5-2019

Methods of Disambiguating and De-anonymizing Authorship in Large Scale Operational Data

Sadika Amreen
University of Tennessee, samreen@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Sadika Amreen entitled "Methods of Disambiguating and De-anonymizing Authorship in Large Scale Operational Data." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Audris Mockus, Major Professor

We have read this dissertation and recommend its acceptance:

Russell Zaretzki, Bruce MacLennan, Jian Huang

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Methods of Disambiguating and De-anonymizing Authorship in Large Scale Operational Data

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Sadika Amreen

May 2019

To Abbu and Ammu

Acknowledgments

First and foremost, I would like to thank Dr. Audris Mockus, my advisor, for encouraging me to ask important open research questions that make strong contributions to the open source software community, but that also benefit to downstream end users. I also thank my committee for their time, support and guidance as I stepped into the unknown: Dr. Russell Zaretzski, Dr. Bruce MacLennan and Dr. Jian Huang.

Further, I would also like to thank Dr. Ostrouchov and his team for their continuous support with all of the pbdR related question that emerged throughout this project. I would also like to thank the Oak Ridge Leadership Computing Facility (OLCF) for granting us access to Titan which has provided us with a vast amounts of computational resources without which this research would not have been possible.

I am grateful to have performed this research in a department that goes above and beyond to create a positive environment for graduate students. Thanks to my friends and peers in the department: Yuxing, Tapajit, Sara and many others who have contributed to my work and development in so many ways. In addition, I would like to thank Denise for her mentorship and guidance when I was starting my career as a graduate student.

Finally, I would like to thank my parents — I am where I am today because of their love and support — and my husband, Reazul, for being supportive and patient throughout this journey.

Note: This work was supported by the National Science Foundation ([Grant No. 1633437]). All opinions, findings, conclusions, or recommendations expressed in this document are those of the author(s) and do not necessarily reflect the views of the sponsoring agency.

Abstract

Operational data from software development, social networks and other domains are often contaminated with incorrect or missing values. Examples include misspelled or changed names, multiple emails belonging to the same person and user profiles that vary in different systems. Such digital traces are extensively used in research and practice to study collaborating communities of various kinds. To achieve a realistic representation of the networks that represent these communities, accurate identities are essential. In this work, we aim to identify, model, and correct identity errors in data from open-source software repositories, which include more than 23M developer IDs and nearly 1B Git commits (developer activity records). Our investigation into the nature and prevalence of identity errors in software activity data reveals that they are different and occur at much higher rates than other domains. Existing techniques relying on string comparisons can only disambiguate Synonyms, but not Homonyms, which are common in software activity traces. Therefore, we introduce measures of behavioral fingerprinting to improve the accuracy of Synonym resolution, and to disambiguate Homonyms. Fingerprints are constructed from the traces of developers' activities, such as, the style of writing in commit messages, the patterns in files modified and projects participated in by developers, and the patterns related to the timing of the developers' activity. Furthermore, to address the lack of training data necessary for the supervised learning approaches that are used in disambiguation, we design a specific active learning procedure that minimizes the manual effort necessary to create training data in the domain of developer identity matching. We extensively evaluate the proposed approach, using over 16,000 OpenStack developers in 1200 projects, against commercial and most recent research approaches, and further on recent research on a much larger sample of

over 2,000,000 IDs. Results demonstrate that our method is significantly better than both the recent research and commercial methods. We also conduct experiments to demonstrate that such erroneous data have significant impact on developer networks. We hope that the proposed approach will expedite research progress in the domain of software engineering, especially in applications for which graphs of social networks are critical.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

While data errors related to identities of individuals, have been studied for over a 70 years, the existing systematic approaches do not work well on the so-called operational data collected in modern online systems such as Twitter, Facebook, or GitHub. Studies that try to understand why the existing approaches do not work well in this context do not exist, yet the need to address this problem is rapidly growing due to the extensive research and applications that rely on such data, such as, analysis of social and technical networks. We, therefore, develop a framework ALFAA (Active Learning Fingerprint based Anti-Aliasing) – that builds on existing state-of-the-art research from multiple domains, adds important innovations, such as behavioural fingerprints, systematizes all stages of identity correction process, increases the credibility of the results through multiple stages of validation and the extensibility of the modeling and training data, and offers theoretically unlimited increases in accuracy as additional training data is added. Furthermore, ALFAA minimizes the manual effort needed to produce training data by focusing on areas where existing models do not have reliable predictions.

ALFAA, unlike prior methods that either rely on theoretical assumptions or heuristics, is more credible because the heuristics are not pre-set, but are discovered via training a machine-learning model and theoretical assumptions are not used, but actual relationships in the specific data that is being corrected are exploited. ALFAA includes

approaches such as text component similarities, supervised learning, and active learning with innovations represented by behavioural fingerprints that summarize complex and unstructured data in ways that help to correct identity errors. We expect that our systematic approach will spur active research and application development for domains where low accuracy of identity data either makes research extremely labor intensive or, especially on the large scales, impossible. In the process of implementing the framework, we make a number of other significant contributions, such as, recognizing different types of errors that need different approaches to correct them, the ground-truth data obtained through manual validation, set of models and code to support reproducibility of the results and to foster future enhancements.

Our vision for the framework, ALFAA, involves two primary aspects: one is the systematization of identity error correction approaches in all domains, and the second aspect relates to increasing accuracy and scale of identity correction in an important domain of software development. To achieve that vision, our work will provide directions to researchers in all domains on how to create suitable set of scalar and behavioural features, how to create training data sets with minimal effort especially in domains where ground truth is extremely difficult or nearly impossible to obtain, and how to increase the accuracy of the resulting models. For the software development domain, we hope to help researchers to improve upon the scalability of the approach – to go beyond the current ability of disambiguating 2 million identities, and reach the scale of all developers in open source which may require 400 times or more computational power (assuming 40M or more developers). Our tools, are immediately applicable to not only developers seeking to build an online profile of their work (much similar to scientific scholar profiles on Google) that unifies work done under various IDs across platforms into a complete collection of contributions, but to researchers as well who can leverage these profiles to build correct collaboration networks to perform analysis of any open-source projects. We are deeply motivated by the fact that the removal of the barriers to accurate determination of identities would potentially lead to reanalysis and correction of existing studies that are not based on accurate reconstruction of identities, and therefore, more vigorous research in this area.

There are a number of factors that were kept in mind while making the design decisions critical to ALFAA. First and foremost, ALFAA is designed to be extensible – as data sets may differ between domains and practices may change over time, therefore, preset heuristics and theoretical assumptions will not suffice. Second, ALFAA outlines ways to minimize manual effort for training – as it is impossible to have resources to do manual validation on hundreds of millions of pairs. Third, ALFAA is designed to address a problem that existing solutions do not – the `Homonym` errors since existing methods are simply not applicable (as explained later in this dissertation). Therefore, the concept of behavioural fingerprints used in ALFAA, not only increases the accuracy of `Synonym` resolution, but opens a completely new opportunity to address the `Homonym` problem as well.

1.1.1 A Brief Historical Perspective

The idea of disambiguation of entities (often referred to as *Record Linkage*), has been current for over 70 years. In 1946, Halburt L. Dunn, the Chief of the National Office of Vital Statistics in the U.S. Public Health Service, eloquently described the term [16].

> "Each person in the world creates a Book of Life. This Book starts with birth and ends with death. Its pages are made up of the records of the principal events in life. Record linkage is the name given to the process of assembling the pages of this Book, into a volume. The person retains the same identity throughout the Book. Except for advancing age, he is the same person. Thinking backward he can remember the important pages of his Book even though he may have forgotten some of the words. To other persons, however, his identity must be proven. "Is the John Doe who enlists today in fact the same John Doe who was born eighteen years ago? "

In this work, Dunn has further emphasized the importance of linking records to the individual, to the registrars of vital records, and to the health, welfare and other agencies and organizations. Needless to say, machines were not available and simple heuristics had to be defined for manual matching. As the volumes of records grew with time, and

the importance of linking those records remained undisputed, a more formal approach was developed by Newcombe et. al [48] in 1959 and a formal mathematical model was introduced by Fellegi and Sunter [17] a decade later in 1969.

However, all of this work applied mostly to the public health domain and was done manually since mechanized searches were unavailable at the time. Later on, record linkage was used in many other fields such as disambiguating census data [68], merging citation data [54], and others. However with the advent of the Internet, record linkage (which also became known as "identity merge") became a useful concept in the studies of communities on social platforms, i.e., in social network analysis, software repository analysis, etc. For example, studies of social diversity [66] are important in assessing the effectiveness of teams as social diversity is a source of creativity and adaptability, and to do such studies, it is important to have the correct identities of team members. In other cases, such as software engineering research, which aims to gain a better understanding of software products and processes, identity merge or disambiguation is also require to combine multiple data sources in a coherent way before any results can be obtained.

In the case of social network data, identity merge is particularly challenging because the persons involved may have different accounts, login credentials such as user names and email addresses which need to be mapped to the person to whom they belong. Furthermore, the existing identities are staggeringly numerous (for example, about 16 million identities were discovered in GitHub alone during the time of this research) and therefore doing disambiguation manually is error-prone if not impossible. Much work has been done to automate this process but a number of problems, discussed later in this thesis, remain to be addressed. In this thesis, we introduce a method that builds upon the traditional record matching techniques, particularly on *operational data*, and leveraging supervised machine learning techniques with the emphasis on expending the minimum manual effort needed to produce the method training sets. We find that our method approach provides results that present a major improvement upon recent state-of-the-art research and commercial efforts in this domain.

1.2 Problem Statement

Operational data (OD), defined as "data left as traces from multiple operational support tools and then integrated with more traditional data" [44], is generated in software development, social networks and other domains, such as patent databases, research publication databases [68], etc. OD can come in a variety of formats and can be derived from interactions between people and machines, such as web applications, or from profiles stored on social networks, or from identities recorded in issue trackers. Although operational data is an integral part of software across many domains, it either lacks a clear structure (such as in bug descriptions or commit messages) or has a very complicated and non-homogeneous structure, as in the source code written in numerous programming languages. OD may also contain a variety of interrelated entities that are often incompletely specified or erroneous. Common examples of such errors include the misspellings or changes of names, duplicative email addresses, and inconsistent user name profiles in different systems. As these digital traces from operational support tools are important in practice, such problems have to be addressed in order to improve the quality of these data. This process is sometimes referred to as "data cleaning" [41], "data cleansing", "data scrubbing", "object identification" [63] etc. The basic objective of these techniques is to detect and amend or remove information that is inaccurate, incomplete or duplicative in order to increase the accuracy of measures and models derived from the data.

> Data cleansing or data cleaning is the process of detecting and correcting (or removing) corrupt or inaccurate reports from a record set, table or database and refers to identifying incomplete, incorrect, inaccurate or irrelevant parts of the data and then replacing, modifying, or deleting the dirty or coarse data. [79]

Our particular concern lies with applications when the problematic data represents a network. The properties of the network measures, such as connectivity or betweenness centrality, may be affected even more profoundly if the constituents of the network, i.e., nodes and edges, are inaccurate. As understanding developer networks plays a

crucial part in the development of software products and processes [4] and improves the understanding of how open source communities evolve over time [40], it becomes critical to correct such problematic data. In particular, the errors generated from developers having different accounts, and multiple login credentials involving user names and email addresses are a specific problem with data from various operational support tools in the realm of software engineering. We refer to these credentials as an user's "signature", as they are intended to be distinctive forms of identification used in authorizing a person's activity in a platform or a tool. It becomes a concern, when a single individual uses multiple such signatures, as well as when multiple individuals use the same signature for their activities. This is especially true when it becomes hard to track the work of individuals such as inventors in patent databases, projects and developers in version control systems, and researchers or publications in the collections of scholarly work, and therefore profoundly affecting most social network or productivity measures.

In version control tools the identity signature may come from a user profile that is stored locally on a device as is the case of credentials in Git. A single individual is, therefore, often represented as multiple identities not only across platforms as studied by [2], but also within a single platform. As discussed, this might impact the productivity measure of a developer by showing lower than actual productivity because his activities have been logged using two or more signatures, or, alternatively, higher productivity because multiple individuals are active using the same signature. As incorrect identities are likely to result in incorrect networks [70], thus making the subsequent analysis and conclusions questionable, such misrepresentation needs to be "cleansed". This can be done by mapping each signature to the actual individual or by finding constituent individuals using a single signature. However, resolving these signatures, which we refer to as `identity resolution`, `disambiguation`, and/or `de-anonymization`, to identify actual developers based on data from software repositories is a major challenge. This is mainly due to the following factors:

1. Lack of ground truth – absence of validated maps from the recorded signatures to actual identities and

2. Very large volumes of data – Millions of signatures of individuals found in a single platform

These issues have been recognized in the field of software engineering [21, 5] and beyond [14]. To cope, with such issues, studies in software engineering have tended to focus on individual projects or groups of projects in which the number of IDs that need to be disambiguated is small enough for manual validation and for which a variety of heuristics have been devised to solve this formidable problem. The social networks of scientific paper authors or patents [68] must handle a much larger set of identities, and the population census [14] has an even larger set of identities. The latter literature refers to the accurate identity problem as "record matching", or how to match records in one table (e.g., a list of actual developers) with records in another table (e.g., code commits) where some of the fields used for matching may differ for the same entity. The way these techniques are applied in practice for author resolution, however, is primarily to resolve `Synonyms` (instances where the same person may have multiple IDs), but not `Homonyms` (instances where the same ID is used by multiple individuals). Figure 1.1 illustrates that the goal of this research is two-fold; that is, it is both disambiguation (finding multiple IDs that represent the same individual) and de-anonymization (finding individuals behind missing and generic identity signatures).

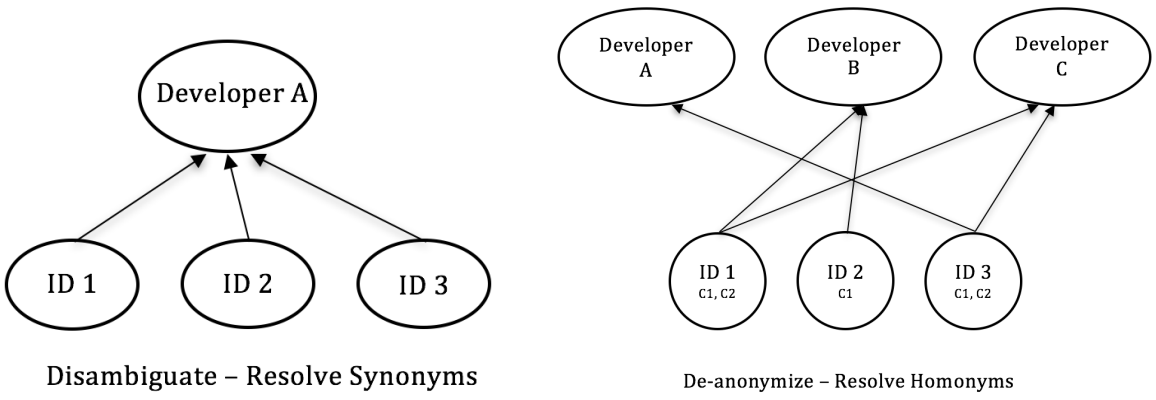

Figure 1.1: Goals of this Research - Correct `Synonym` and `Homonym` errors on data collected from Version Control Systems (VCS)

Examples of Synonym Errors:

ID 1 - Pradeep:pradeep@arrayfire.com

ID 2 - pradeep:pradeep@arrayfire.com

ID 3 - pradeep:pradeep.garigipati@gmail.com

Examples of Homonyms:

ID 1 - A Student:secretmessage@protonmail.com

ID 2 - Jenkins:jenkins@jenkins.hpcloud.net

ID 3 - (no author):(no author)

In software engineering and, in particular, in code commits to a version control system, author information in a commit is often ambiguous and generic (drawn from roles or names of projects). This leads to the same credentials being reused for multiple individuals, such as, logins ("root", "Administrator"), group or tool IDs ("Jenkins Build"), or identifiers seeking to preserve anonymity ("John Doe", "name@domain.com"). Furthermore, software data does not have a database structure similar to population census data, where the birth-date field helps to resolve homonyms, and, more generally, appears to contain a substantially larger fraction of records with errors. In this work, we pose a number of research questions whose answers will help to address these issues. These are the following:

1. What are the most common reasons for identity errors in version control data?
2. Can we encode developer behavior from version control data to help correct identity errors?
3. Does our approach improve upon existing matching techniques in software engineering that are used in research and commerce?
4. What is the impact of identity errors on actual collaboration networks among developers?

