of mashups, repositories, and sentiment) impact the AIS score, they can come-up with a strategy to boost-up the influence of their APIs. We use Non-Negative Least Squares (NNLS) regression [81] to learn a weight value for each API feature. NNLS assigns weights to features according to their correlation degree to AIS scores. The most relevant features are given high coefficients, while non relevant ones are given negative coefficients.

NNLS replaces negative coefficients by 0. This will automatically get rid of non relevant features from the model. Figure 3.3 summarizes the coefficients assigned to each API feature. Features with the biggest coefficient values have the highest impact on AIS scores. For instance, the number of articles in ProgrammableWeb is strongly related to the AIS score. This shows that more articles
published in the developer community may increase API influence. Figure 3.3 also states that StackOverflow features have little impact on AIS scores.

The next step is to define AIS prediction models. These models are useful to assign initial influence scores for newly deployed APIs, hence dealing with the traditional cold start problem. They also assist API providers in predicting the evolution of their API scores. We introduce three prediction models (Table 3.3). To evaluate and compare the models, we calculate the adjusted R-squared\textsuperscript{82} and Akaike Information Criterion (AIC)\textsuperscript{83}. The adjusted R-squared estimates the variance between predicted and real scores. AIC measures the goodness of the fit for the model. The model with the smallest AIC value and highest adjusted R-square is selected as the best-fitting model.

Table 3.3 summarizes our three prediction models. Model 1 uses all extracted features to predict the AIS score. It has a low adjusted R-squared value: 0.5788. Hence the model does not fit the trend perfectly. This is because AIS scores depend on developers' sentiments, which are hard to predict. To deal with this issue, we introduce the adjusted AIS score ($\text{AIS}_{\text{adjusted}}$). $\text{AIS}_{\text{adjusted}}$ is a variant of the original AIS score that eliminates developers' sentiments. The following formula (5) computes API\textsubscript{i}'s adjusted AIS score.

$$\text{AIS}_{\text{adjusted}(i)} = \sum \#M_i + \#R_i$$

(5)

The second and third models predict the AIS adjusted scores. Model 3 uses all extracted features. Model 2 omits the features extracted from StackOverflow since our study shows that StackOverflow has little impact on API adoption across communities (Figure 3.3). Both models display high adjusted R-squared: 0.77 for Model 2 and 0.78 for Model 3. The models also have low AIC values: 1346.347 for Model 1 and 1323.208 for Model 2. This makes both models suitable
for predicting the adjusted AIS score, with a slight advantage to Model 3 as it uses more API features than Model 2.

### 3.4 Experiments

The goal of our experiments is to assess FAME’s ability to accurately predict influencer APIs. We evaluate the second and third prediction models (Table 3.3) using five independent datasets. The datasets regroup APIs deployed during five different periods between 2005 to 2019. For each API, we compute the adjusted AIS score and use the models (2 and 3) to predict that score. We then compute the recall and precision for each dataset using both models. The recall refers to the fraction of influencer APIs that are correctly identified within each dataset. It is the number of influencers that are successfully predicted divided by the number of all APIs that are identified as influencers.

<table>
<thead>
<tr>
<th>Prediction Model</th>
<th>Available Features</th>
<th>Number of Selected Features</th>
<th>Selected Features</th>
<th>Adjusted R-squared</th>
<th>AIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model 1</td>
<td>AIS</td>
<td>18 (SDA + UDA)</td>
<td>12 Features</td>
<td>0.57</td>
<td>4400.073</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>ProgrammableWeb</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Articles, #Libraries, #Followers and ChangeLog</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Google Search Engine</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Spread Score by EndPoint</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Hacker News</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Comments</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>GitHub</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Commits, #Users, #Wiki and #Topic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>StackOverflow</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Post and Ave Post View</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 2</td>
<td>AIS-adjusted</td>
<td>15 (SDA)</td>
<td>7 Features</td>
<td>0.77</td>
<td>1346.347</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>ProgrammableWeb</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Articles and #Followers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Google Search Engine</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Spread Score by Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>GitHub</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Commits, #Users, #Wiki and #Topic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 3</td>
<td>AIS-adjusted</td>
<td>18 (SDA + UDA)</td>
<td>9 Features</td>
<td>0.78</td>
<td>1323.208</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>ProgrammableWeb</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Articles and #Followers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>Google Search Engine</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Spread Score by Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>GitHub</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Commits, #Users, #Wiki and #Topic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• <strong>StackOverflow</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• #Post and Max(Post) View</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3 Prediction Models
It can be also seen as the percentage of influencer APIs that are successfully predicted. Figure 3.4.a shows that up to 86% and 88% of influencers are successfully recalled (i.e., predicted) by Model 2 and Model 3, respectively. Both models have a stable recall but leveraging more features in Model 3 allows a better prediction than Model 2. The precision checks the accuracy with which scores are predicted for APIs. It is the number of precisely predicted influencer APIs divided by all recalled APIs. It can be seen as the percentage of precisely recalled influencers. If the weight difference is less than a threshold value, the influencer is assumed to be precisely identified. We used 0.03 as a threshold; this value represents the average of the difference between the predicted and computed scores. Figure 3.4.b shows that both models identify influencer APIs with up to 71% precision.

3.5 Conclusion

We propose FAME (inFluencer Apis in developer coMmunitiEs), a novel approach for the identification and prediction of influencer APIs in service-oriented environments. To the best of
our knowledge, FAME is the first influencer model that treats APIs as first-class citizens. We define influence as the degree to which an API is used in mashups and service-oriented applications. We extract and analyze several structured and unstructured features from various programming communities. We use the Stanford NLP parser to perform sentiment analysis and evaluate developers’ opinions towards using APIs. Such opinions are expressed through posts in StackOverflow, commit comments in GitHub, and articles in ProgrammableWeb. We aggregate API features to compute influence scores and cluster APIs into influencer tiers according to those scores. We use Non-Negative Least Square (NNLS) regression to identify the most significant API features and predict the evolution of influence scores for newly deployed and existing APIs. Finally, we conduct extensive experiments on real-world and large datasets extracted from multiple programming community platforms. Experiments reveal that the proposed approach predicts up to 87% influencer APIs.
CHAPTER 4
TEAM: Leveraging Experiences from Developer Communities for API Recommendation

4.1 Introduction

In service-oriented environment, Web service composition allows dynamic integration of Web services. The performance of the involved services determines the overall performance of the composed service [84]. Hence, it is crucial to pick high-quality services to form service composition. A large body of research proposed quality of service (QoS) as a metric for composition selection (e.g., [85], [86], [87]). However, very few contributions looked at the quality of experience (QoE) from developers as a metric for selection. Understanding and measuring developers’ QoE is a vital and challenging task especially with the substantial number of available APIs that provide similar functionalities. For instance, the popular ProgrammableWeb API directory currently shows more than one thousand APIs that provide payment service (as of October 2021). Another important factor is that the technological advancements over the last decades in terms of software and hardware change the way systems are designed and perform [88]. This is led to a shift from QoS to QoE because for different consumers, the same level of QoS does not guarantee the same perceived level of quality. However, according to state of art [51] quality of service and quality of experience have a strong positive correlation. As a result, QoE can be used to select services, particularly when QoS data is unavailable.
Furthermore, to evaluate the service accurately, we need to obtain the experience from the developers' communities, not end-users who do not have any professional experience with services. For instance, if we are looking at regular users' experience of the most famous Mapping services which are Google Maps and Bing Maps. Both services have mobile Apps and rely heavily on standard API [89]. Based on the Apple store2 rating which based on regular users' QoE, in October 2021 both Apps have a similar rating of 4.7 out of 5 regardless of the number of reviewers. Because both services have great functionality and features from the normal user's point of view. However, from the developer's perspective, Google Maps tend to have more repositories, libraries, and questions/answers in the developer's communities. Indicating that developers are more engaged with Google Maps than Bing Maps. Moreover, developers in the programmers' communities have been supporting each other in the area of contributing and enhancing the quality of experience of the service. For example, in the case of facing any technical issues with a selected service, assuming that developer Bob is responsible for deciding which services to pick for building his application. After exploring the available services, he decided to choose Google Maps API1 or Bing Maps API2 as they have the same functionality. Therefore, he turned to developers' communities to ask for some technical details about the services. Since he needs to build the application in a short period of time, he will choose either one once he gets the clarification for his concern. To that end, we consider the waiting time to get an answer as a metric (which is the response time of the service) reflecting the quality of experience of the service from the developer's communities.
In this chapter, we introduce TEAM (Quality of Experience-based Api recommendation) [18], an approach and tool that leverages prior API development experiences to recommend APIs. We analyze StackOverflow, GitHub and other platforms using programmers' activities to quantify the QoE toward a service (Fig 4.1). To the best of our knowledge, this is the first work to measure QoE for Web service based on developer's communities. The main contributions of this paper are summarized below:

- We propose a QoE model that utilizes more than twenty features from various programming communities. Those features capture non-technical, technical and educational information about Web services. We study and analyze those features to turn it to QoE metrics. Then, we propose four major QoE attributes from developers’ communities.

- We train three different types of classifiers (RF, SVM, and NN), to predict the API usage by using their quality features. Then we compare the performance of each classifier. Next
we interpretation the model to determining quality features importance and identifying the most significant explanatory factors related to API usage.

- We conduct experiments on six real-world programming platforms: GitHub, StackOverflow, ProgrammableWeb, HackerNews, Google Trend and YouTube.

### 4.2 Motivation

Back to the aforementioned example of developer Bob. Let’s consider both services have the same response time from the developers' communities, for instance, GitHub and StackOverflow. Now Bob needs to explore any available resources that would help. He starts by surfing the Internet and collect data about both APIs (Table 4.1). Below is a summary of his findings. After searching, he found huge valuable tutorials on YouTube, the number of supported programming languages on GitHub, and the developer interactions (Q/A) on StackOverflow.

<table>
<thead>
<tr>
<th>Availability</th>
<th>Google Maps API</th>
<th>Bing Maps API</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>YouTube</strong></td>
<td>- Avg View: 37,372 Views</td>
<td>- 7,585 Views</td>
</tr>
<tr>
<td></td>
<td>- Avg Like: 288 Liked</td>
<td>- 47 Liked</td>
</tr>
<tr>
<td></td>
<td>- Avg Dislike: 14 Disliked</td>
<td>- 3 Disliked</td>
</tr>
<tr>
<td></td>
<td>- Tutorial: 19+, 29+, 9+</td>
<td>- Tutorial: 687</td>
</tr>
<tr>
<td></td>
<td>- Java: 1941</td>
<td>- 6+</td>
</tr>
<tr>
<td></td>
<td>- Python: 572</td>
<td>- 0</td>
</tr>
<tr>
<td></td>
<td>- C#: 172</td>
<td>- 10+</td>
</tr>
<tr>
<td><strong>GitHub</strong></td>
<td>- Java: 24</td>
<td>- 47</td>
</tr>
<tr>
<td></td>
<td>- Python: 24</td>
<td>- 124</td>
</tr>
<tr>
<td></td>
<td>- C#: 24</td>
<td>- 124</td>
</tr>
<tr>
<td><strong>StackOverflow</strong></td>
<td>- # Questions: 26,344</td>
<td>- 936</td>
</tr>
<tr>
<td></td>
<td>- # Answers: 19,751</td>
<td>- 1,270</td>
</tr>
<tr>
<td></td>
<td>- # Corrected Answers: 12,986</td>
<td>- 495</td>
</tr>
<tr>
<td></td>
<td>- Avg Answer Score: 3.2</td>
<td>- 1.3</td>
</tr>
<tr>
<td></td>
<td>- Corrected ratio: % 49.29</td>
<td>- % 52.88</td>
</tr>
<tr>
<td></td>
<td>- All Answers ratio: % 74.97</td>
<td>- % 135.68</td>
</tr>
<tr>
<td></td>
<td>- Ave response: 6:58:30</td>
<td>- 7:18:12</td>
</tr>
</tbody>
</table>

| Table 4.1 Google Maps & Bing Maps Analysis |
However, the result is confusing because some of the features of the APIs' have similar values. For example, if we look at the developers' interactions (Q/A) on StackOverflow, we will see that API1 has more questions and answers than API2. But API2 has more correct answer ratio. Moreover, API1 has more repositories and video tutorials than API2. As a result, he reached a level of uncertainty regarding what features to be selected for deciding the best API to pick. Therefore, providing a QoE model from developers' communities would help him to get a better idea about the developers experience of specific API.