Even with significant work that has been done to advance methods in disambiguating identities in authorship for research papers and patents [14, 68, 77, 60], correcting the identity errors in a software engineering context appears to be challenging, as the nature of the errors appears to be quite different. Therefore, we seek a better understanding of their nature and extent in order to tailor the correction techniques for the software engineering domain and propose a solution in the form of a framework (ALFAA), that is extensible and can be adopted in other domains as well.

1.3 Motivation

With the explosion of online activities enabled by the internet, online social networks have become an integral part of our daily lives. A simple Google search on 'Social Network Analysis' yields over 177 million results showing the vast extent and importance of this topic. Social network analysis is based on relationships among social entities (that interact) and on the patterns and implications of those relationships [72]. To drive such analysis, it is imperative that the social entities are represented correctly in the network. For example, studies on code attribution [3] i.e. code that is taken from StackOverflow and committed to repositories such as GitHub, require the correct representation of people on the network to answer questions such as "How often is code from StackOverflow reused in GitHub projects?", and to accurately measure developer productivity [67]. Accurate representation of any online social network is also required to examine social connected-ness in developer communities as people involved in software development are connected mostly through and around code [64, 2]. The importance of accurate representation was discussed by Wang et. al. in [70] where it was shown that identity errors have a large impact on reconstructed social networks. In addition, however, to the importance of having correct identities on any social network, we focus on why it is important to have correct developer identities in operational data. This section discusses the two driving factors that motivate the research undertaken in this PhD study, keeping in mind that the data we consider for our research is derived from version control systems.

1.3.1 Correcting Operational Data

As pointed out in [24], operational data collected in the course of system operation is a common source of measurement in data science. However, due to the size, complexity, and non-experimental nature of OD, observations and measures derived from it suffer from the lack of context, missing observations, and incorrect values. Examples of lack of context include absence of explicit markers in the data that would indicate different purposes an individual was engaged in when using the system. For example, an email sent to a mailing list may be a question, a code submission, or a patch to an existing bug, or a general announcement. Some activities under study may occur off-line, thus not leaving traces in the system. That may have a strong impact on the social network structure if no instances of the social interaction were recorded on the system. This dissertation is mostly concerned with solving the third challenge of the operational data i.e., correcting mis-specified values associated with various identities.

Specifically, we investigate the feasibility of correcting social network data for each record, i.e., commits, by assigning the most likely identity associated with it using an amalgamation of traditional string similarity techniques and behavioural fingerprinting – that is, disambiguating. When the identity record is empty, we attempt to insert a label for that record; thus, this task can also be considered as missing data estimation task. For missing and multi-person IDs, the task can be thought of as de-anonymization. A similar problem in the database field is referred to as the record matching problem ([74, 77]).

1.3.2 Mitigating Risks in the Software Supply Chain

One of the greatest motivations for identity de-duplication is in the realm of Software Supply Chain. To better understand the concept of Software Supply Chain (SSC), it is important to clarify what a physical supply chain is and what aspects of it may be applied to a software context. A supply chain is a system of organizations, people, activities, information and resources involved in moving a product or service from a supplier to a customer[1]. Supply chains function as a result of diverse markets, widespread

[1]https://en.wikipedia.org/wiki/Supply_chain

collaboration, and, most importantly, successful risk management. Flexibility is crucial in supply chains; with emerging markets and increased number of corporate players, supply chains need to be able to respond to changes within the market and the political climate through its fulfillment of demand[2]. Finally, a thriving supply chain network requires end-to-end visibility between the supplier and the customer. This can eliminate potential risks by allowing entities within the network to see the flow of goods through the supply chain, therefore minimizing the effect of disruptions.

Like the physical supply chain, a software supply chain consists of a complex network of organizations, developers and projects that are connected through various software dependencies. An end-to-end visibility would require understanding of the nature of the individual developers' participation in a collaborative environment and it is of utmost importance that these developers be represented correctly. For example, Project P_B, developed by developer D_B, maybe dependent on Project P_A, developed by Developer D_A, through the libraries/packages of P_A. In essence, P_A is a supplier for P_B and the same concept can be relayed to the developer and therefore, D_A is a supplier for D_B. Therefore, from the SSC perspective, a customer (P_A or D_A) has a direct or indirect upstream dependency on its supplier (P_B or D_B).

As with physical supply chains, software supply chains are vulnerable to risks that may cause disruptions across the network. For example, if an update is released in Project P_A without allowing backward compatibility, it may cause a breakage in Project P_B. To mitigate such risks it may be necessary for P_B to have a secondary supplier just as physical supply chains do. To minimize risk, therefore, P_B has a "backup dependency plan" on Project P_C developed by developer D_C. However if in actuality, D_C and D_A are the same developer using two aliases, Project P_B's risk mitigation scheme becomes ineffective. Therefore, it is critical that user credentials (such as D_A and D_C) are linked so as to allow greater visibility across the network.

[2]http://cerasis.com/2015/07/30/best-in-class-supply-chain/

1.4 Goals and Contribution: Bridging the Gap

In this study, we propose, outline and develop a framework (ALFAA), focusing on customizing solutions for the types of errors found in operational data. ALFAA uses new a approach that significantly outperforms existing techniques, in terms of reduced manual effort in the correction process as well as increased accuracy. Through this work we discuss how the problem we address is similar to the record matching problem as introduced in [17], and what additional challenges we face as a result of domain specificity. A detailed discussion is provided on traditional record matching techniques and state-of-the-art techniques that have been used in solving this problem. The following points summarize the contributions of this study:

1. **A framework and a corrected data set:** In this work, we design a framework, ALFAA (Active Learning Fingerprint Based Anti-Aliasing), that allows tuning so that this method can be used for other domains. Furthermore, the models that have been trained for this research can be further improved simply by adding larger training data instead of requiring effort intensive design and application of customizable heuristics. As an outcome of this work, we have produced a corrected set of developer identities on several open-source projects on GitHub that maybe used for practical purposes or research related to software engineering/ repository mining, etc. We believe that correct representation of networks plays a critical role in spurring research.

2. **Understanding and classifying errors:** Identity errors in the software engineering context appear to be quite different from those in other domains that have been researched in the past. We therefore seek a better understanding of their nature - the mechanisms that lead to variations i.e. errors and missing values in identity representation and broadly classify errors into categories. This enables us to tailor the correction techniques for the software engineering domain.

3. **Use activity patterns of developers for disambiguation and show that they positively impact results:** Most traditional record matching techniques use string

similarity of identifiers (typically login credentials) i.e. name, username and email similarity. A broad spectrum of approaches ranging from direct string comparisons of name and email [56] to supervised learning based on string similarity [68] has been used to try to solve the identity problem. However, such methods do not help with homonyms, which are common in software engineering data. We, therefore, need additional pieces of information (an analogue to date of birth for census records). For this purpose, we propose to enhance the string similarity-based techniques with what we call *behavioral fingerprints* or activity patterns that tend to be more similar if different IDs are used by the same individual and less similar for IDs of distinct individuals.

We evaluate the accuracy of our approach on a large sample of over 16K OpenStack contributors and compare it to the approaches of a commercial method and a recent research method. We also apply our method on a large sample of 2 Million contributors in 18 large ecosystems hosted on GitHub. We compare our results to the recent research method and through a manual process and show that our method produces fewer errors than the existing state-of-the-art method. Finally, we conduct an experiment to show that identity errors have significant impact on developer collaboration networks and that correcting them is crucial before conducting any analysis.

4. **Reduction in manual effort required to build validated data sets:** Much of the existing literature [68] relies on the availability of adequate manually corrected data to use in the correction process. This is often unavailable and expensive to acquire for any particular domain. In this work, we reduce the amount of manual effort needed for correction by using a method called *active learning*. Through this method, we are able to identify a small training data that can be manually labeled and will be used to train an effective model.

1.5 Thesis Organization

This thesis is divided into six chapters. The remaining of this thesis is organized as follows:

Chapter 2 provides a foundation for the research contributions discussed in the following chapters through a comprehensive survey of the work done in record linkage. The history of record linkage is discussed along with and the relevant work done by researchers in the past in diverse domains. In the subsequent sections of this chapter, we also define some commonly used terminology in our research in the following chapters.

Chapter 3 dives into the details of the data collection process for this research. We describe our data collection process from Git repositories used for version control in software development. Specifically, we focus on GitHub, which offers plans for both private repositories and free accounts, which are usually used to host open-source software projects. We discuss how data is stored in Git, and how we extract data from this system, i.e., our process of discovering projects, retrieving information, storing and cleaning the data. Furthermore, we describe the data retrieved focusing on errors and we discuss methods used to classify the errors. We include a list of keywords that we discovered that may be used to identify homonym errors in large scale data.

Chapter 4 discusses the framework we designed to correct identities of developers obtained from GitHub. This includes measures of string similarity as used by traditional record matching techniques augmented with measures of developer behaviour which we refer to as "behavioral fingerprints". We also discuss how to create an effective training set for supervised classification with minimum manual effort.

Chapter 5 reports the results generated by the prototype built on a small subset of the data as well as techniques used to validate the training data. This chapter also provides a detailed evaluation of the how the proposed technique compares against a commercial effort and recent research. Finally, this chapter discusses limitations of our work.

Chapter 6 concludes by reiterating over the research questions and the answers we found, as well as the summary of contributions made in this thesis. Lastly, it proposes some ideas that can be implemented to advance this research in the future.

Chapter 2

Background Literature and Significance

2.1 Overview

In this chapter we review some existing research literature in order to gain familiarity with work that has been done thus far on record matching and the approaches applied to accomplish similar goals as our work. In this chapter, we provide a foundation for the research contributions discussed in the following chapters through a comprehensive survey of the work done in this field. Here, we talk about some definitions, assumptions for our work and the relevant work that was done by researchers in the past. In the following sections of this chapter, we also define some commonly used terminologies in our research in the chapters to follow.

Note: Throughout the rest of this document we use the terms record linkage, identity linking, identity deduplication, identity merge, record matching and identity disambiguation interchangeably.

2.2 History of Record Linkage

The field concerning linkage of records has been around for more than half a century. The term "record linkage" has been used to refer to the process of merging two or more separately recorded pieces of information related to a particular individual or family [16], in as early as 1946. There were various defined heuristics for manual matching as

"mechanized searching" was unavailable and there was "no clear demonstration that machines could carry out record linkages rapidly enough, cheaply enough and with sufficient accuracy to make this practicable" [48]. The usage of such methods were mainly in the field of public health i.e. for keeping track of people exposed to low levels of radiation in order to determine causes of their eventual deaths, for assessing fertility differentials, genetic defects in human populations etc. Some heuristics used in manual matching of these records were birth and death registration information, marriage registration, etc as defined in a study conducted by H.B Newcombe et. al in 1959 [48].

A decade later, the first mathematical model "to provide a theoretical framework for a computer-oriented solution" for record linkage was introduced by Ivan Fellegi and Alan Sunter [17]. The interest in this field was driven by

1. The creation of large files (particularly by various administrative programs) that required maintenance over long periods of time that contained information whose value could be increased by linking individual records in other files,

2. Increasing awareness on the potential impact of linking records in medical and genetic research and finally

3. Improvements in mechanization, making record comparison between medium sized files economically and technically feasible.

The theory behind the mathematical model for record linkage as described by Fellegi and Sunter in [17] is as follows:

There are two populations A and B whose elements are denoted by a and b respectively. We assume some elements of these populations are common to both the populations.

Therefore, the set of ordered pairs:

$$A \times B = \{(a, b); a \in A, b \in B\}$$

is the union of two disjoint sets

$$M = \{(a, b); a = b, a \in A, b \in B\}$$

and

$$U = \{(a, b); a \neq b, a \in A, b \in B\}$$

where M and U are the matched and unmatched set respectively.

Each unit in A and B has a number f characteristics associated with it (e.g. name, age, sex, marital status, address at different points in time, place and data of birth etc.)

This model serves as the basis of many record linkage methods practiced today. Since then, the problem has been investigated in many fields - at times via elementary and ad-hoc rules and at times via formal mathematical models using "computer matching techniques" that were tested using statistical methods. One such example of the more structured approach is the US census [78, 77] where administrative data from multiple sources is brought together to aid in making policy decisions - as an alternative to designing special surveys to collect data - as existing data from administrative sources might contain more information which is also more accurate due to improvements over many years. Other examples include linking records of inventors in the United States Patent and Trademark Office (USPTO) database [68] to link records of the companies, organizations and individuals or government agencies to which a patent is assigned. These studies adapt the methodological advancements in statistics [18] moving away from the previously used ad-hoc weights, thresholds and decision rules in their disambiguation process. More examples include linking records in synthetic census data [14] and in the construction of web services that integrate crowd-sourced data such as CiteSeer [33].

With the advent of the internet, there has been a rapid increase in large-scale software development involving a large number of project workers. This requires a significant amount of communication and coordination and many platforms or tools have been built to aid and expedite these development processes. With this increase in

collaboration, the volumes of data also multiplied rapidly. Existing hand-coded rules and thresholds is tedious and requires domain knowledge and approaches requiring such rules become inefficient. One way to reduce the tedium of hand-coding is to use machine learning algorithms to find the deduplication function [78, 76, 27]. By the early 2000s, identity disambiguation became an important topic in the field of empirical software engineering research [21] and in mining social networks [5] to correctly identify people in a software ecosystem for various purposes. These include building social diversity data set from thousands of GitHub projects [66], assessing the contributors total activity within projects [22] in Open Source Software and across platforms [80] and in mailing lists [75]. Many of these methods are reliant on simple string matching heuristics. The issue of developer identities has been a serious problem in software repository mining, particularly when trying to combine information from different types of data sources in a coherent way where the available data concerning persons involved in a project may be dispersed across different repositories [23, 53].

In summary, identity disambiguation is an well researched area that has evolved over decades, starting with its inception to solve problems with data in the public health sector with ad-hoc methods, moving on to a formal mathematical model introduced in 1969 that serve as the basis for most research conducted in this area till date. Since its inception, record linkage has been used in a wide variety of fields for various purposes and the heuristics and methods used for linkage have become increasingly complex. For the rest of this chapter, we focus mostly on the identity disambiguation done in the domain of software engineering. The following sections discuss the application of algorithms specific to record linkage methods used in email social networks, version control systems or software repositories, bug tracking systems, online communication platforms etc. as well as the research gaps that were identified, thereby motivating our work in this field.

2.3 Identity Merge in Software Repositories

To gain a better understanding of software products and processes, software repository mining research uses empirical studies, experiments and statistical analysis. One of

the challenges in software repository mining is how information coming from various data sources can be combined in a coherent way. For many projects, data concerning developers involved is dispersed across several repositories and may be accessed using different tools. For example,

1. Version Control Systems (VCS) - are used to store and track changes in source code (e.g. CVS, Subversion, Git)

2. Communication platforms - mailing lists, discussion boards, development forums (e.g. StackOverflow, StackExchange) are used for communication between developers and/or users

3. Bug Tracking Systems - are used to track bug reports and change requests (e.g. Bugzilla, Jira)

Reconciliation of data from various sources (from within a platform or across platforms) is crucial in order to carry out empirical studies on software repositories - which involves the challenge of merging identities involved in the software process. The tools used to access various data sources often have different mechanisms to access and modify the data and contributors to each data source may have different accounts, login credentials such as user names and email addresses and may need to be correctly mapped to the actual developer to which the credential belongs.