4.3 TEAM Architecture

The TEAM (quality of Experience-based Api recoMmendation) architecture is composed of three modules (Fig 4.2): Data Extraction and Categorization (DEC), Features Engineering (FE), and the TEAM Model. DEC extracts raw data related to APIs from developers' communities and analyzes it for the prepossessing operations. Such as removing unrelated data and normalizing the key data. Then, we categorized the data into two main components, API usage, and developers' activities. These tasks are executed periodically and offline. FE will transform the cleaned and normalized data into features that will be used in TEAM model. The TEAM model aggregates all features and classifies the API usage based on the QoE features. The result of this classification will determine the importance of QoE features that affect developers' usage. In addition, the classification models will be used to tweak the QoE features in order to advance API usage in developers' communities.

Moreover, TEAM architecture aims to model the QoE of the services to evaluate the service quality by using what is usually used in QoS metrics (response time availability, reliability, and security). We predefined QoS metrics into new QoE metrics. Those metrics will be
concentrated on the developers' communities to quantifying the four metrics. The developers' communities such as GitHub and StackOverflow are the main source to extract the developers' experience. However, other platforms such as YouTube can be seen as sim-developers' communities. APIs providers use this platform to upload tutorial videos about the features they provide and usability guidelines of their APIs. For example, official Google developers account which has more than 2 million subscribers posted many tutorials about Google services such as Google Maps API. Those metrics can not be captured straightforward but they need to be analyzed to form the QoE metrics. We will explain each metrics in the QoE dimension section 4.4. Furthermore, we implemented a tool that allows the developers to examine the QoE to find the best service that can meet their requirements.

![Figure 4.2 QoE Architecture](image_url)
4.3.1 Data Extraction Categorization (DEC)

Data related to APIs are scattered in different formats such as programming forums, technical news, and educational tutorials. In order to measure the quality of experience related to APIs, we need to gather scattered data from multiple platforms. DEC collects and forms the developers' experience or (activities) and computes developers' usage of particular API based on the public data in programmers' communities. The main source of information is ProgrammableWeb. We crawl programmableWeb.com and obtained the key information about APIs (API names, Articles, Sdks, Libraries and Followers). This information obtained by using the Scrapy framework. From this information, we obtain a list of API names that is used to filter developers' communities in order to extract relevant information about listed API. For example, in GitHub we use Node.js to crawl GitHub repositories, we captured repositories information such as the number of watches, stars, and forks. Moreover, we obtained the issues information like created and closed date and time.

Besides, StackOverflow offers their data publicly and can be stored in XML format from Stack Exchange. We focused on questions and answers and relevant information for instance the time taken to 'close' questions on StackOverflow and the textual content in those posts. Furthermore, as a technology news platform, we used HackerNews and we employ predefined rules which include the URLs for loading the required data thus the number of comments and stories related to APIs. However, other platforms, to give an example, YouTube and Google Trends can be seen as sim developers' communities. We use YouTube API for searching tutorials about APIs and extract some key information about them such as the number of views,
subscriptions, likes, dislikes, and comments. Also, we capture the popularity of APIs on Google search engine by employing *gtrendsR*\(^\text{15}\) package to retrieve the number of hits from Google Trend.

The extracted data can be categorized into two main components API usage and developers' activities. For API usage we define a metric to model the degree to which developers use an API in their mashup or service-oriented applications. To explore the API usage in the communities of developers, we propose three metrics that can be obtained publicly.

1. The sum of repositories \(R\) in GitHub related to API \(i\).
   \[
   API_{Usage(i)} = \sum \#R(i) \quad (1)
   \]
2. The sum of mashup \(M\) in ProgrammableWeb related to API \(i\).
   \[
   API_{Usage(i)} = \sum \#M(i) \quad (2)
   \]
3. The sum of followers \(F\) in ProgrammableWeb related to API \(i\).
   \[
   API_{Usage(i)} = \sum \#F(i) \quad (3)
   \]

We analyzed all metrics by using statistical tests of 2520 APIs. Table 4.2 shows the detailed statistics of each metric. Metric 1 & 3 shows a large standard deviation (SD) which indicates that the data points are far from the mean. However, Metric 2 notes a smaller standard deviation which means that they are grouped closely around the mean, and it has zero median because more than half of the values are zeros. We utilized the Shapiro-Wilk test to analyze the normality assumption. Based on this test, all metrics distributions can be assumed as not normally distributed where p-value < 0.05. In addition, we applied Skewness and Kurtosis tests to get insights into the shape of the distribution. The skewness test for Metrics 1, 2 and 3 are 10.26, 34.43 and 15.48 respectively. All metrics show a positive skewness, but the distribution of Metric 2 has highly skewed. The

\(^{15}\) https://scrapy.org/
Kurtosis test is 142.90, 1402.83 and 324.57 for Metric 1, Metric 2, and Metric 3 respectively. This indicates that all distributions have heavier tails which are considered as a Leptokurtic distribution. However, we convert all metrics into multi-classes with three labels (Low, Medium and High). We labeled each class based on the average of the API usage for each proposed metrics. The detailed class labels are illustrated in Table 4.3.

<table>
<thead>
<tr>
<th>API Usage Metric</th>
<th>Mean</th>
<th>SD</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric 1</td>
<td>282.43</td>
<td>1251.82</td>
<td>26</td>
</tr>
<tr>
<td>Metric 2</td>
<td>4.94</td>
<td>59.57</td>
<td>0</td>
</tr>
<tr>
<td>Metric 3</td>
<td>33.13</td>
<td>138.38</td>
<td>7.57</td>
</tr>
</tbody>
</table>

Table 4.2 Descriptive Statistics of API Usage Metric

Accordingly, we note the distribution of each class as it is shown in Figure 4.3. Metric 1 & 3 reveals the majority of API usage recognized as medium and high usage. However, Metric 2 shows that about 80% of API usage is recognized as low usage. Accordingly, we face a class imbalance problem where the classes are not distributed equally. [90] have shown that for several classification algorithms, balanced data provides increased overall classification performance.
compared to imbalanced data. We utilized the synthetic minority oversampling technique (SMOTE) [91] to rebalance the proposed classes. The SMOTE algorithm creates artificial data based on the nearest neighbors of the minority class instances. This process balance the training dataset to build TEAM model, however, the test dataset was never modified to avoid any bias.