In the following subsections, we discuss a number of well known literature regarding identity merge - the problems addressed and the solutions proposed by them.

2.3.1 Identity Merge using String Similarity and Other Surface Attributes

Traditionally identity merging algorithms were commonly based on matching surface attributes such as strings (names and emails of persons). A popular algorithm that still serves as the basis of many identity merge algorithm has been designed by Bird et al. [5] specifically to deal with identities belonging to committers in a code repository or mailers

participating in a mailing list. In this work Bird et al. conducts pre-processing which includes cleaning and normalization of the data. This is followed by splitting a name into its components using whitespaces and commas as delimiters. Bird checks the following conditions to determine whether an identity should be merged. An identity is merged if at least one of the following holds true.

- If the normalized Levenshtein similarity [46] between two names (l_1 and l_2) of users is greater than a defined threshold.

$$similar(l_1, l_2, t) = \begin{cases} True, & \text{if } 1 - \dfrac{levenshteinDistance(l_1, l_2)}{max(size(l_1), size(l_2))} \geq t \\ False, & \text{otherwise} \end{cases}$$

- If the similarity between components of names of users is greater than a defined threshold.
- If the similarity between user names (also referred to as email prefix) of users is greater than a defined threshold.
- If the name components of one user matches the user name (components) of another user i.e 'John Doe' and 'john.doe'.
- If the name component of one user matches a user name of another user that is a partly abbreviated version i.e. 'jdoe' for 'John Doe'.

Initially, Bird's algorithm makes an incorrect assumption that a person's name always contains two parts - the first and last name. This was later modified in an article [23] - on the comparison of various merge algorithms software repositories - to include splitting a name into an arbitrary number of parts using a larger number of delimiters/separators i.e. space( ), comma(,), plus(+), period(.), hyphen(-) and underscore(_). This article iterates over a number of existing and new identity merge algorithms and measures their effectiveness, using metrics such as precision and recall by applying them on large ongoing Open Source Software (OSS) projects.

Another study [56] to link accounts across social networks such as StackOverflow, Github and Twitter, discusses some techniques such as using different attributes of user profiles, to build user interaction networks. The authors discovered 4,132,407 and 4,288,132 accounts in StackOverflow and GitHub respectively. These sets of accounts were matched to each other and the resulting set was matched to Twitter accounts using the three techniques discussed below:

1. Explicit matching - This is the process of identifying users through the links provided by users in one platform to their accounts in other platforms. In this study, 4,536 users of StackOverflow who use Github and 10,068 who use Twitter were identified as well as 433 Github users who use Stackoverflow and 7,012 users who use Twitter were identified using this strategy.

2. Attribute-based matching - This technique uses unique attributes of users, such as email (MD5 email hash) and profile image (GravatarID[1]) to connect profiles across multiple platforms. Using this method 604,083 user overlaps between StackOverflow and Github was discovered.

3. Fuzzy matching - This method applies less accurate user attributes such as login names and profile images to match profiles across platforms.

This study by [37] analyzes names across social networks and identifies the redundant information in different display names for the same real-world entity. The authors report through their analysis that 45% of users tend to use the same display names across online social networks. The awareness of redundant information between the display names can benefit many applications such as user identification across social networks. This study covers a number of analysis for display names such as length similarity, character similarity, letter distribution similarity etc., each derived from a number of metrics. This study is a relevant compilation of work that will be necessary once we reach the validation stage in our research.

There are also studies[74] on about entity-name clustering which is a task of taking a single list of entity names and assigning them to clusters such that all names in a

[1]https://en.gravatar.com/ Gravatar, a globally recognized avatar

cluster are co-referent (i.e. refer to same real-world entity). This study presents various techniques that are scalable and adaptive.

2.3.2 LSA for Identity Merging

While string matching and approaches using thresholds are reported to perform fairly, work using more sophisticated heuristics such as Latent Semantic Analysis (LSA) on names of GNOME Git authors was also used for disambiguation [29]. In this work, authors classify existing identity merge algorithms into two groups: endogenous and exogenous algorithms. Endogenous algorithms, such as the one used in [5, 11], try to match full names or email addresses shared by different identity signatures (IDs) used by individuals. These algorithms only use the information available in the repositories the identity signatures come from. Contrary to this, exogenous algorithms [53, 51] also use external information in addition to heuristics to aid in the matching process e.g. GPG (GNU Privacy Guard) key servers to determine if email addresses are linked. Such approaches may not always prove to be useful in OSS and many open-source projects do not use GPG keys.

LSA uses a sparse term document matrix which describes the occurrence of terms in documents. Each document used in this research is the email address and the terms in a document is the name parts i.e. components of the email address (that has been normalized). The authors in [29] argue that LSA is robust against more types of differences in aliases as reducing the dimensionality of the term document matrix eliminates much of the "noise" in the data. The aliases corresponding to pairs of documents for which the cosine similarity meets a defined threshold are merged.

Differences in names corresponding to the same email address have been categorized in [29] as follows: ordering (Rajesh Sola, Sola Rajesh), mis-spelling/spacing (Rene Engelhard, Fene Engelhard), diacritics (Demurget, Demurget), transliteration ($\gamma\iota\omega\rho\sigma\gamma\zeta$, Georgios), nicknames (Jacob "Ulysses" Berkman, Jacob Berkman), punctuation (J.A.M. Carneiro, J A M Carneiro), middle initials (Daniel M. Mueth, Daniel Mueth), middle names/patronyms (Alexander Alexandrov Shopov, Alexander Shopov), additional

surnames (Carlos Garnacho Parro, Carlos Garnacho), incomplete names (A S Alam, Amanpreet Singh Alam), diminuitives/variants (Mike Gratton, Michael Gratton), irrelevant information incorporated in the name (e.g. the name of the project: Arturo Tena/libole2, Arturo Tena), user name instead of names (mrhappypants, Aaron Brown), artifacts of the tooling used by developers when committing/storing/migrating data (e.g. timestamps "(16:06) Alex Roberts", or commit messages in addition to names, "Fixed a wrong translation ib ja.po. T.Aihana"). The differences can be one of these or can be combinations of the above.

2.3.3 Machine Learning for Identity Merge

With increasing adoption of Machine Learning (ML) in various domains, the modern methods for record linkage include unsupervised approaches, semi-supervised learning approaches, and supervised learning approaches. In a research by [68], the authors evaluate some work using unsupervised, rule and threshold based approaches [18, 57], some semi-supervised learning algorithm trained on statistically generated artificial labels [36] and provide the first **supervised** learning approach for the United States Patent and Trademark Office (USPTO) database (United States Patent and Trademark Office – an online database of all patents issued in the United States). Rule and threshold based approaches require leveraging expert knowledge which may only belong to domain experts.

Disambiguation methods are needed to track inventors and assignees across their patents or link their information to other data sources as inventors and assignees in the USPTO database are not given unique identification numbers. The supervised method discussed in this work uses two extensive sets of labeled inventor records as the training data and the random forest algorithm to disambiguate inventors. This approach is shown to be effective on data sets such as USPTO which has an ample amount of hand labeled inventor records (over 150,000) to act as a training data set for supervised learning.

However, it is difficult to gather validated ground truth data in the case of operational data and this inadequacy in availability of ground truth for our data set of developers

from projects hosted on GitHub causes a hindrance to employing any supervised learning approach directly. Past research on de-duplication of authors in citations [54] has leveraged a technique called active learning, which starts with limited labels and a large unlabeled pool of instances, thereby, significantly reducing the effort in providing training data manually. The active learning method uses an initial classifier to predict on a very small number of labeled instances. This is a challenging task in a large collection of data, since random sampling of data for manual labeling may not bring out the best cases that have subtleties from which a classifier can learn. Therefore the active learner which is a set of initial classifier brings out these cases through generating a confusion region – where there are confusions between outcomes of the classifiers. This confusion region can be extracted and manually labeled for it to serve as the training data for the actual classifier. A study [62] on de-anonymizing social network profiles show that browsing history can be linked to social media profiles such as Twitter, Facebook or Reddit accounts by analyzing distinctive patterns in the browsing data. A user's likelihood of visiting a URL is governed by the URL's overall popularity and whether the URL appeared in the user's Twitter feed. Next, for each user, their likelihood of generating a given anonymous browsing history is computed. Finally, the user most likely to have generated that history was identified. According to this study, the user's web browsing behavior is controlled by two parameters: The recommendation set (a set of links appearing on the user's feed by the user's friends etc. on the network) and the recommendation factor (the user's responsiveness to the recommendation set). The browsing history can by equated to the files touched by a developer in our case and we may be able to use the likelihood estimates as described in this paper to group IDs.

2.3.4 Scalability in Record Linkage

One of the greatest obstacles as discussed by [61] is the scalability of approaches to solve the record linkage or entity resolution issue. Brute force requires all-to-all comparisons which can become prohibitive for moderate sized data sets. To overcome this, a method called "blocking" is used, "which involves partitioning data files into 'blocks' of records

and treating records in different blocks as non-co-referent a-priori. The blocking methods used range from very basic ones that pick certain fields (e.g. geography, or gender and year of birth) and places record in the same block if and only if they agree on all such fields, to highly application-specific ones based on placing similar records into the same block, using techniques of "locality-sensitive-hashing" (LSH). As blocking divides records into mutually exclusive blocks, records only within the same blocks can be linked. This requires expertise and domain knowledge to pick out reliable and error-free fields for blocking. Such fields can also be unreliable for many applications, therefore blocking may miss large proportions of matches.

This research clearly shows that much work is left in areas of scalable record matching, that is also generic and doesn't require domain expertise to perform record linkage.

2.4 Resources

In this section, we discuss the research related to various techniques, packages and tools that we use in our work. In the following subsections, we go over a vector embedding technique used in finding textual similarity between two people during the disambiguation and de-anonymization process, the active learning technique that can help us minimize the number of hand labeled instances required for supervised learning, and the record linkage package which is extensively used in this research to compare strings representing author identities.

2.4.1 Vector Representation of Documents

Previous work to identify anonymous authors via "linguistic stylometry" on social network data show that humans have their own style of writing [45]. Therefore, in this research, we propose a method to correctly identify individuals using text as one of the markers of their identity. As the vector representation of paragraphs (Doc2Vec) is inspired by the work in learning vector representations of words (Word2Vec), we start off

by discussing in brief how the latter uses neural networks in unsupervised learning to create a vector space for words.

Word2Vec

Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct the semantics of words. Word2Vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.

After training, the vector for a word produced by the Word2Vec model, represent that word's relation to other words. Word2Vec can utilize either of two model architectures to produce a distributed representation of words: *continuous bag-of-words* (CBOW) or *continuous skip-gram*. In the continuous bag-of-words architecture, the model predicts the current word from a window of surrounding context words. The *bag-of-words* assumption indicates that the order of context words does not influence prediction. In the continuous skip-gram architecture, the model uses the current word to predict the surrounding window of context words. The skip-gram architecture weighs nearby context words more heavily than more distant context words. Figure 2.1 illustrates the architecture of these models.

Doc2Vec

Doc2vec (or paragraph2vec) modifies the Word2Vec algorithm to unsupervised learning of continuous representations for larger blocks of text, such as sentences, paragraphs or entire documents. In the Doc2Vec architecture, the corresponding algorithms are *Distributed memory* which considers the word order and *Distributed bag of words (DBOW)* which ignores the context words in the input, but forces the model to predict words randomly sampled from the paragraph in the output. These concepts have been

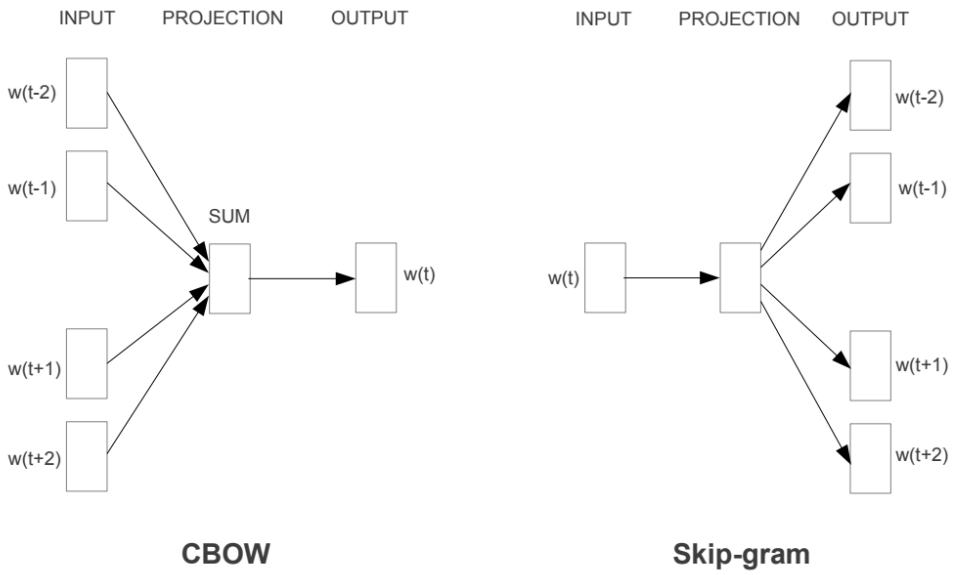

Figure 2.1: The CBOW and Skip-gram architecture used in creating word embeddings

Taken from "Efficient Estimation of Word Representations in Vector Space" (Thomas Mikolov, Quoc Le). The CBOW architecture predicts the current word based on the context, and the skip-gram predicts the surrounding words given the current word.

implemented in a open-source library "Gensim" which has been used to generate document vectors in this research.

Gensim's implementation of Doc2Vec

Gensim started off as a collection of various Python scripts for the Czech Digital Mathematics Library in 2008, where it served to generate a short list of the most similar articles to a given article [52]. Through time, Gensim improved its efficiency and scalability to become a robust software to perform unsupervised semantic modelling from plain text. Gensim is licensed under the OSI-approved GNU LGPLv2.1 license and is free for both personal and commercial use. Because of Gensim's robustness, efficiency and most importantly scalability, we chose this implementation of generating paragraph vectors for our research.

2.4.2 Active Learning

Many existing methods for record matching rely on hand-coded rules, if-else conditions and thresholds to determine whether two records are a match. Machine learning algorithms reduce the tedium of hand-coding by relegating the task of finding the de-duplication function to a learner from a intelligently crafted data set. However, the success of such algorithms are dependent on providing a training set that covers 'duplicates and non-duplicates that will bring out the subtlety of the functions'. These examples are hard to find as the best examples to teach a learner may be spread far apart in the data set and will therefore require extensive search. For example, finding a set of non-duplicates that may be mistaken as a duplicate can be a crucial observation in a training set that might significantly affect the performance of a classifier. These problems are addressed in [54, 13] where the construction of the deduplication function is automated using training sets that include the challenging training data. This is called the active learner. An active learner builds several redundant de-duplication functions and exploits the disagreement among them to discover inconsistencies among duplicates in the dataset. This automates the challenging task of identifying potentially confusing record pairs from a large data set, thereby significantly reducing manual effort as the user now has to hand-label the selected pairs only. This not only reduces manual effort but also increases accuracy of the learning method by providing it with a diverse set of data.

2.4.3 Record Linkage Package

The `RecordLinkage` package (available on CRAN and R-Forge) deals with detecting homonyms and synonyms in the data. When dealing with data from different sources, homonym and synonym error is likely to occur. In this paper [55], talk about dividing the various vast number of methods, to de-duplicate data files, into two broad classes. One class consists of stochastic methods and the other non-stochastic methods from the machine learning context. In this paper, the authors outline the capabilities of the `RecordLinkage` package such as using blocking (reducing data pairs by joining two records that coincide on a specified field i.e. the first name or the last name etc.) or using

phonetic functions and string comparisons (through implementing the Jaro-Winkler or Levenshtein distance for string similarity etc).

This research also discusses machine learning methods with the premise that record linkage can be understood as a classification problem. Therefore, machine learning procedures such as clustering methods, decision trees or support vector machines are usable for linking personal data. The records linkage package implements both unsupervised (k-means and bagged clustering) and supervised classification for record matching. Even though the unsupervised method is attractive as they do not require training data, the authors note that the quality of the resulting classification can vary significantly for different data sets and might yield unsatisfactory results.

As this tool was developed mainly for empirical evaluation of record linkage methods, it gave us some background on existing procedures that have been implemented to solve the identity-linking problem. However, similar to the other studies, these techniques heavily rely on the visible user attributes from their profiles, which may be insufficient for our data.

2.5 Research Gap

Even with a plethora of research that has been done in this field, there are a number of gaps that need to be bridged. In these section, we iterate over what these shortcomings are and propose approaches that address these and that could be combined with other, more specialized approaches, especially for resolving homonyms more precisely. For example, the productivity outlier detection and reallocation approach [82] detects when the number of commits or changes is highly unusual and distributes the authorship to other committers. Such an approach would help to both identify homonyms and redistribute authorship to each developer.

Most of the approach used in previous research can only link users who provide an explicit bridge to other accounts or with common attributes such as user name or email. This is inadequate when we want to match users who may not have common attributes across accounts. The ID of an user is based on the local .git configuration file of the device

(computer, laptop, server etc.) from which the user commits[2]. Commits may also be not linked to any user in case of the following reasons:

1. Unrecognized author (with email address) - When the user's address has not been added to the account settings. The user's email address can be added to GitHub's email settings, however the user may choose different email addresses from work or home.

2. Unrecognized author (no email address) - If the user used a generic email address that can't be added to the user's email settings. The commit email address can be set in Git, to link your future commits, however, old commits will not be linked.

3. Invalid email - This means that the email address in the user's local Git configuration settings is either blank or not formatted as an email address. Again, the email can be updated in Git Configuration but old commits will not be linked.

When the common source of developer identity data i.e. the string representing a developer in code commits of version control systems is missing, linking users even within one platform, such as Git, becomes a tedious and non-trivial problem which cannot be solved using "surface" attributes. While significant amount of work has been done addressing what we call the `Synonym` error, when multiple strings represent the same developer, we found no studies trying to address the `Homonym` errors, where multiple developers are identified by the same author string in a commit. In this work, we aim to outline better solutions to correcting `Synonym` errors that yield better accuracy as well as solutions to `Homonym` errors.

Past record-matching or disambiguation techniques done on patent or citation records involved supervised learning, and have proven to be more accurate than manually derived rule and threshold based approaches. However, supervised approaches require a large number of 'good' training observations that can be used by the learner to derive the best disambiguation function. As discussed in Section in 2.3.3, disambiguation done on the USPTO record successfully used supervised learning owing to over 150,000 manually

[2]https://help.github.com/articles/why-are-my-commits-linked-to-the-wrong-user/

labeled records as used as the training data. This approach is hard to be replicated on operational data provided that it is expensive to generate such carefully-crafted large data sets for training.

To overcome this, we adapt *Active Learning* discussed in 2.4.2 . We select a minuscule subset of the entire data, create all pairs between them and manually label them as matches or non-matches. These are used to build preliminary classifiers. The disagreements in the prediction of these classifiers are selected to be carefully hand-labeled again to be used as the training data for the actual classifier. This ensures that data that is most likely to be confused by the learner are captured in the training data making the learning process more effective despite reducing manual effort.

The need for correct identification of individuals in any network remains undisputed as it plays a critical role in the analysis of these networks that can yield dependable results. Therefore, a complete systematic approach to solving this problem is important in spurring research and help in commerce. In this work, we also aim to fill in a void for a much needed framework that can be customized and extended to solve this problem.

Chapter 3

Data Collection and Classification

3.1 Data Collection and Storage

Mining Git repositories is a complex task and this is made even more difficult when it is done on a large scale [24]. As millions of projects are developed on open-source collaborative platforms like GitHub, Bitbucket, GitLab, SourceForge etc, data retrieval, organization and storage requires massive undertaking. This chapter dives into the details of the data retrieval and storage process for this research. We collected data for this work from various platforms between 2005 and 2018, over a span of 13 years. In this work, we particularly focus on projects hosted on GitHub, which is a web-based Git repository hosting service. It offers both plans for private repositories and free accounts, which are usually used to host open-source software projects. As of December 2018, GitHub has more than 28 million users and 62 million repositories which includes 28 million public repositories, making it the largest host of source code in the world. This also gives us access to a the largest collection of developer IDs in Open Source Projects. The following sections describe how data is stored in Git, the process of discovering projects, retrieving information, storing and cleaning the data. More information and detailed explanation of data retrieval and storage can be found in [39].

3.1.1 Git Overview

Git is a free and open source distributed version control system, designed by Linus Torvalds in 2005, to track source code changes during the process of software development. It can handle projects of any size with speed and efficiency[1].

How Git Stores Data

Git is a content-addressable file-system. This means that Git has a key-value data store that allows the user to insert any content into a Git repository and provides the user with a unique key to later retrieve that data. This unique key is a 40 character checksum hash - the SHA-1 hash[2] (Secure Hash Algorithm 1), calculated based on the content of that object. Git contains three types of objects: blobs, commits and trees.

- **Blob object**: A Git *blob* (Binary Large Object) is the simplest object, which is often a file, but it can sometimes be a symlink as well. It is the object type used to store the contents of each file in a repository (does not store the filename). The file's SHA-1 hash is computed and stored in the blob object. These endpoints allows the user to read and write blob objects to the user's Git database on GitHub.

- **Tree object**: Directories are represented by a *tree* object. They refer to blobs that have the contents of files (filename, access mode, etc) and to other trees representing subdirectories. A single tree object contains one or more entries, each of which is a SHA-1 hash of a blob or subtree with its associated mode, type and filename.

- **Commit object**: A *commit* object specifies the top-level tree for the snapshot of the project at that point; the author/committer information(which uses the name and email configuration settings and a timestamp), a blank line and then the commit message. If there are more than one commit, it means that the commit is a merge. If there are no commits it means that it is an initial commit [69].

Figure 3.1 shows the various Git objects and how they are related.

[1] https://git-scm.com/
[2] http://eagain.net/articles/git-for-computer-scientists/

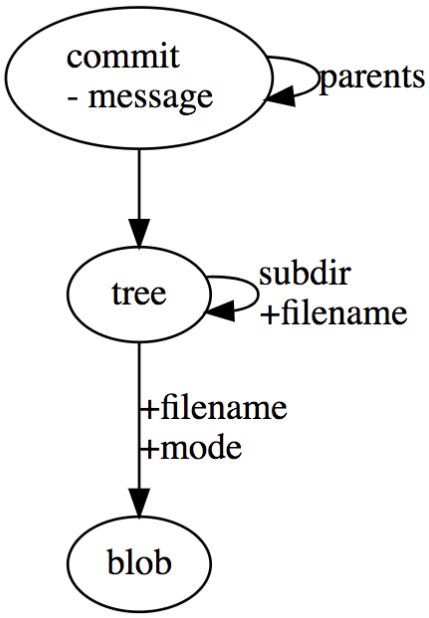

Figure 3.1: Structures within Git: The Blob, Tree and Commit

Taken from "Git for Computer Scientists" and shows the structures within Git.

3.1.2 Data Collection

The Data Collection Process can be decomposed into 3 broad steps [39]:

1. **Data Discovery**: We start off the data collection process by discovering forges hosting version control systems and open source projects. For example, SourceForge, Savannah supports CVS, SVN and Git and have a large collection of open source projects. Next, we discover a list of large and well- known projects like Linux kernel, Gnome, KDE and Mozilla. These projects also have mini-forges involving projects related to the main project. Open-source forges like Bitbucket and GitHub usually provide rest API to discover the list of projects. Project discovery can be done through:

    - Using Search API: APIs are specific to each forge and come with various caveats. Most APIs tend to limit discovery for user or IP address.

- Using Search Engine: Search engines, such as Google or Bing, can supplement the discovery of open source project repositories on collaborative forges, when the forge does not provide an API or it is non-functional. It should be noted that this process is not thorough in discovery.
- Keyword search: Some forges provide keyword based search of public repositories, which is a complimentary approach when a forge does not provide APIs for the enumeration of repositories and the results returned from search engines are lacking.

The discovery phase outputs a list of git, mercurial, svn URLs. The list is subdivided so that each sub-list contains URLs that would result in approximately 2TB of data when downloaded. We only select the URLs that have been updated or have not been previously selected.

2. **Data Retrieval**: Data retrieval can be done in parallel on a large number of servers with substantial network bandwidth and storage. All the latest version control systems (Git, Mercurial, Bazaar) have clone functionality to create a replica of the repository. With over 62 million repositories, data retrieval is a massive undertaking - a single thread shell process on a typical server CPU with no limitations on network bandwidth can clone 20,000 to 50,000 repositories in 24 hours (this drastic variation is dependent upon the size of the repository or forge). Cloning is done on a Newton cluster for 2TB chunks at a time. These chunks are then archived using modified pax files (as tar and cpio have maximum file size restrictions). The pax files are currently stored offline on 4/8/10TB disks.

3. **Data Extraction**: Code changes are organized into commits that can change one or more source code files within a project. Once a repository is cloned, the Git objects (discussed in Section 3.1.1) are extracted from each repository and store these in a single database. The collected data is tabularized and with each row including an ID, the commit hash, the committer name and email, the timestamp of commit, file and source. This process of extraction is performed on a cluster (36 nodes, 16 cores,

256 GB memory) and takes approximately two hours for a single node to process 50,000 repositories.

4. **Data Storage**: The whole collection of Git repositories may replicate the same Git object hundreds of times and therefore can occupy vast amount of storage space. This redundancy is not only problematic space-wise, but also makes performing analytics on this data an even more difficult and unnecessarily computationally expensive task. To avoid such redundancies, all Git objects are stored in a single database. The details of data storage are explained in [39] with more details.

Figure 3.2 shows the various stages of data - from discovery to extraction and organization. Maps of relations were created and stored for fast access of relevant data. Figure 3.3 shows the fields that were explicitly recorded for this research - the name comprising of the first and last name, an email address, the time in which the commit was made, a commit hash which is unique to each commit, a set of files that was changed in the commit, a set of projects that was changed in the commit and finally the commit message which is a text explaining the changes made in the commit.

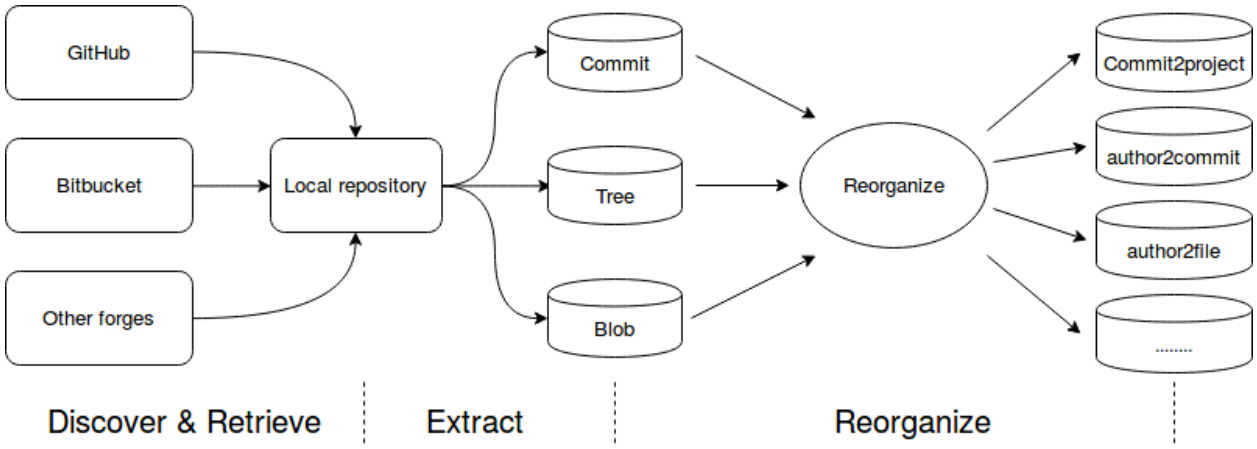

Figure 3.2: Data flow: Data discovery and organization

Taken from "World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data" (2019). Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, Audris Mockus. [39]

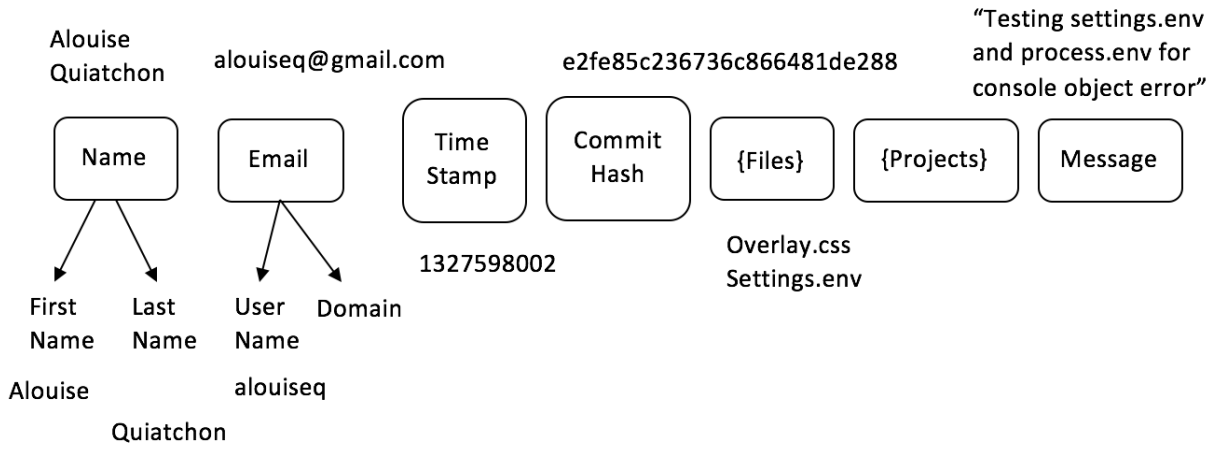

Figure 3.3: Extracted data from a commit

3.2 Classification of Errors

In order to improve existing identity resolution approaches or create new ones, we need a clear understanding of the nature of the errors associated with the records related to identities. Here, we look at some of the types of errors that have been previously defined for record matching and outline how errors in developer identities derived from operational data varies from the predefined errors.

3.2.1 Errors Classified in Existing Literature

Studies done on census data show that common errors may be typographical, a variation in the phonetic spelling of a name, or the reversal of the first and last names, among others. In certain cases, correctly spelled identical names may belong to different persons, for example, 'John Smith'. Different domains have to their disposal a range of information that can act as quasi-identifiers such as date of birth or addresses in the case of census data. However, date of births may also be inadequate to uniquely identify a person since there can be multiple "John Smith's" with the same birth date.

A study on the personal name comparison [12], discusses the characteristics of names and their variations. Even names within the English speaking world can have different forms for various reasons. People change their names over time (the most common

case is when getting married). In many cases nicknames are often used or are given to people - these can be the short forms of their names such as 'Liz' for 'Elizabeth', 'Bob' for 'Robert' etc. Other studies [29] have identified errors as a result of transliteration, punctuation, irrelevant information incorporated in names, etc. Errors on general (not just for names) have been classified in [30] as (1) Typographical errors where a person makes a typing error but is aware of the actual word (example 'Sydnye' instead of 'Sydney'), (2) Cognitive errors that arise from misconceptions or lack of knowledge (example, 'Sydny' instead of 'Sydney') and finally (3) Phonetic errors that comes from spelling the actual word based on how it sounds (example, 'Sidny' instead of 'Sydney'). An error can arise because of one of the above issues or a combination of them and can alter names significantly. This makes name matching based on string similarity a daunting task. Therefore, various matching techniques are employed to resolve these issues - phonetic encoding i.e. soundex algorithms [26, 31] or pattern matching of strings i.e. Levenshtein or Edit distance [46], Jaro-Winkler distance [81] or a combination of both phonetic and pattern matching.