<table>
<thead>
<tr>
<th>Class Labels</th>
<th>Metric 1</th>
<th>Metric 2</th>
<th>Metric 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>=1</td>
<td>=0</td>
<td>=0</td>
</tr>
<tr>
<td>Medium</td>
<td>&gt;1 &amp; &lt;=282</td>
<td>&gt;0 &amp; &lt;=5</td>
<td>&gt;0 &amp; &lt;=33</td>
</tr>
<tr>
<td>High</td>
<td>&gt;282</td>
<td>&gt;5</td>
<td>&gt;33</td>
</tr>
</tbody>
</table>

Table 4.3 Class Label Setting of API Usage Metric

4.3.2 Features Engineering (FE)

Features engineering is the process of adding new features that are calculated based on the existing ones [92]. This a common practice technique in building supervised learning models. Since the representation of the features has a significant impact on machine learning performance, the QoE features cover different aspects such as the quality of reply or the type of developers involvement in StackOverflow or GitHub. We want to know how the model react to each feature. Therefore, we used two techniques to advance the extracted data from DCE

**Expand existing features:** this task aims to construct new features from existing features. For example, we labeled the time needed to answer a question in StackOverflow or a closed issue in GitHub as response time RT. However, to understand the interactions of existing features we extend RT into different features thus \( RT_{weight}, RT_{quality}, \) and \( RT_{sentiment} \). Those features are
calculated by computing the ratio of RT with other factors including the sentiment score of the
text.

Imputing missing features: QoE parameters are scattered across multiple platforms and
cannot be extracted from one single source. In addition, not all APIs are exposed in all platforms.
Therefore, we report only 87 APIs appeared in all platforms and 2635 APIs have been reported
with missing data. Removing a lot of missing values in the data might result in a data reduction
and substantial bias throughout the inference process [93]. Since we are looking to build a model
that can predict QoE accurately, we would need to have as many samples as possible. Therefore,
we applied two imputation techniques that can fill the missing data correctly. We employ two well-
known imputation methods:

1. We applied k-nearest neighbor (KNN) imputation to fill missing values with a
weighted mean of the 10 nearest neighbors by leveraging R package DmWR. We
describe this dataset as knn - dataset.

2. We applied Multivariate imputation by chained equations (MICE) based on
random forest (RT) by utilizing R package MICE. This method uses a machine
learning technique which can support non-linearities and ensure parameter estimations
were less biased [94]. We describe this dataset as mice - dataset.

4.4 QoE Dimensions

We proposed four metrics to evaluate the QoE which are response time, availability, reliability
and security. (Table 4.4) summarizes all QoE features.
Table 4.4 QoE Features in Developer Communities

<table>
<thead>
<tr>
<th>QoE parameters</th>
<th>Definition</th>
<th>Metrics</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Response Time</strong></td>
<td>The time to get response from the communities</td>
<td>1. $RT_{first}$: The response time to get the first response.</td>
<td><strong>GitHub</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. $RT_{close}$: The response time to close/issue in GitHub and answer/question in Stack Overflow</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. $RT_{involvement}$: The response time multiple by the type of involvement (Original, Expert...).</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. $RT_{sentiment}$: The response time multiple by the polarity (Sentiment Analysis) of the post.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. $RT_{prop}$: The multiple # of Star in Stackoverflow and like in Github. This is to test how effective comments</td>
<td></td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>The Resources available for developers in the community</td>
<td>6. # of available Articles in Programmable Web</td>
<td><strong>GitHub</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7. # of available SDKs in Programmable Web</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8. # of available Libraries in Programmable Web</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9. # of available Followers in Programmable Web</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10. Avg stories in Hacker News.</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>11. Avg comment in Hacker News.</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>12. Avg view of tutorials in YouTube.</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13. Avg subscribe of tutorials in YouTube.</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>14. Avg like of tutorials in YouTube.</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>15. Avg dislike of tutorials in YouTube.</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16. Avg number of comments in YouTube tutorials.</td>
<td><strong>Stack Overflow</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>17. Avg( watch ) for repositories in GitHub.</td>
<td><strong>GitHub</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>18. Avg( star ) for repositories in GitHub.</td>
<td><strong>GitHub</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>19. Avg( fork ) for repositories in GitHub.</td>
<td><strong>GitHub</strong></td>
</tr>
<tr>
<td><strong>Reliability</strong></td>
<td>Measure the consistency and frequency of the API in the community</td>
<td>20. Avg hits in the last 5 years from Google Trend</td>
<td><strong>Programmable Web</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>21. Avg change log in programmable Web</td>
<td><strong>Google Trends</strong></td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td>Analyze the security subject in the community</td>
<td>22. Analyze posts that discuss about security subject using Latent Semantic Analysis (LSA) and sentiment analysis of those post using our formula.</td>
<td><strong>GitHub</strong></td>
</tr>
</tbody>
</table>

**Response time** - developers often turn to programming communities seeking help or clarification to solve intricate problems, for instance StackOverflow. However, the time it took to get a response from developer communities will accelerate the development process. In addition, the response time can reflect the developers’ satisfaction and engagement [95]. Many studies have been proven
that the key of popularity of Q/A platforms such as StackOverflow is the how fast (time laps) to get first answer to question. In such crowdsourcing platform a reputation used as a reward mechanism, for this reason, members try their best to gain good reputation. However, considerable research proven that the developers need to clarify their questions and use proper tag in their posts to get fast response [96]. The time frame of the first response and the time to a closed or an answered post are important to solve an urgent issue. Furthermore, the level of experts and skills of developers that are involved in the response play important role in the quality of the answer. Although, [97] suggests that experts tend to collaborate more than non-expert and their answers are more likely to be accepted or up-voted. Therefore, considering the response time with the level of expertise shows the quality of the response. Therefore, we need to distinguish the level of expertise for each response. Hence, the response is highly likely to be helpful from experts than beginners. For that reason, we propose four categories (Beginners, Skilled, Intermediate and Experts). Since the main sources in our data is the (StackOverflow and GitHub) where they have different data structure. The StackOverflow provides a reputation system as rewards to motivate their users to answer more question. However, to reduce their bias of their reputation system towards the easy questions, they offer various badges (Gold, Silver and Bronze) to encourage the developers who answer difficult questions. however, in GitHub we consider the users activities in the platform such as the number of followers, following, stars, and public repositories. From our dataset we find that 84,579 users in StackOverflow and 20,931 users in GitHub post comments related to APIs. From StackOverflow, their reputation ranges from 1 to 1,195,958 with an average of 7,072. However, the majority of developers have low number of badges from 0 to 35 and the maximum number are 768, 8,492 and 8,783 for gold, silver, and bronze badges respectively.