3.2.2 Errors in Developer Identities

Similar to identity errors in census and other data, developer identities have spelling errors, transliteration, variation in order of the names etc. Here, we define a developer identity string as a combination of the developer's name and email address Name <email>, for example John Smith <john.smith@domain.com>. This is the same as how the developer information is stored in a Git commit. The name and email of the developer is stored in a Git configuration file of the specific device a developer is using at the moment. Once Git commit is recorded, it is immutable like other Git objects, and cannot be changed. Once a developer pushes their commits from the local to remote repository, that author information remains. A developer may have multiple laptops, workstations, and work on various servers, and it is possible and, in fact, likely, that on at least one of these computers the Git configuration file has a different spelling of their name or email or at times both. Furthermore, complications are, at times, introduced by

the use of tools where tools allow for default placeholders to remain at the time of commit, some Git clients may provide a default value for a developer as well, for example, the host name. As a result, the same developer can end up being represented as multiple identities associated with commits. Sometimes developers do not want their identities or their email address to be seen, resulting in intentionally anonymous name, such as, "John Doe" or email such as "anonymous@localhost". Similar to other scenarios, developers may change their name over time, for example, after marriage, creating ambiguity.

In order to design solutions to correct these errors, we need an procedural examination of the errors. First we inspect randomly selected developer IDs to understand the nature of these errors and why they occur. We then inspect the most common names or user names, keeping note of anything unusual found in names and emails. We also came across many additional types of errors when we manually labeled our data in the active learning phase as we discuss in Chapter 4. The resulting anomalies from developer identities were grouped using the open-card-sort method [3] and resulted in two primary categories: `Synonym` and `Homonym` errors.

Synonyms

`Synonym` errors are introduced through when an individual uses IDs that vary through spelling mistakes, capitalization (or absence) of names, introduction of a middle name, last name change due to marriage, abbreviation of a name, adding extra space(s), adding period, reversal of first and last names, transliteration of non-ascii characters, irrelevant information incorporated into names. These errors can arise from one or a combination of the above cases and are introduced when a person uses different strings for names, user-names or email addresses. For example, "utsav dusad<utsavdusad@gmail.com>' and "ut-savdusad<utsavdusad@gmail.com>" are identified as synonyms. Spelling mistakes such as "Paul Luse<paul.e.luse@intel.com>" and "paul luse<paul.e.luse@itnel.com>" are also classified as `Synonyms`, as "itnel" is likely to be a misspelling of "intel". `Synonym`

[3] Card sorting is creating a set of 'cards' and asking reviewers to sort them in categories that makes sense to them. Open card sort allows participants to create and label their own categories.

IDs typically contain components that carry some information about the identity of the individual.

Homonyms

`Homonym` errors are introduced when an individual provides or uses IDs that cannot be tied to a single individual. For example, these may be identifications related to generic roles ("Admin", "root", "dev'), names of projects ("Jenkins", "Travis CI", "Ubuntu', "Openstack", "Vagrant"), names of organizations ("cisco', "cmart", "walmart"). The IDs may contain components that seeks to preserve anonymity or are simply placeholders injected by tools ("nobody", "your name", "test", "anonymous", "me", "JohnDoe"). Other examples include miscellaneous terms such as "Bot", "EC2Users", "Server" etc. It may also occur when multiple individuals use a single generic ID, for example, the ID "saper<saper@saper.info>" could be used by multiple individuals in the organization. For instance "MarcinCieslak<saper@saper.info>' is an individual who may have committed under the above organizational alias. Missing data is also categorized under `Homonym` errors – such as when an individual leaves the name or email field empty, for example, "chrisw<unknown>". Contrary to `Synonym` IDs, `Homonym` IDs do not reflect the identity of a single individual and, therefore, is harder to associate to an individual. Figure 3.4 illustrates a summary of the two broad categories of errors and their subcategories along with some examples.

The ten most frequent components (full name, first name, last name, user name and email) of developer identity is shown in Table 3.1. A large number of components such as 'unknown', 'nobody, 'root, 'Administrator are a result of `Homonym` errors. It is important to understand the difference in nature of these two kinds of errors because they require distinctly different approach in resolution.

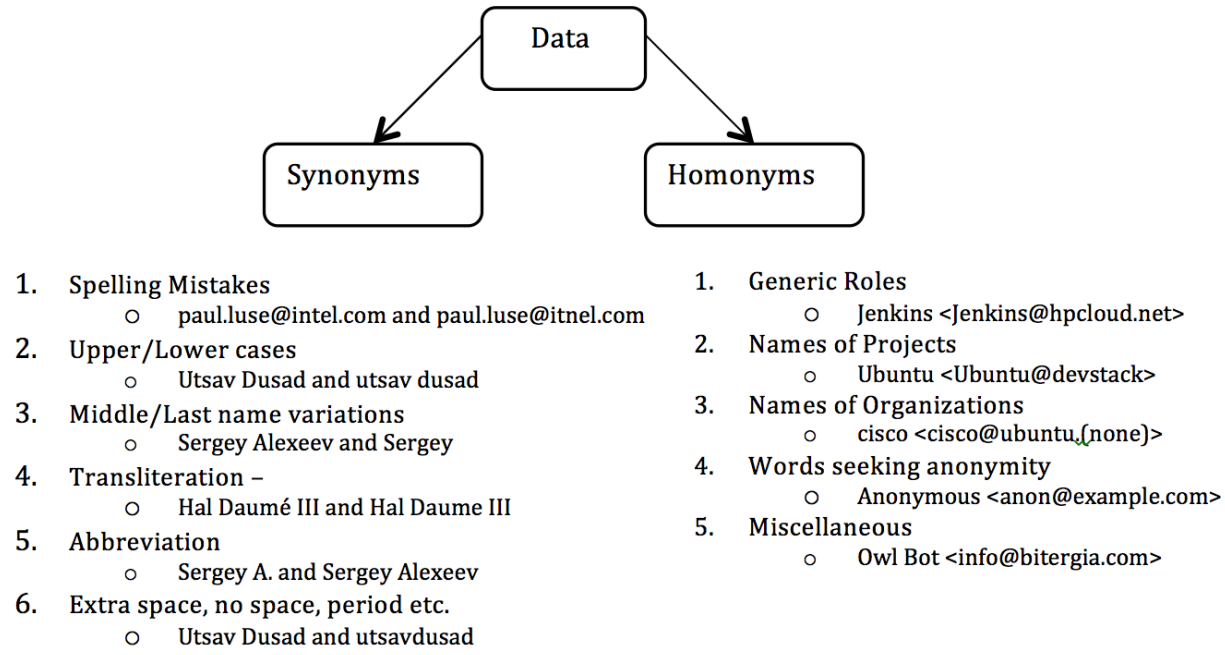

Figure 3.4: Types of errors in developer identities discovered through open card sort

Table 3.1: Data Overview: The 10 most frequent components of developer IDs. A large number of the most frequent components are a result of `Homonym` errors.

Name	Count	First Name	Count	Last Name	Count	Email	Count	User Name	Count
unknown	140859	unknown	140875	unknown	140865	<blank>	16752	root	72655
root	66905	root	66995	root	67004	none@none	9576	nobody	35574
nobody	35141	David	45091	nobody	35141	devnull@localhost	8108	github	19778
Ubuntu	18431	Michael	40199	Ubuntu	18560	student@epicodus.com	5914	ubuntu	18683
(no author)	6934	nobody	35142	Lee	10826	unknown	3518	info	18634
nodemcu-custom-build	6073	Daniel	34889	Wang	10641	you@example.com	2596	<blank>	17826
Alex	5602	Chris	29167	Chen	9792	anybody@emacswiki.org	2518	me	14312
System Administrator	4216	Alex	28410	Smith	9722	=	1371	admin	12612
Administrator	4198	Andrew	26016	Administrator	8668	Unknown	1245	mail	11253
<blank>	4185	John	25882	User	8622	noreply	913	none	11004

3.3 Data Overview

The rest of the work deals with the commit messages extracted from Git over the span of roughly 13 years (2005 to 2018). We were left with approximately 865 Million commits after the data collection process. The number of unique projects discovered and extracted was roughly 25 Million (25,343,808) from github.com and bitbucket.org. The unique number of developer IDs (combination of names and emails) found was about 23 Million.

To provide a better understanding of the data set, we generate a number of statistics. We found that 44.92% IDs have less than 3 commits and 7.3% IDs have more than 100 commits. 62.8% of the names of developers are single words and 36.7% of the developer names are 2-4 words. The average length of commit messages is 12 words with 75% of the messages being less than or equal to 9 words. The average number of unique words per commit message is also 9 words. However, not all of these messages are useful in our context as there may be duplicate template messages. This data set also includes developers that have committed an insufficient number of times for the Doc2Vec algorithm to work effectively. Therefore, we filter this data to exclude commits and developer IDs that may adversely affect the performance of the learning algorithm and with the purpose of getting a cleaner and usable set.

The next chapter outlines the methods for correcting the errors as well as evaluating the performance of the proposed method by comparing it to a commercial effort for disambiguating author identities as well as a recent research method.

Chapter 4

Methods for Data Correction

4.1 Overview

In this chapter, we talk about methods to correct `Synonym` (disambiguate) and `Homonym` (de-anonymize) errors in operational data. We use some of the traditional record matching techniques for `Synonym` resolution as we have the IDs[1] of individuals that can be potentially linked. However, the case for `Homonym` errors is quite different as, it includes missing data and multiple individuals using a single ID and almost always carries no information about the identity of the person. Therefore, the exogenous methods defined in [29] are inadequate for de-anonymization. We use similar approaches - behavioural fingerprint - to address each of these problems and design some experiments using supervised learning to prototype our method. The methods outlined in this chapter are used to design the proposed framework ALFAA.

4.2 Disambiguation: Correcting Synonyms

As defined in Chapter 3, `Synonyms` are created from the variations in developer IDs due to misspellings, case differences, middle or last name variations, transliteration, abbreviations and differences in characters such as space, period etc. or a combination

[1]We refer to IDs as a representation (a string of characters) of an individual, such as a developer, on a database

of these. This results in a single developer being represented by multiple aliases IDs. To resolve `Synonym` errors, we assume that each developer we consider for the disambiguation process will have at least one ID that is uniquely representative of that individual. These IDs may contain one or multiple components including first name, last name (one or both) and email address (user name and some domain). We describe the workflow of ALFAA using our disambiguation algorithm in the following subsections through outlining the process we implemented on developers of the OpenStack projects hosted publicly on GitHub.

4.2.1 Disambiguating OpenStack Developers

As the full set of authors extracted is too large to experiment with and validate, we set out to find a subset of this data that includes a sizable set of projects. We select the developer IDs from the OpenStack ecosystem[2] to serve as the data set to build our prototype, conduct experiments, and perform evaluation. This set of open-source projects fits our requirement in prototyping based on ALFAA for the following reasons.

1. The data set has an adequate number of developers and projects to design and test our proposed methods.

2. The number of developers in this data set can be processed in a reasonable amount of time using a reasonable amount of computational resource.

3. Disambiguation has been implemented by a commercial firm, Bitergia[3], for the OpenStack foundation, and therefore, this data set is a suitable candidate to evaluate our research method by comparing it against existing state-of-the-art industry standards.

We discovered 1,294 projects that are currently hosted on GitHub and have 16,007 distinct developer IDs associated with the commits in the OpenStack ecosystem. ALFAA's

[2]The OpenStack project is a global collaboration of developers and cloud computing technologists producing the open standard cloud computing platform for both public and private clouds. It lets users deploy virtual machines and can handle different tasks for managing a cloud environment on the fly - https://www.openstack.org, https://opensource.com/resources/what-is-openstack

[3] Bitergia mapped multiple developer IDs to an unique identifier representing a single developer as well as mapping contributors to their affiliated companies

disambiguation process, which is reliant upon supervised classification, is done in three phases as shown in Figure 4.1.

1. **Define predictors** - Define the features that will be used by a supervised classifier for classification. In this case, we compute string similarity, frequency similarity and behavioral similarity.

2. **Active learning** - Find a small but effective set to be used as training data for the classifier. This is done in order to reduce manual effort in labeling, yet find instances for the training set that encapsulate the subtleties of the data critical to the training of the classifier. In this case, we use a preliminary classifier to extract a small set from the large collection of data and manually generate labels for final classification.

3. **Classification and Clustering** - Use the defined features and manually labeled outcome as training data to perform supervised learning and classification on pairs of IDs. Compute transitive closure on pairs and create clusters representing each real developer and, finally, manually dis-aggregate incorrectly clustered IDs.

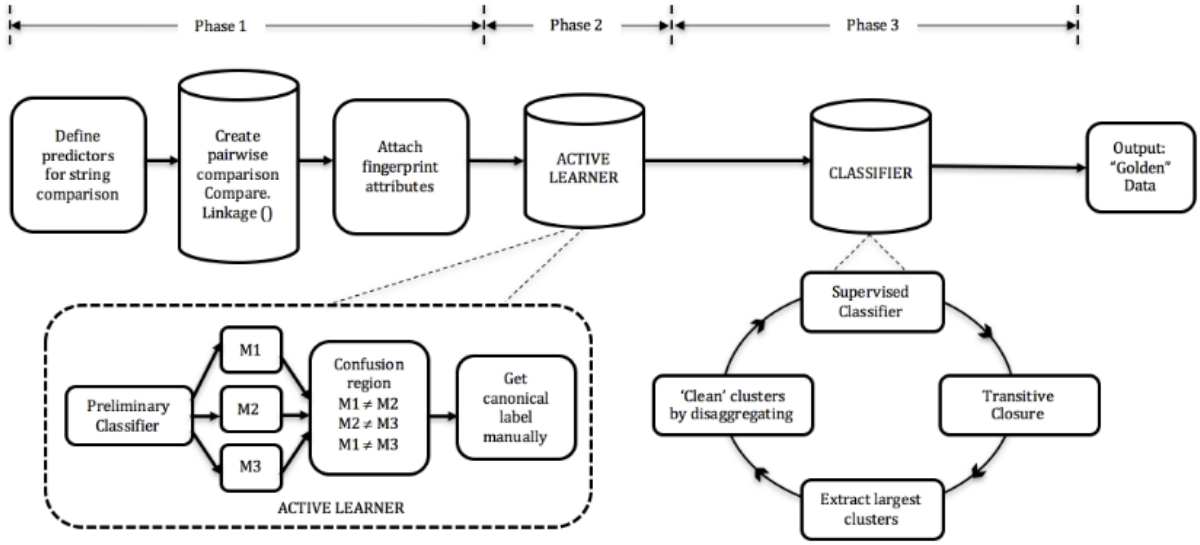

Figure 4.1: Flowchart of the three phases of the disambiguation process of ALFAA: defining predictors, active learning and classification.

To begin, we follow traditional record linkage methodology and identity linking in software [5] as discussed in details in Chapter 2. We first split the information in the developer ID string into its components and define similarity metrics for all developer pairs. We use the string similarity between components as features in our disambiguation process. We also incorporate the term frequency measure for each of the attributes in a pair, to emphasize more on the less common components of IDs. We introduce *behavioral fingerprints* to be used as additional features for disambiguation besides string similarity. Finally, these features are used as input to a supervised learning method to predict whether each developer ID pair represents the same individual. The details of this process are outlined in the following subsections.

4.2.2 Phase 1: Define Predictors

As shown in Figure 4.1, Phase 1 consists of defining predictors for comparison between pairs of components of developer IDs. We first compute string similarity and then define and compute the behavioral fingerprint predictors.

String Similarity Measures

As discussed in Chapter 3, each developer ID consists of the developer's name and email and is stored in the following format - "name <email>", e.g. "Example User <username@example.com>". In certain cases, one or more of these fields may be empty, as defined in `Homonym` errors. We break down each ID and define the following components for each user.

1. Developer ID: String as extracted from source as shown in the example above
2. Name: String up to the space before the first "<"
3. Email: String within the angle ("< >") brackets
4. First name: String up to the first delimiting character i.e. space, "+", "-", "_", ",", "." and camel case encountered in the name field

5. Last name: String after the last delimiting character i.e. space, "+", "-", "_", ",", "." and camel case encountered in the name field

6. User name: String up to the "@" character in the email field

In the case where there is a string without any delimiting character in the name field, the first name and last name are replicated. For example, "bharaththiruveedula <bharath_ves@hotmail.com>" would have "bharaththiruveedula" replicated in the name component, the first name component and the last name component. Additionally, a field called "inverse first name" was introduced whereby the first name component of one ID in the pair was compared to the last name component of the other ID in the pair. We introduce this field to make sure that ALFAA is able to capture the cases where developers reverse the order of their first and last names.

The next step is to calculate the similarity (or difference) between each pair of components extracted from the developer IDs. Studies in the past have used various measures to compare strings. The following are some methods to compute similarity or distance between sequences.

- **Levenshtein distance or Edit distance** is a metric proposed by Vladimir Levenshtein in 1965 [35]. It is defined as the smallest number of single-character operations (insertions, deletions and substitutions) required to change a string to another string.

- **Damerau-Levenshtein Distance** is an extension of the Levenshtein distance by including transpositions among its operations in addition to the three single-character edit operations (insertions, deletions and substitutions). According to Damerau [15], these four operations include 80% of human spelling errors.

- **Jaro Similarity** defined by Matthew A. Jaro in 1989 [28] is calculated using the number of common characters (that are within half the length of the longer strings)

and the number of transpositions. It is defined as

$$sim_j = \begin{cases} 0, & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{otherwise} \end{cases}$$

where $|s_i|$ is the length of string s_i, m is the number of matching characters and t is half the number of transpositions.

- **Jaro-Winkler Similarity** is a variant of the Jaro algorithm proposed in 1990 by William E. Winkler . It improves the Jaro algorithm by applying ideas based on studies [50] that found fewer errors typically occur at the beginning of names. The Jaro-Winkler similarity modified the Jaro similarity so that differences at the beginning of the string has more significance than differences at the end. It is defined as

  $$sim_w = sim_j + lp(1 - sim_j)$$

  where l is the length of a common prefix at the start of the string up to a maximum of four characters and p ($<= 0.25$) is a scaling factor for how much the score is adjusted upwards for having common prefixes.

In the exploratory phase of implementation, both Levenshtein and Jaro-Winkler similarity was calculated for each author pair as seen in previous studies [5, 29], which are standard measures for string similarity. To do this, we used an existing implementation of the measures in the RecordLinkage [55] package in R, namely the levenshteinSim() and jarowinkler() functions were used. In this phase, we found that the Jaro-Winkler similarity produces better scores which are more reflective of similarity between developer strings than the Levenshtein score (we manually scanned though a list of IDs and scores computed). We, therefore, decide to use this measure in the to create similarity scores between components of author IDs for our study.

Adjustment Factors for String Frequency

Greater confidence in matching can be achieved if two IDs share components that are uncommon compared to sharing components that are common such as "John". In addition, components that do not carry any information about authorship of commits and are used by numerous individuals such as "nobody" or "root' (as a result of homonym errors) should also be disregarded from consideration in the similarity detection process. This extra information, if properly encoded, could be exploited by a machine learning algorithm making disambiguation decisions. We, therefore, incorporate frequency adjustment for ID components in ALFAA. To do this, we count the number of occurrences of the attributes for each author as defined in Section 4.2.2 i.e. name, first name, last name, user name and email for our data set. The similarity between developer pairs, developers a_1 and a_2, was calculated for each of these attributes as follows:

$$f_{sim} = \begin{cases} \log_{10} \dfrac{1}{f_{a_1} \times f_{a_2}} & \text{if } a_1 \text{ and } a_2 \text{ are non-fictitious} \\ -10 & \text{otherwise} \end{cases}$$

where f_{a1} and f_{a2} are the frequency of names of authors a_1 and a_2 respectively. To remove the components that carry no information about the authorship, we do the following: A list for each component of a developer ID - names, first names, last names and user names and emails that had 200 most frequent strings in each category was generated from the full data set of authors. The first 10 most frequent strings for each component found here is shown in Table 3.1). The components that were common names but are non-`Homonym` errors (in other words non-fictitious), i.e. names that could truly belong to a person such as "Lee", "Chen", "Chris", "Daniel" etc., were manually removed from each component list. The string frequency similarity of a pair of name or first name or last name or user name was set to -10 if at least one element of the pair belongs to a string identified as fictitious. This was done in order to let the learning algorithm recognize the difference between the highly frequent strings and strings that are not useful as author identifiers. We chose -10 as threshold as we found that the value for other highly frequent terms were significantly greater.

Behavioral Fingerprints

In addition to the spelling of the name and emails, authors of commits might leave their signatures in the way they compose commit messages (in the style they compose text), the files they commonly modify, or the time zones they work in. Fingerprints can be defined as something that uniquely identifies an individual such as a trait, or a trace or characteristic and therefore, these signatures are similar to the concept of fingerprints when combined. To encode these fingerprints or behavioral attributes of developers into ALFAA, we designed three similarity measures as follows:

- Similarity based on files modified — If two developer IDs modify similar sets of files, they are more likely to represent the same person.
- Similarity based on time zone — If two developer IDs commit from the same time zone, it is an indication of geographic proximity, and therefore, there is a higher likelihood that the IDs belong to the same developer.
- Similarity based on commit message text — Two developer IDs sharing writing style and vocabulary increase chances that they represent the same entity.

We operationalize the above behavioral fingerprints as outlined below.

1. **Files modified**: The files that are modified during a commit is extracted from each commit. From this, a map of file to developers can be created, whereby we have a list of developers for each file that was modified. Each modified file is inversely weighted using the number of distinct developers who have modified it (similar to down-weighting common names as evidence of identity). For example, if a file was modified by 5 unique developer IDs, then its weight would be 0.2. This is done with the same reasoning in mind as the concept of Term Frequency - Inverse Document Frequency (TF-IDF) [38, 59] where the weight of the files modified by many developers is diminished and increases the weight of files that are modified by a few as files modified by many developers do not contribute to help in the identification of modifiers. The weights of files is used to compute the pairwise

similarity between developer IDs. The similarity score is computed by adding the weights of the files, W_f, modified by both developers in a pair. In other words, it is the intersection between the set of files modified by developer ID a_1 and set of files modified by developer ID a_2. A similar metric was found to work well for finding instances of succession (when one developer takes over the work of another developer) [43]. The weight of a file is defined as follows where A_f is a set of developers who have modified file f and f_{a1} and f_{a2} are the sets of files modified by each developer.

$$W_f = \frac{1}{A_f}, \text{where } A_f = |a_1^f, ..., a_n^f|$$

$$Sim_{a_1 a_2} = \sum_{i=1}^{n_{a_1 a_2}} W_{f_i}, \text{where } n_{a_1 a_2} = |f_{a_1} \cap f_{a_2}|$$

2. **Time zone**: We discovered 300 distinct time zone strings (due to misspellings) from the commits and created a "developer by time zone" matrix that had the count of commits by an author at a given time-zone i.e. each row of this matrix is an array of the number of commits made by a given developer, each column of the matrix is the number of commits made by all developers in that time-zone. All time-zones that had less than 2 entries (developers) were eliminated from further study. Each author was therefore assigned a normalized time-zone vector (with 139 distinct time zones) that represents the pattern of his/her commits. Similar to the previous metric, we weighted each time zone by the inverse number of authors who committed at least once in that time-zone. We multiply each author's time zone vector by the weight of the time zone. We define weight of each time-zone W_T and developer a_i's time-zone vector (TZV) as:

$$W_T = \frac{1}{A_T} \text{where } A_T = |a_1^t, ..., a_n^t|$$

$$(TZV_{a_i}^t) = \left(\frac{C_{a_i}^t}{A_T} \right)$$

Here, $(C^t_{a_i})$ is the vector representing the commits of an author a_i in the different time-zones t and (A_t) is the vector representing the number of developers in different time-zones. The pairwise similarity metric between developer a_1 and developer a_2 is calculated using the cosine similarity between two developer vectors:

$$tzd_{a_1 a_2} = cos\_sim(TZV_{a_1}, TZVa_2)$$

where TZV_{a1} and TZV_{a2} are the authors' respective vectors.

3. **Text similarity**: Previous work has been done by [45] to identify anonymous authors via *linguistic stylometry* on social network data, whereby the "the writing style of an anonymous author is compared against a corpus of texts of known authorship". This study, which yielded reasonable accuracy on large-scale data sets, indicates that individuals have their own "style" of writing i.e. in the vocabulary they use and in the way they compose text. Therefore, we consider an individual's written text as one of the markers of his (or her) identity.

   As the third fingerprint to be used as a predictor in ALFAA, we consider the short messages that developers have to write while committing their code to a version control system. Figure 4.2 is an example of a snippet of commit messages and the identity of the developer who wrote it. These messages are typically short (mean commit length for our data set: 12 words with 9 unique words on the average). To compare the styles of written text between developers, we need to be able to embed text written by each developer. As these messages are different in nature and size in comparison to messages on social networks, we conduct a preliminary study to test the viability of using commit messages as a fingerprint. The details of this study is described in Appendix A. As we received a positive outcome from this study, we continue to use the Gensim's implementation[4] of the Doc2Vec [34] algorithm to generate vectors embedding messages written by each developer ID.

[4]https://radimrehurek.com/gensim/index.html

All commit messages for each developer ID that contributed at least once to one of the OpenStack projects were gathered from the collection described above and a Doc2Vec model was built. This generated 200 dimensional vectors for each of the 16,007 developer in the OpenStack data. To compare the vectors for similarity, cosine similarity[5] between these vectors was calculated between each pair of vectors representing IDs of developer a_1 and developer a_2.

4. **Gender Similarity**: Gender is sometimes used as a metric in record linkage literature [71]. We obtain the gender of the users from the commit blob, as one of the following - Male, Female or Undetermined. The similarity between author pairs are determined as follows:

$$gs_{a_1 a_2} = \begin{cases} 0.5, & \text{if } G_{a_1} \text{ or } G_{a_2} = \text{Undetermined} \\ 1, & \text{if } G_{a_1} = G_{a_2} \\ 0, & \text{if } G_{a_1} \neq G_{a_2} \end{cases}$$

where G_i represents the gender of author i. However, we dropped this from our implementation as our data had too many "undetermined" instances for it to be useful to the learner.

```
Alouise Quiatchon;alouiseq@gmail.com;both radial and eyes components are working simultaneously
Alouise Quiatchon;alouiseq@gmail.com;fixed all identical console object error
Alouise Quiatchon;alouiseq@gmail.com;latest build with css of new feature proto
Alouise Quiatchon;alouiseq@gmail.com;fixed box seam to play nice with eyes and radial components
simultaneously
Alouise Quiatchon;alouiseq@gmail.com;added images to nav headers for each page
Alouise Quiatchon;alouiseq@gmail.com;testing settings.env and process.env.NODE_ENV
Alouise Quiatchon;alouiseq@gmail.com;updated gruntfile to reflect new static location
```

Figure 4.2: Example of data used to build Doc2Vec models - a developer (ID) as shown by the first two semi-colon separated fields, and some of the commit messages composed by the developer

[5]Cosine similarity is ideal for our data set (over euclidean or similar distances) as each vector in our data set is formed from varying lengths of text and therefore occurrences of a word in text t_1 maybe more than in text t_2 simply because t_1 is longer and this will therefore reflect in the vector. Cosine similarity corrects for this.

However, there are several potential drawbacks of these distance metrics. For example, high similarity scores for files modified between two IDs may mean that two different individuals are working on the same project thereby editing the same files at alternating times. High text similarity may mean that the developers are working on the same project and therefore may be using similar vocabulary in the commit messages. Fortunately, our approach leaves the decision to include a specific feature to a machine learning algorithm. As we show later in the results, the behavioral similarity measures, in particular, text similarity, are important predictors for disambiguation.

4.2.3 Phase 2: Active Learning

Supervised classification requires ground truth data and which may require manual classification. Manual classification is time consuming, error-prone and may be based upon a human's subjective decision. The nature of our data makes it extremely hard to obtain absolute ground truth and, therefore, manual classification based on gathered information is required to create a training set for supervised classification. To do this, we used two raters (as discussed in details later in this chapter). We found some differences between two raters (provided with same instructions) and further errors discovered in the *Active Learning* phase where the learner indicated manual classification error that was again verified by both raters. Since manual classification of all possible pairs is impossible (assuming 6 pairs can be labeled in one minute, it would require roughly 100 people each working approximately 7000 hours to hand label the existing OpenStack collection of over 256 million pairs), it is necessary to identify a small subset of instances so that the classifier would produce accurate results on the remainder of the data. Randomly sampling pairs of IDs to compare for manual labeling is not likely to work either: in our case, the chance that 2000 randomly selected pairs from 256M author pairs would belong to the same person is close to zero (assuming, on average, two developer IDs per person, 2000 pairs would represent 10^{-5} fraction of the entire sample). Therefore, we need a solution that minimizes manual classification effort, which can be achieved through *Active Learning* [54]. In a nutshell,it is a process where variations in predictions

of a preliminary classifier fitted on different subsets of the classified data are manually resolved. It helps ensure that the pairs that are most likely to be confused by the learner are added to the training data. The expensive manually classified training data, therefore, is only collected where the initial classifier is not consistent. As found in other work [54], we also discovered that this approach achieves very high accuracy with relatively few manually classified pairs. The following subsection outlines the design and implementation details of the *active learner* used in ALFAA.

Active Learning Design Details

In order to seed the training set for the active learning procedure, we have to resolve the problem of how to select a set of initial pairs for manual labeling. If we randomly select authors, the fraction of matches obtained will be very low as described above. To increase the proportion of matches in this seed data, we identify non-homonym developer IDs where there were at least two distinct emails for the same name (first and last) or where there were at least two distinct names for the same email. We then sample 2,825 pairs from this set, so that for each pair either name or email is the same. These 2,825 pairs were manually labeled as a match (1) or a non-match (0). We found 2,016 matches and 809 non-matches in this set. We also calculated the string similarity and behavioural similarity for each pair in this set. We then randomly partitioned this data into ten parts and fit three classifiers on nine parts of the data (each of these 9 parts had overlapping and different observations). A typical *active learning* approach would then use these classifiers to predict matches on the data outside these pairs. However, since manual labels may be prone to errors, we add an additional step of using these three models to predict matches on the data that has already been manually labeled. Each classifier was used to predict outcomes for the all 2,825 pairs. As expected, the three classifiers trained on different training subsets yielded slightly different predictions. There were 2,345 pairs where all three learners did not agree (i.e at least one learner had a prediction different from the other two). This is the confusion region of the learner, as shown in Table 4.1. Table 4.2 shows an example of the correct predictions made by the learners where the manual labels

Table 4.1: Disagreement between learners – Instances that generate the confusion region in the preliminary classifier

Learner1	Learner2	Learner3
Link	Link	No-Link
Link	No-Link	Link
No-Link	Link	Link
No-Link	No-Link	Link
No-Link	Link	No-Link
Link	No-Link	No-Link

Table 4.2: Manual labeling errors in training data identified by the preliminary classifier in the active learning phase

Developer ID1	Developer ID2	is_match	P1
scottda <scott.dangelo@hpe.com>	scott-dangelo <scott.dangelo@hp.com>	0	L
scott-dangelo <scott.dangelo@hp.com>	scottda <scott.dangelo@hpe.com>	0	L

were incorrectly assigned. The column "is_match" shows the manual labels and 'P1' is the outcome of the classifier. We made appropriate correction in the manually classified data and the resulting set was used as the training data for the next iteration of the *active learner*.

We used all 16 attributes (name, email, first name, last name, user name, inverse first name, name frequency, email frequency, last name frequency, first name frequency, user name frequency, files modified, time-zone, and text similarity) in the initial `Random Forest` model. We obtained the importance of each predictor in this initial model and dropped the attributes with low importance from all subsequent models.