---

16 https://stackoverow.com/help/whats-reputation
We applied unsupervised clustering with K-means clustering to identify the level of expertise based on users features as shown in Table 4.5. The result from the clustering as shown in Fig 4.4 shows that the largest group consists of 79,786 and 10,903 users in StackOverflow and GitHub, who have low reputation and number of repositories. We name this group as beginners. Similarly, we name the Skilled, Intermediate group based on their activities. However, the group with the highest activities is by far have the smallest size with 6 and 3 users in StackOverflow and GitHub, we call this group as expert or API owner.

<table>
<thead>
<tr>
<th>Source</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>GitHub</td>
<td>1- Number of Followers</td>
</tr>
<tr>
<td></td>
<td>2- Number of Following</td>
</tr>
<tr>
<td></td>
<td>3- Number of Stars</td>
</tr>
<tr>
<td></td>
<td>4- Number of Public Repositories</td>
</tr>
<tr>
<td>StackOverflow</td>
<td>1- User Reputation Score</td>
</tr>
<tr>
<td></td>
<td>2- Number of Gold Badges</td>
</tr>
<tr>
<td></td>
<td>3- Number of Silver Badges</td>
</tr>
<tr>
<td></td>
<td>4- Number of Bronze Badges</td>
</tr>
</tbody>
</table>

Table 4.5 Developer Features

Figure 4.4 StackOverflow and GitHub Users K-Mean Clustering
We propose five metrics to analyze the response time. All metrics are calculated by (minutes).

- $\overline{RT_{first}}$: The average response time to get the first answer to a question in StackOverflow or response to an issue in GitHub. However, from our data set we find that some post take from 1 minute to about 5 years to get the first response.

- $\overline{RT_{close}}$: The average response time to answer a question in StackOverflow or to close an issue in GitHub.

- $\overline{RT_{weighted}}$: The average response time $RT_{close}$ divided by the type of developer’s involvement (Beginners, Skilled, Intermediate, and Experts). Where: $W$ is a value from 1 to 4, where 1 for Beginners and 4 for Experts.

\[
\overline{RT_{weighted}}(i) = \frac{RT_{close}}{W} \quad (4)
\]

- $\overline{RT_{quality}}$: The average response time $RT_{close}$ divided by the number of answer score $S$ in StackOverflow or number of contribution score $C$ of the user in GitHub. The answer score shows the quality of the accepted answer based on the StackOverflow society. The contribution score indicates how active the user is in the GitHub community. [98] Finds that higher levels of developer’s contributions are those who ensure that the project will succeed.

\[
\overline{RT_{quality}}(i) = \frac{RT_{close}}{#S \text{ or } C} \quad (5)
\]

- $\overline{RT_{sentiment}}$: The average response time $RT_{close}$ divided by the sentiment value of the response. Where: SentimentScore is the polarity value from 1 to 3 where 1 for negative, 2 for neutral and 3 for positive polarity.
\[ RT_{sentiment}(i) = \frac{RT_{close}}{SentimentScore} \] (6)

The purpose of dividing the \( RT_{close} \) by different dimensions (type of involvement, number of star or contribution and sentiment) is to identify the significant factor that can render the response time, we will discuss that more in the experiment section.

**Availability** - Developers spend most of their time investigating online documentation by looking at the official API documentation, Q&A websites, tutorials, etc. For example, the official LinkedIn API documentation guides the developers who need support to use StackOverflow by tagging the question with `linkedin` tag `#linkedin`. Therefore, the resource data spread across various platforms. Providing multiple documentation resources to developers can guide them to learn more about API functionality and solve other issues in the development phase. In this quality metric, we determined the availability as the available resources for developers to build their projects smoothly.

Another important aspect that helps to determine an API’s availability is the ability to spread across different platforms. For example, when developers are addressing complicated challenges, they turn to programming groups (e.g., GitHub) to exchange practices, expertise, skills, and brainpower. Therefore, the number of watches, star and fork for repositories indicate how programmers take advantage of using those repositories. For instance, one of the popular repositories for Twitter API is "gsferik/twitter" which has 135 watch, 4259 stars and 1195 forks. This indicates how many developers use this repository and it can reflect the availability of this repository.

Moreover, other API features can be found in ProgrammableWeb such as (Articles,
SDKs, Libraries and Followers). Besides, a news articles in HackerNews has an important role to expose the APIs functionality. Additionally, APIs video tutorials are one documentation resource that has seen a rapid increase in developer's communities [99]. For instance, Google Maps API has an official channel on YouTube that contains API tutorials and explains other API features. However, Bing Maps API does not have an official channel about the API, but many developers who like this API still post tutorials about it. This could explain why developers use Google Maps APIs more than Bing Maps API. Therefore, we crawl the metadata of YouTube videos that contain the name of the API and we analyzed the tutorials to find the engagement features of the videos like the number of views, likes, dislikes, comments, and subscriptions.

**Reliability** - Developers are usually looking for reliable APIs that have many releases and updates published to solve issues, enhance performance or promote new functionalities. This support will make APIs more reliable in the communities. Accordingly, measuring the consistency and frequency of active API in the communities will interpret if the API performance is consistent or not. Therefore, we propose two factors to represent the API reliability in developer communities as below:

- **Google Trend Information**: Analyzes the popularity of search queries in Google search engine. Hence, we leverage `gtrendsR` library [100] to retrieve search information particularly trends or number of hits in the past 5 years. Afterwards, we will take the average of the hits as an indication of the consistency and frequency of search-related to API.

- **Change logs (LifeCycle)**: The number of versions and updates that have been reported about APIs. This information can be found by utilizing the monitoring services like change
log. For that reason, we capture the change-log information from ProgrammableWeb to obtain the update activities.