4.2.4 Phase 3: Classification

In the third phase of ALFAA which is the final classification phase, we train a classifier using the training data gathered from the *active learner* and do a 10-fold cross-validation. The developer IDs found in the 2,345 pairs that were manually classified in the `active learning` phase, were used to generate all ID pairs. This led to the creation of 5,499,025 pairs out of which 5,515 pairs were marked as matches (1) and the rest are non-matches

(0). Following existing record matching literature [], we build a `Random Forest` model using observations that meet the following criteria: string similarity of at least one component derived from the developer IDs is greater than 0.8, the file similarity between the pair is greater than 0, time-zone similarity between the pair is greater than 0.9, and text similarity between the pair is greater than 0. This simple heuristic was introduced to reduce computation time by dropping instances that are unlikely to contribute to learning. Using this model, we perform a 10-fold cross validation, the confusion matrix generated from this is shown in Table 4.3. It is evident that false positive and false negative errors generated by this model are extremely low.

The final predictor of identity matches involves a transitive closure on the pairwise links obtained from the classifier[6]. The result of the transitive closure is a set of connected components with each cluster representing a single author. Once the clusters are obtained, we consider all clusters containing 10 or more elements since a significant portion of such clusters had multiple developers grouped into a single component. The resulting 20 clusters - 44 elements in the largest and 10 elements in the smallest cluster among these, were then manually inspected and grouped. This manual effort included the assessment of name, user name and email similarity, projects they worked on, as well as looking up individual's profiles online where names/emails were not sufficient to assign them to a cluster with adequate confidence.

Table 4.3: Confusion matrix of 10-fold cross validation of the `Random Forest` Model to classify developer ID pairs as matches (1) or as non-matches (0)

	0	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
0	549,609	3	548,181	3	549,470	4	551,139	2	550,112	5
1	0	993	2	1,108	0	1,079	0	1,039	1	1,012
	0	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
0	549,211	1	549,404	1	547,961	3	548,730	2	549,569	2
1	3	1,067	0	1,032	0	1,019	0	1,086	0	1,010

[6]We found that more accurate predictors can be obtained by training the learner only on the matched pairs, since the transitive closure typically results in some pairs that are extremely dissimilar, leading the learner to learn from them and predict many more false positives

An example of cluster reassignment is given in Table 4.4 where we dis-aggregated a single large cluster of 11 IDs to 3 smaller clusters. The first column is the author ID, the second is the cluster number the ID was assigned to by the algorithm, the third column is the manually assigned cluster number after disaggregation. We noticed that, the largest cluster of size of 44 included all IDs that were associated with 'root' and therefore were not representative of any actual developer.

Therefore, we dis-aggregated the entire cluster to form 44 single element clusters. The output of this phase is a cleaned dataset in which we have corrected synonym errors via machine learning and fixed some of the homonym errors by inspecting the largest clusters. Since an experiment selecting a sample of pairs from this resulting set and validating them had very low level error rates, we use it as a reference or "golden" data set representing developer identities for the further analysis.

We use one of the models from the 10-fold cross validation to predict links or non-links for our entire data set of over 256M pairs of records from the 16,007 OpenStack developer IDs. The classifier found 31,044 links and we generated an additional 3,293 links through transitive closure. Therefore, we have 34,337 pairs linked after running the disambiguation algorithm. Using this, we constructed a network that had 10,835 clusters that were later manually inspected and disaggregated using the procedure described in subsection 4.2.4. Finally, we were left with 10,950 clusters, each representing an author, with 14 elements in the largest cluster, corresponding to the highest number of aliases by a single individual as shown in Table 4.5. The results extracted in each phase is illustrated in Figure 4.3.

The results and evaluation of this method applied on the 16,007 IDs in the OpenStack project is reported in Chapter 5, using standard measures of precision and recall as well as more specific measures - splitting and lumping as done in record matching literature. The next chapter also details an extensive evaluation to reflect on how this approach compares to other techniques that are existing in this domain.

Table 4.4: An example of cluster cleanup through manual disaggregation - Cluster number 22 that was identified after transitive closure and was split into 3 clusters after a manual check

Author Identity	**Cluster#**	**New Cluster#**
AD <adidenko@mirantis.com>	22	1
Aleksandr Didenko <adidenko@mirantis.com>	22	1
Alexander Didenko <adidenko@mirantis.com>	22	1
Sergey Vasilenko <stalker@makeworld.ru>	22	2
Sergey Vasilenko <sv854h@att.com>	22	2
Sergey Vasilenko <sv@makeworld.ru>	22	2
Sergey Vasilenko <svasilenko@mirantis.com>	22	2
Sergey Vasilenko <xenolog@users.noreply.github.com>	22	2
Vasyl Saienko <vsaienko@mirantis.com>	22	3
vsaienko <vsaienko@cz5578.bud.mirantis.net>	22	3
vsaienko <vsaienko@mirantis.com>	22	3

Table 4.5: 14 Developer IDs in the largest cluster corresponding to an individual with highest aliases after manual disaggregation

AuthorID
Greg Holt <gholt@rackspace.com>
tlohg <z-github@brim.net>
Greg Holt <greg@brim.net>
tlohg <gholt@rackspace.com>
Greg Holt <gregory.holt@gmail.com>
gholt <z-launchpad@brim.net>
Greg Holt <gregory_holt@icloud.com>
gholt <z-github@brim.net>
Greg Holt <z-github@brim.net>
gholt <gregory.holt+launchpad.net@gmail.com>
Gregory Holt <gholt@racklabs.com>
gholt <gholt@rackspace.com>
gholt <devnull@brim.net>
gholt <gholt@brim.net>

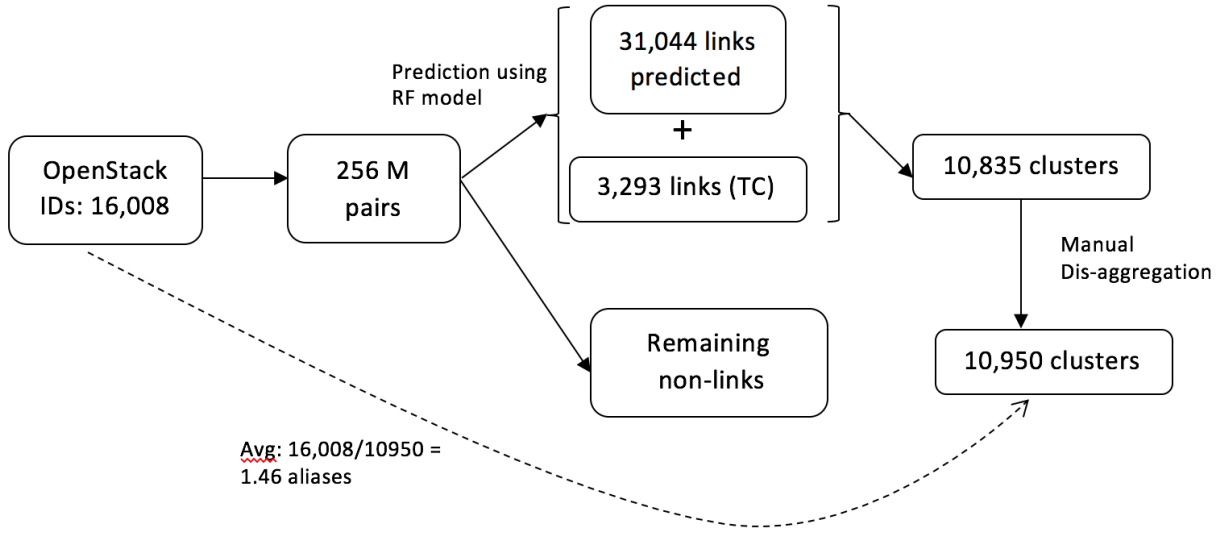

Figure 4.3: Flowchart showing the results from each phase of the disambiguation process from 16,007 OpenStack developers

4.3 Deanonymization - Correcting Homonyms

Homonym errors leave no information in a developer ID that can be used in the process of de-anonymization. For example, an ID as "A Student <secretmessage@protonmail.com>" divulges no information about the identity of the individual associated with it. Therefore, the process of resolving `Homonyms` differ from that of resolving `Synonyms`. One of the larger challenges in the `Homonym` correction process is its identification. As the data sets we are dealing with are enormous, manual identification is not possible. Therefore, we need a codifiable method that enables the identification and extraction of `Homonyms`. Figure 4.4 outlines the workflow of the `Homonym` correction process.

4.3.1 Prototype Design

To demonstrate the feasibility of our proposed method we design an experiment using a small sample of developer IDs. As it is hard to obtain ground truth for our data (it is dependent on developer response to surveys) we design this experiment by simulating `Homonyms`. We follow the 4 steps described in the `Homonym` correction process and report our findings through the experimentation process.

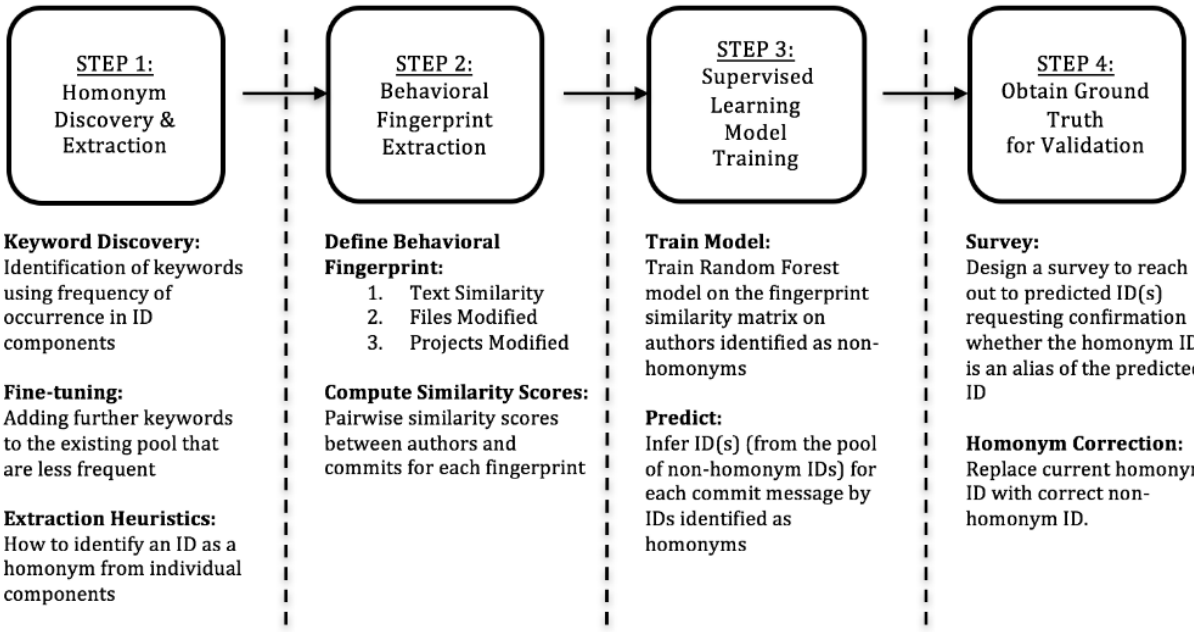

Figure 4.4: Concept of the `Homonym` Correction Process

4.3.2 Phase 1: Discovery and Extraction

The identification of `Homonyms` is a non-trivial task because it is hard to define and codify characteristics of developer IDs that make it a `Homonym`. Deep learning methods such as Named Entity Recognition (NER) models are trained to identify entities from contextual data which is absent in our data set. NLTK packages also assume proper name formats that are required to be present in order to be able to identify proper nouns. As our data is not normalized [5], (many IDs belonging to a single developer contain a single word without spaces and/or without capitalization) the IDs representing individuals are not identifiable through simple heuristics such as checking for spaces, camel-case, hyphens etc. Therefore, it is also not simple to isolate `Homonyms` using any identifiable unique characteristics from IDs that may belong to an actual developer. One way to discover/identify `Homonyms` would be to use the brute force method of examining all developer IDs that have more than a single commit. Needless to say, that would be an impractical approach given the size of our data set.

In this work, we devise an approach to discover `Homonyms` from a large collection of developer IDs and commits. Before we discuss the discovery process, it is important

to understand the structure of a developer ID and its components. A developer ID appears as a combination of a name and email address e.g. "A Student <secretmessage@protonmail.com>". Each developer ID is composed of the following components:

1. Name: String up to the space before the first "<"
2. Email: String within the "< >" brackets
3. First name: String up to the first space encountered in the name field
4. Last name: String after the last space encountered in the name field
5. User name: String up to the "@" character in the email field

The most frequently occurring components of IDs are likely to contain keywords that are common in `Homonyms` i.e. root, unknown etc. However, this might not be true for all the cases as names such as "Alex", "Lee", "Smith" etc. are likely to be commonly used names of people as shown in Table 3.1. Therefore, we start the discovery process through an amalgamation of automated and manual efforts in our data set of 16 million developer IDs. We begin by looking at the 50 most frequent first names, last names, user names to create a list of words, the presence of which would characterize a developer ID as a `Homonym`. We refer to this set of words as `Homonym` keywords. To add to this list, we subset IDs where a single component i.e. first name or last name contains any of these keywords and look for any additional keywords to append to the existing pool. To ensure we also include the less frequent `Homonym` keywords we exclude the IDs that have the first name component match to a keyword and look at the components of the remaining IDs. Any additional `Homonym` keywords that was discovered in this process i.e. ("apple", "jenkins") was added to the existing pool. However, we were able to compose the bulk of the `Homonym` keyword list by looking at the most frequent components of developer IDs. The latter process helped us refine that list further by adding the less frequent `Homonym` keyword. We also added corresponding keywords with proper capitalization when absent and vice-versa i.e. "Admin", "admin".

Table 4.6 shows the count of the 15 most frequently appearing `Homonym` keywords found in each of the components i.e. name, first name, last name etc. We discovered 94

Table 4.6: The 15 most frequent `Homonym` keywords discovered from over 16 Million Developer IDs

name	count	first_name	count	last_name	count	user_name	count
a	390466	a	390686	a	390608	root	111646
unknown	182559	unknown	182580	unknown	182565	nobody	48046
root	107360	Unknown	145966	User	147095	ubuntu	31717
nobody	47521	root	107493	root	107498	info	28187
Ubuntu	32894	nobody	47524	nobody	47526	github	27656
Unknown	15708	Ubuntu	33442	Ubuntu	33100	me	21289
Administrator	9277	Administrator	9305	Administrator	16725	admin	20534
nodemcu-custom-build	6073	(no	7212	Unknown	15744	mail	16712
=	5430	Your	6506	user	7574	Administrator	15566
user	4722	nodemcu-custom-build	6073	author)	7200	git	13118
admin	4466	=	5480	Name	6507	contact	12491
User	2590	user	4825	nodemcu-custom-build	6073	none	12110
Admin	2407	admin	4527	=	5448	student	11327
git	2272	User	3522	admin	4680	devnull	9835
student	2267	Admin	2664	Admin	4056	dev	8666

unique `Homonym` keywords which are categorized in Table 4.7. Table 4.8 shows the count of IDs where each component of the corresponding ID matched a `Homonym` keyword in our data set. We found 213,372 Ids where all components belonged to the `Homonym` keyword list, which is about 1.3% of the total number of IDs.

NOTE: We also found names such as "John Doe" which are typically used to preserve anonymity. The individual components of such names are intentionally left out of the `Homonym` keyword list as only the occurrence of both components together makes an ID a `Homonym`. We found 881 occurrences of names with last name "Doe", majority of which had first names that are not "John" indicating that these are non-`Homonym` IDs belonging to real developers.