**Security** - Security was always a popular and important topic in software quality. Therefore, most of developers discuss and interact with each other to expose their security-related subject. The programming communities such as GitHub and StackOverflow have numerous discussions about security-related and others topics. Accordingly, we filtered developers posts that contain Web service. In GitHub we look at the posted issues and comments, where in StackOverflow we consider the questions and answers. We clean and pre-process those posts to identify security-related post. Then, we analyze developers' comments and assign API Security Satisfaction Score (API\textsubscript{SeS}\textsubscript{Score}) for each APIs. For that reason, finding API\textsubscript{SeS}\textsubscript{Score} consists of two stages:

**Identifying Security-Related Posts:** we have 3,999,055 posts in StackOverflow, we filtered those posts by security-related keywords. We leverage two strategies to filter
security related posts. Filtering posts by using predefined keywords and applying text mining technique to extract new security-related keywords. First, we borrowed 127 predefined security related keywords from a previous study [101] that has been extracted from GitHub. The keywords were reviewed and corrected manually. We filtered the comments by using these keywords and we found 503,015 posts in StackOverflow. Afterwards, we applied extra analysis to find if service-oriented posts use different keywords for security. Therefore, we employed a topic modeling technique which is a method for unsupervised classification of textual documents. Latent Dirichlet Allocation (LDA) is a common model for topic modeling. We applied LDA in our dataset to obtain more topics (keywords) of security-related posts. In the LDA model, a document is the text body of the post. We reviewed the extracted result from the top 10 keyword terms from the top 30 topics manually. We select 10 keywords (token, session, override, database, root, redirect, thread, automat, client and machine). Then, we filter the comments with the new keywords list, and we found 877,013 posts related to security. Finally, we applied sentiment analysis to the security-related post to find API sentiment security score $API_{sent-sec}(i)$ for each API $i$.

- **Calculating API security satisfaction score:** $API_{SeScore}$ Identifying security satisfaction score of APIs is crucial for eliminating security issues during software development. For that reason, we define two main factors to identify $API_{SeScore}$. First, the developers' opinion about security-related posts, for that we employ the sentiment analysis for identified comments $API_{sent-sec}(i)$. Second, the reported security issues repository information. For that reason, we leverage the *Common Vulnerabilities and Exposures CVE*
the world's largest and most authoritative vulnerability repository. To that end, we define

API\textsubscript{SeScore} as shown in the formula below:

\[
API\textsubscript{SeScore}(i) = \frac{Ave(\text{API}_{\text{sent-sec}})}{\text{Max}(\text{API}_{\text{sent-sec}})} \div \frac{\#\text{CVE}(i)}{\text{Max}(\#\text{CVE}(i))} \tag{7}
\]

Where \text{API}_{\text{sent-sec}} denote the API \(i\) sentiment security score and \#\text{CVE} denote the number of reported CVE entries for API \(i\). This formula considers a positive relation with \text{API}_{\text{sent-sec}} and a negative relation with the \#\text{CVE}. Accordingly, a higher API\textsubscript{SeScore} indicates more secure API. However, in case if there is no CVE reported we consider the divisor of formula 7 to 1.

4.5 TEAM QoE Model

In this section, we give details about the TEAM approach for identifying and classifying API usage based on the QoE features. We define API usage to show the degree to which community members use an API to develop service-oriented applications as we defined in the formulas (1, 2, and 3). Then, we labeled those metric into three classes as we described in 4.3.1. We utilize three different classification techniques (RF, SVM, and NN) to build the TEAM model. First, we describe the fundamental of each classification technique. Then the process to conduct feature analyses on the best classification model.

\[\text{https://cve.mitre.org/}\]
Random Forest (RF) - The RF is a supervised machine learning algorithm that combines multiple individual decision trees (DTs). The algorithm is used to solve regression and classification problems. The structure of RF as a classification model is shown in Fig 4.6. Let's assume that the input for RF algorithm is a training data TD = \((X_1, Y_1), (X_2, Y_2), \ldots, (X_i, Y_i)\) contains \(i\) observations, \(X_1\) denotes the independent variable vector owing \(M\) features as \(X_i = (X_{i1}, X_{i2}, \ldots, X_{iM})\), \(Y_i\) denotes the dependent variable. The RF algorithm starts with bootstrapping or bagging, which is fit to an independent bootstrap sample from the DT. Then, for each bootstrap sample BS, a separate decision tree is built. After that, each decision tree will generate an output. The next step is to take those outputs and calculate the majority decision or average them to get the final outcome. In this study, the RN was implemented using random forest package in R [102].
Support Vector Machine (SVM) - The Support Vector Machine, or SVM, is a supervised learning approach that can be used to solve classification and regression issues. The basic principle behind SVM is that it constructs a line or hyperplane that divides data into classes [103].

The linear SVM assumes that multidimensional data in the input space may be separated linearly as shown in Figure (4.7). There are three lines in the SVM model. The first one is \( w \cdot x - b = 0 \), which is the marginal line or margin. The lines \( w \cdot x - b = 1 \) and \( w \cdot x - b = -1 \) denote the position of the closest data points of both the classes. The circles lying on the marginal line called the support vectors. Moreover, SVM are based on a collection of mathematical functions known as the kernel. The kernel's goal is to take data and turn it into the needed format. For that reason, we use different types of kernel functions to maximize the classification accuracy of our SVM such as (linear, nonlinear and polynomial).
Neural Networks (NN) - The ANNs are inspired by biological neural networks found in animal brains. ANNs are well known for classification and regression problems. The difference between classification and regression is that classification outputs a prediction for classes or categories, while regression provides numeric values. Moreover, ANNs form a collection of basic units called neurons, and designed in layers as illustrated in Figure 4.8. The first (left) layer is the input layer while the last (right) layer is the output layer. The layers in between are called hidden layers. Moreover, when there are multiple hidden layers, the network is named a deep neural network (DNN). However, there are no straightforward rules for finding the best number of hidden nodes and layers in an ANN. The optimum number of hidden nodes and layers is defined based on the best prediction results, and the required number of hidden nodes and layers is usually chosen by adapting the number of hidden nodes during the learning process. We use ReLU (Rectified Linear Units) as a hidden layer and SoftMax as output layer. We implement the network using Keras library [104]. However, for evaluation, we run a 10-fold cross-validation technique to avoid overfitting.
4.6 Implementation and Performance

In this section, we describe the TEAM prototype implementation. Then, we evaluate the performance of our approach using real-world data and APIs.

4.6.1 TEAM Tool

We implemented a web interface for the TEAM approach dashboard using Shiny application in R. The package enables us to build an interactive web app to explore our model. Figure 4.9 shows the main TEAM dashboard interface. It consists of two main parts, the QoE visualization, and QoE prediction. The visualization page allows the API consumers to select API from the provided list, then it will reflect the QoE parameters as an interactive graph. Moreover, the application will provide five similar APIs using a k-nearest neighbor algorithm [105]. In addition, the QoE features are updated periodically to consider the new changes. This information can support the developers in their discussion to find the right API that can meet their requirements (e.g., response time, availability, reliability, and security).

The prediction page Figure 4.10 will help API providers to identify which QoE features need to be changed to attract more developers to use their API. This page enables the user to choose the dataset, classification model, and which API to predict. Afterward, it allows developers to edit QoE parameters and then predicts the API usage. Therefore, this will guide API providers to focus on specific features more than others such as publishing articles in ProgrammableWeb. However, Other features may not have the same effect on API usage. The dashboard employs the data after using min-max normalization to range from 0 to 100. Furthermore, API sponsors are able to set their plan to increase API popularity. However, in the case of the targeted API is not exist in the
list, the user can enter the newly developed API features to predict their future usage. Moreover, we provided more information about the model such as QoE abbreviation and dataset details under the abbreviation page.

Figure 4.9 The APIs QoE visualization page
4.6.2 Experiments

The aim of the experiments is to assess the ability of the TEAM model to correctly classify API usage from QoE parameters. We used several performance metrics to evaluate the
classification algorithms such as confusion matrix, Macro_Precision, Macro_Recall, and Macro_F-measure. We ran our experiments on a 64-bit Windows 10 Home environment, in a machine equipped with an Intel i9 and 32 GB RAM. We use RStudio for implementing the random forest. However, there are four fundamental measurements need to be calculated in classification applications:

1. True Positive (TP) Number of outcomes where the model is correctly labeled to a class.
2. True Negative (TN) Number of outcomes where the model is correctly rejected from a class.
3. False Positive (FP) Number of outcomes where the model incorrectly labeled to a class.
4. False Negative (FN) Number of outcomes where the model incorrectly rejected from a class.

We explore the classification evaluation metrics by focusing on precision, recall, F-measure, and Accuracy. Precision\text{\_rate} is the ratio of the number of true positives to the total number of reports labeled by the classifier as belonging to the positive class:

\[
\text{Precision}_{\text{rate}}(c_i) = \frac{TP}{TP + FP} \quad (8)
\]

Recall\text{\_rate} is the ratio of the number of true positives to the total number of reports that belong to the positive class:

\[
\text{Recall}_{\text{rate}}(c_i) = \frac{TP}{TP + FN} \quad (9)
\]

F - measure \((c_i)\) shows the harmonic mean of precision and recall:

\[
F - \text{measure}(c_i) = \frac{2 \times \text{Precision}_{\text{rate}}(c_i) \times \text{Recall}_{\text{rate}}(c_i)}{\text{Precision}_{\text{rate}}(c_i) + \text{Recall}_{\text{rate}}(c_i)} \quad (10)
\]
Accuracy is the proportion of correctly classified observations out of all the observations:

\[
\text{Accuracy} = \frac{TP_{all} + TN_{all}}{N} \quad (11)
\]

We labeled the API usage into three classes (Low, Medium, and High) as we described in section (4.3). Macro-averaging for (precision, recall, and F-measure) are computed after calculating the classification metrics for all classes and then taking their average, for an interested class \(c_i\) of API usage (here \(i = 1 : 3\)).

\[
\begin{align*}
\text{Macro Precision} &= \sum_{i=1}^{3} \frac{\text{Precision}_{rate}(c_i)}{3} \\
\text{Macro Recall} &= \sum_{i=1}^{3} \frac{\text{Recall}_{rate}(c_i)}{3} \quad (12) \\
\text{Macro F-measure} &= \sum_{i=1}^{3} \frac{\text{F-measure}(c_i)}{3}
\end{align*}
\]

Supposing that each class has a \(\text{Precision}_{rate}(c_i)\), \(\text{Recall}_{rate}(c_i)\), and \(\text{F-measure}(c_i)\), then the Macro_Precision, Macro_Recall, and Macro_F-measure can be calculated to evaluate the overall classification performance as shown in formula 12. Table 4.6 shows the classification models performance for both dataset MICE and KNN. The best classification result based on the Macro_F-measure is random forest classifier for MICE - metric - 1 with 60.98% Macro_Precision, 70.85% Macro_Recall and 65.49% Macro_F-measure. Moreover, for KNN date the metric - 2 reveals the best performance with 59.75% Macro_Precision, 71.82% Macro_Recall and 65.19%. In addition, metrics 1 and 2 show acceptable indicators to classify API usage based on the developers' communities. The justification for that is the number of reported mashups in ProgrammableWeb and the number of repositories related to API in GitHub can reflect the API activities on other platforms (e.g., HakerNews, CVE, Google Trend, YouTube). Furthermore,
metric 3 is not a sufficient indicator to classify API usage using the proposed quality of the experience features.

<table>
<thead>
<tr>
<th>API Usage Dataset</th>
<th>Metric</th>
<th>Classifier</th>
<th>Macro_Precision</th>
<th>Macro_Recall</th>
<th>Macro_F-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MICE</td>
<td>1</td>
<td>RF</td>
<td>60.98%</td>
<td>70.85%</td>
<td>65.49%</td>
</tr>
<tr>
<td>MICE</td>
<td>2</td>
<td>RF</td>
<td>51.54%</td>
<td>54.89%</td>
<td>53.09%</td>
</tr>
<tr>
<td>MICE</td>
<td>3</td>
<td>RF</td>
<td>55.24%</td>
<td>51.54%</td>
<td>53.28%</td>
</tr>
<tr>
<td>MICE</td>
<td>1</td>
<td>DNN</td>
<td>38.77%</td>
<td>44.68%</td>
<td>41.44%</td>
</tr>
<tr>
<td>MICE</td>
<td>2</td>
<td>DNN</td>
<td>35.65%</td>
<td>35.64%</td>
<td>35.64%</td>
</tr>
<tr>
<td>MICE</td>
<td>3</td>
<td>DNN</td>
<td>31.67%</td>
<td>35.20%</td>
<td>33.33%</td>
</tr>
<tr>
<td>MICE</td>
<td>1</td>
<td>SVM</td>
<td>32.15%</td>
<td>33.23%</td>
<td>32.65%</td>
</tr>
<tr>
<td>MICE</td>
<td>2</td>
<td>SVM</td>
<td>43.87%</td>
<td>35.19%</td>
<td>38.97%</td>
</tr>
<tr>
<td>MICE</td>
<td>3</td>
<td>SVM</td>
<td>31.35%</td>
<td>33.08%</td>
<td>32.12%</td>
</tr>
<tr>
<td>KNN</td>
<td>1</td>
<td>RF</td>
<td>61.67%</td>
<td>69.94%</td>
<td>34.16%</td>
</tr>
<tr>
<td>KNN</td>
<td>2</td>
<td>RF</td>
<td>59.75%</td>
<td>71.82%</td>
<td>65.19%</td>
</tr>
<tr>
<td>KNN</td>
<td>3</td>
<td>RF</td>
<td>53.36%</td>
<td>57.60%</td>
<td>55.36%</td>
</tr>
<tr>
<td>KNN</td>
<td>1</td>
<td>DNN</td>
<td>44.57%</td>
<td>48.21%</td>
<td>46.27%</td>
</tr>
<tr>
<td>KNN</td>
<td>2</td>
<td>DNN</td>
<td>37.42%</td>
<td>40.82%</td>
<td>39.02%</td>
</tr>
<tr>
<td>KNN</td>
<td>3</td>
<td>DNN</td>
<td>32.41%</td>
<td>36.02%</td>
<td>34.10%</td>
</tr>
<tr>
<td>KNN</td>
<td>1</td>
<td>SVM</td>
<td>71.78%</td>
<td>34.16%</td>
<td>46.21%</td>
</tr>
<tr>
<td>KNN</td>
<td>2</td>
<td>SVM</td>
<td>39.70%</td>
<td>31.50%</td>
<td>35.12%</td>
</tr>
<tr>
<td>KNN</td>
<td>3</td>
<td>SVM</td>
<td>31.04%</td>
<td>32.07%</td>
<td>31.49%</td>
</tr>
</tbody>
</table>

Table 4.6 Quantitative Performance Metrics for API Usage Metrics Classification
Furthermore, we utilized two datasets (knn-dataset and mice-dataset) in this experiment as described in section 4.3. Both datasets contain 2635 APIs to assess and compare the model performance, the data were split into training and testing sets in of 0.70 and 0.30, respectively. Figure 4.11 illustrates the accuracy of classification models. It can be seen that random forest (RF) outperforms other models. Specifically, (Metric 2 - KNN) has the best performance with 88.31% accuracy. Therefore, we consider this model as the best model to utilize the TEAM approach.

![Classification Accuracy](image)

**Figure 4.11 Models Performance**

Moreover, we investigate the best two classifiers (RF-MICE-metric-1 and RFKNN-metric-2). Figure 4.12 (A) shows the confusion matrix of RF-MICE-metric-1 where the majority of the APIs consider as medium usage. The classifier identified 482 out of 533 medium API usage class accurately. Figure 4.12 (B) illustrates the confusion matrix of RF-KNN-metric-2 where the majority around 75.8% of the dataset consider as low API usage. The classifier recognized 553 out of 564 low API usage class accurately.
The last part of our experiment is to study the variable importance to understanding and evaluating which variables have the most predictive power. Variables with high importance have a significant impact on the outcome values while variables with low importance might be omitted from a model. Figure 4.13 illustrates the variable importance of the best model (RF-KNN-metric-2). We noted that article from ProgrammableWeb, the average watch and fork from GitHub are the most explanatory variables for the model. However, the average response time to get the first answer from GitHub has the lowest explanatory power for the model.
4.7 Conclusion

We propose TEAM (quality of Experience-based Api recommendation), an approach and tool that leverages prior API development experiences to recommend APIs. We extract structured and unstructured information from various developer communities such as GitHub, StackOverflow, ProgrammableWeb, HackerNews, and YouTube to build a Quality of Experience (QoE) model. We train three different classifiers (random forest (RF), support vector machine (SVM), and neural network (NN)), to recommend APIs and predict their usage. We conduct extensive experiments on real-world and large data sets extracted from developer communities to evaluate the proposed approach.
CHAPTER 5

Conclusion and Future work

5.1 Conclusion

The aim of this Ph.D. dissertation is to combine social computing and artificial intelligence (hence the term social intelligence) to assist developers in building mashups or service composition. We introduce techniques to collect and analyze historical API-related data in developer communities and turn such data into useful information to support mashup development. To the best of our knowledge, this research is the first to combine social computing (i.e., leverage developer data from communities), artificial intelligence (i.e., machine learning and natural language processing), and service computing (i.e., APIs, service composition, and mashup) to recommend APIs for mashup development. First, we propose a crowdsourcing-based approach called CrowdMashup to recommend teams for mashup development [15]. To the best of our knowledge, this is the first research to recommend developers not APIs for mashups and service composition. Second, we introduce FAME, a multidimensional model to analyze and evaluate the influence of APIs on mashup development [17]. To the best of our knowledge, this work is the first to extend the well-known concept of “influencers” (usually people) in social media to APIs and service-oriented environments. Third, we propose TEAM, a model to leverage experiences from developer communities for API recommendation [18]. The model assesses the quality of APIs for mashup development. The originality of our model is that it looks at quality from a developer’s
perspective (e.g., how is an API provider responsive to fixing bugs) not from the traditional API’s or API provider’s perspective (e.g., invocation execution time).

5.2 Future Work

Service evolution and service documentation are two research directions we plan to pursue in the future. We would like to study the APIs evolution behaviors based on the developers’ communities. This direction will analyze APIs features over time. For instance, analyzing the APIs versions with reported bugs to understand how fast API providers respond to reported bugs. We also intend to automate the process of service documentation based on the developers’ communities. Our vision is to detect APIs issues and their solutions from developers’ communities. Moreover, the advantage of building documentation from the developers’ communities is that the communities include the missing information which is not clear or missed on the official documentation.
Bibliography


[104] François Chollet et al. keras, 2015.