As we do not have ground truth for actual `Homonyms` found in our data set, we simulate `Homonym` IDs by sampling a number of developer IDs and anonymizing a part of the commit messages associated with those IDs (we set aside the actual developer IDs associated with these commits to measure accuracy of our proposed method). This effectively simulates `Homonym` errors in our data as now we have no information related to the identity of the developer making the commit. We sample 24 (an arbitrarily chosen number) developers, from the set of 14,877,729 IDs where none of the components are `Homonym` keywords, and ensure that among these 24 developer IDs there are groups of IDs that belong to the same project. We extract the commits (identified by the commit

Table 4.7: Full list of keywords compiled and used for the homonym discovery process

Organization	Projects	Roles	Anonymous	Misc.
Mozilla	Openstack	Administrator	Anonymous	Bot
mozilla	openstack	administrator	anonymous	bot
Google	Jenkins	Admin	You	Daemon
google	jenkins	admin	you	daemon
Linux	Ubuntu	Support	Your	Server
linux	ubuntu	support	Name	server
Cisco	Mac	none	Me	Auto
cisco	mac	None	me	auto
Walmart	GitHub	Dev	God	BuildTools
walmart	Github	dev	god	mail
Apple	github	Student	hello	www-data
apple	Git	student	None	nodemcu-custom-build
cmart	git	Guest	none	devnull
		guest	(no	noreply
		Root	author)	build
		root	Unknown	info
		iguest	unknown	ad
		User	Nobody	contact
		user	nobody	example
		Default	Public	=
		default	public	a
		Author	Test	EC2 Default User
		author	test	

Table 4.8: Developer IDs statistics for components found as `Homonym` Keywords

Component Matched	Count
Name	844,306
First Name	992,063
Last Name	1,028,777
User Name	451,256
All Comp.	213,372
None Comp.	14,877,729

hash) that belong to each of these IDs and remove all duplicate commit messages (in terms of text, each commit message has a unique hash) from the commit message pool of each developer. We also filter out any commit message shorter than five words and longer than 50 words. We split each of the 24 processed commit message sets into ten equal parts and create ten training and ten testing sets for the ten-fold cross validation. Each training set has nine parts of commit messages with known developer IDs and each test set has the remaining one part of commit messages, where associated developer IDs have been removed, of each of the 24 developers. In Table 4.9, we report the number of commits we extracted from these 24 developers, the number of unique commit messages, the average length (in terms of word count) of each commit message derived from the unique commit messages, the total number of words in the unique commit messages and the number of unique vocabulary used by each developer.

4.3.3 Phase 2: Behavioral Fingerprint Predictors

In this step, we extract the three behavioral fingerprints - text similarity, files touched similarity and project similarity.

1. **Text Similarity:** We determine text similarity between non-`Homonym` IDs and commit messages by `Homonym` IDs by using Gensim's implementation of Doc2Vec. We build a Doc2vec model (using dm-mean and a window size of 2) to extract 200 dimensional numeric vectors representing each non-`Homonym` developer ID. To train this model, we use the training data set containing nine parts of commits made by each developer and we build ten such models for cross validation. Since we want to compute similarity between a non-`Homonym` developer ID and a commit message we need to extract the numeric vector representation of each commit. To do this, we use model.infer_vector() for each commit message made by a Homonym ID in the test set. To evaluate the effectiveness of using this embedding technique, we generate the 3 most similar, 2 most similar and the most similar non- `Homonym` IDs to a given commit and obtain the number of times the actual developer ID associated with the commit was correctly identified by the Doc2Vec model. Table 4.9

Table 4.9: Sampled developer commits and text similarity results

ID	Total Commits	Unique_Commits	Avg_length	Total_words	Unique_words	top1	top2	top3
Matthew Martin	51,064	25,138	9.34	234,768	1,716	96.42	98.25	98.56
Chao Yu	33,085	1,024	32.2	32,972	2,085	85.11	90.83	92.77
Andreas Jaeger	6,974	3,724	24.7	91,944	13,513	77.09	88.82	92.52
Fabien Potencier	69,001	12,362	18.4	226,855	33,458	65.64	76.98	83.04
Eudaldo Alonso	44,270	5,035	9.45	47,558	5,187	65.59	76.08	81.31
Emilien Macchi	10,535	4,848	22.8	110,734	15,490	65.24	74.89	80.73
Dan Carpenter	43,259	6,123	28.3	173,013	18,986	59.42	69.57	75.02
Rene Rivera	29,514	5,779	13.7	79,159	11,988	58.96	72.84	80.10
Daniel James	55,371	11,710	13.1	153,083	21,236	55.72	73.93	81.88
Chris Wilson	46,265	12,130	23.6	286,417	36,774	54.51	65.26	71.67
Laurent Montel	68,589	26,691	10.8	289,358	36,198	53.87	67.57	75.16
Allen Winter	28,069	8,231	17.8	146,617	27,223	49.78	64.74	73.91
Eric Dumazet	53,070	1,972	24.5	48,221	6,960	49.27	56.51	60.49
Glenn Morris	78,997	11,968	14.6	174,196	28,758	47.04	60.41	69.02
Monty Taylor	21,673	7,353	20.0	146,998	23,678	44.67	61.32	71.36
Stephen Kelly	22,136	9,595	13.9	133,628	22,263	40.76	56.46	66.59
Kent Fredric	46,046	25,425	9.92	252,113	33,111	37.85	49.01	56.91
Michael Bestas	41,752	8,109	12.6	102,038	17,771	31.47	40.50	46.64
David Cramer	24,260	10,050	9.72	97,715	16,519	30.64	42.55	50.58
Greg Kroah-Hartman	57,521	4,456	21.3	94,759	11,477	29.27	40.56	47.23
Christian Faulhammer	42,053	7,518	9.02	67,836	10,841	29.26	47.84	57.25
Junyuan Xue	1,672	527	8.84	4,657	1,412	18.67	30.19	36.48
Julio Camarero	29,091	5,471	10.6	57,759	10,314	15.61	34.18	42.50
Simon Legg	1,342	478	9.38	4,482	1,411	3.47	5.56	8.10

shows the average results from the ten fold cross validation. The last 3 columns in Table 4.9 shows the average percentage of correctly identified (and whether the correct developer was in the two/three most similar) developer ID corresponding to each commit. The results show that the models were able to predict the correct developer ID significantly greater than random guess (1/24), we conclude that the Doc2Vec models are effective in capturing the characteristic of commits made by developers. We therefore generate a list of similarity scores between each of the 24 developer IDs and each commit message (by `Homonym` IDs) to be used as a predictor for our supervised learning method discussed later in this section.

2. **Files touched Similarity** We extract the files that have been modified by a developer and the files that were modified by each commit. As discussed earlier, we weight each file with the inverse number of authors who modified that file. For each author-commit pair we extract the common files modified by both author and the commit and generate the similarity score for this pair by summing over the weights of these files.

3. **Project Similarity** We extract the projects that have been modified by a developer and the projects that were modified by each commit. As discussed in section 4.2.2, we weight each project with the inverse of number of authors who modified that project. For each author-commit pair we extract the common projects modified by both author and the commit and generate the similarity score for this pair by summing over the weights of these projects.

4.3.4 Phase 3: Supervised Classification

We use each behavioral fingerprint as predictors to build a `Random Forest` model. We create training and testing set for the model and perform a ten-fold cross validation. We have about 4.6 Million rows in our training data and about 500,000 rows in our testing data set. Roughly 5% of the observations in our training data set is a match. As the ratio between match and non-matched observation is too low, we drop all observations where text similarity score is below 0.25 and file similarity score is 0 *or* where text similarity score is below 0.25 and project similarity score is 0. This conditioning creates a better balance in the number of positive and negative outcomes in the training set as well as helps us to train a `Random Forest` model faster by requiring less memory.

We test these models by generating probabilistic outcomes for the binary classes and report the results in Table 4.10. We assign a match between a developer and commit if the probability for a certain outcome (0 or 1) is greater than 0.6. Table 4.10 reports the number of observations for each outcome in the training and testing data set and the corresponding precision and recall for each fold. We obtained the highest precision to be 99.7% and the highest recall to be 99.8% and the lowest precision and recall to be 80.39% and 75.16% respectively.

Table 4.10: Results from 10-Fold cross validation of `Random Forest` using the three fingerprints as predictors

	Fold 1		**Fold 2**		**Fold 3**		**Fold 4**		**Fold 5**	
Test	0 58,754	1 20,015	0 54,740	1 18,377	0 47,054	1 16,217	0 34,028	1 12,757	0 13,343	1 6,957
Train	0 130,833	1 75,733	0 192,138	1 78,440	0 206,484	1 80,788	0 219,066	1 84,252	0 242,096	1 90,070
Precision	80.39		86.34		90.53		89.8		93.69	
Recall	75.16		65.36		73.34		67.25		81.97	
	Fold 6		**Fold 7**		**Fold 8**		**Fold 9**		**Fold 10**	
Test	0 6,833	1 6,068	0 4,748	1 4,594	0 4,748	1 4,594	0 3,047	1 3,560	0 2,315	1 2,362
Train	0 249,296	1 90,968	0 249,526	1 92,150	0 250,651	1 92,422	0 252,080	1 93,454	0 250,885	1 94,657
Precision	95.96		95.66		95.7		98.98		99.7	
Recall	90.98		86.79		89.31		96.15		99.8	

Chapter 5

Results and Evaluation

5.1 Overview

In this chapter, we report how ALFAA performs through standard measures - precision and recall, using the prototype designed described in Chapter 4. We also report splitting and lumping errors which is used in record matching literature. To evaluate our approach we compare ALFAA to existing state-of-the-art methods that are used commercially and in research. We also report the impact of incorporating behavioral fingerprints by comparing the measures of two models - a model built with behavioral fingerprint and a model that is built without. Such erroneous data has an impact on developer networks. We therefore, conduct an experiment to assess the impact of the split and lump errors on networks by comparing the erroneous network to its corrected version.

5.2 Results

The standard measure of accuracy in binary classification is precision and recall. Precision is the number of true positives (the number of IDs that are correctly classified as a match) divided by the total number of elements that are identified as belonging to the positive class by the classifier (true positives and false positives which is the number of IDs that are incorrectly identified as a match). A perfect precision of 1.0 indicates that a classifier never incorrectly classifies a pair as a match. Recall is defined as the number of true

positives divided by the total number of elements that actually belong to the positive class (true positives and false negatives which is the number of items that are incorrectly classified as a non-match). A perfect recall score of 1.0 indicates that a classifier never incorrectly classifies a non-match. Ideally, we would like our classifier to minimize both false positive and false negative errors.

$$Precision = \frac{tp}{tp + fp}$$

$$Recall = \frac{tp}{tp + fn}$$

The random forest classifier produced an average precision of 99.9% and an average recall of 99.7%. This was calculated by averaging the precision and recall values produced by each model in the ten-fold cross validation shown in Table 4.3.

Ten-fold Cross Validation

Average Precision: 99.9%

Average Recall: 99.7%

Since record matching is a slightly different problem from traditional classification, the literature introduces two additional error metrics: splitting and lumping [58]. Lumping occurs when multiple author IDs are identified to belong to a single developer. The number of lumped records is defined as the number of records that the disambiguation algorithm incorrectly mapped to the largest pool of IDs belonging to a given author. Splitting occurs when an ID belonging to a single developer is incorrectly split into IDs representing several physical entities. The number of split records is defined as the number of author IDs that the disambiguation algorithm fails to map to the largest pool of IDs belonging to a given author.

Since, these two metrics only focus on the largest pool of IDs belonging to a single developer and ignores the other clusters of IDs corresponding to the same unique developer, the work in [68] modifies these measures to evaluate all pairwise comparison of author records made by the disambiguation algorithm. In this work, two measures

are discussed – splitting and lumping errors. Splitting errors occur when a single unique developer is mapped to multiple identity signatures. Lumping errors occur when multiple IDs are incorrectly mapped to a developer's largest cluster of IDs by the disambiguation algorithm. According to the latter approach, we create a confusion matrix of the pairwise links from the golden data set and the links created by the classifier and calculate splitting and lumping in the following manner:

$$Splitting = \frac{fn}{tp + fn}$$

$$Lumping = \frac{fp}{tp + fn}$$

5.3 Evaluation

In this section, we address the questions related to the accuracy of the manually labeled training data and to compare the proposed approach to work from the commercial domain as well as recent research. It is important to note that we are evaluating our algorithm trained on a small amount of training data and, as with other machine learning techniques, we expect the accuracy of the models to increase with more training data added in the future.

5.3.1 Accuracy of the Training Data

It is extremely hard to gather absolute ground truth for the data set that is considered in this study. In the case of `Synonyms`, it would require reaching out to the contact address found in each developer ID to confirm or reject if a certain commit was made by that developer or whether a list of other IDs belong to the developer being reached out to. In either case, it is hard to get accurate response from users as it is hard for them to identify their commits and all variants of their credentials especially those that were unintentional spelling mistakes. The other reason that makes attaining ground truth hard is the fact that response rates from surveys are extremely low. Regardless, we made efforts to reach out to developers and report the findings below. To compensate for the the absence of ground

truth, investigations are made to ensure the accuracy of the training data. To do this, we measure the inter-rater reliability between two raters who manually labeled the training dataset to see how much homogeneity or consensus there is between the raters. A high agreement between raters indicate that the training data can be deemed reliable.

To measure inter rater reliability, two independent human raters were assigned who are both raters are PhD students in Computer Science with a good understanding of the problem. They were presented with a spreadsheet containing 1,060 pairs of OpenStack developer IDs and were asked to mark it using the following protocol. Each rater was instructed to inspect each pair of developer IDs (full name and email) listed in the spreadsheet and supplemented by developer's affiliations (see Section 5.3.2, the dates of their first and last commits in the OpenStack projects, and their behavioral similarity scores. Based on these factors (listed next to the pair in the spreadsheet), each rater was instructed to make a decision and

1. Mark a developer ID pair as a match (1) if the two IDs are almost certainly from the same person.

2. Mark a developer ID pair as a non-match (0) if the two IDs are almost certainly not from the same person.

3. Mark a developer ID pair a number between 0 and 1 reflecting the raters subjective probability that they are representing the same person.

The raters were further encouraged to search for developers on GitHub or Google if they did not feel confident about their decision. For cases where both raters marked a developer ID pair as a match (1) or a non-match (0), we found agreement between the two raters in 1,011 cases and disagreement in 17 cases. By thresholding the 32 cases that had probability value greater than zero and less than one to the nearest whole, we obtained 1,042 instances of agreement (98.3%) and 18 cases of disagreement (1.69%).

Inter Rater Reliability

1,042 instances of agreement and 18 cases of disagreement out of 1,060 cases.

98.3% agreement and 1.69% disagreement.

We also compare the ten-fold cross-validation predictions described above with the second rater (whose input was not used for training). The numbers of disagreements between the second rater and the predictions over ten folds ranged from 11 (1.03%) to 18 (1.69%) (a mean of 15.18). The second rater had, therefore, similar or better agreement with the prediction than with the first rater.

We, thus, have established the degree to which the two raters agree on the decision, but not necessarily that either of the raters was correct. To validate the rater's opinions we, therefore, administered a survey to a randomly selected set of developers. The survey provided respondents with a set of commits with distinct developer strings. All commits, however, were predicted to have been done by the respondent and each respondent was asked to indicate which of the commits were the ones made by them. From a randomly selected 400 developers 69 emails bounced due to the delivery problems. After 20 days we obtained 45 valid responses, resulting in a response rate of 13%. No respondents indicated that commits predicted to be theirs were not submitted by them, for an error rate of 0 out of 45.

Survey response

0 errors reported out of 45 responses corresponding to 13% response rate.

This allows us to obtain the bound on the magnitude of error. For example, if the algorithm has the error rate of 5%, then we would have less than one in ten chances to observe 0 out of 45 observation to have errors[1]. After establishing high accuracy of the training data we proceed to compare ALFAA to an approach that was implemented by professional commercial effort.

[1] Assuming independence of observations and using binomial distribution.

5.3.2 Comparison with a Commercial Effort

OpenStack is developed by a group of companies, resulting in an individual and collective interest in auditing the development contribution of each firm working on OpenStack. This task was outsourced to Bitergia[2], which is a company dedicated to performing software analytics. The disambiguation data on OpenStack developers produced by Bitergia was collected for this study. This data was in a form of a relational database (mysql) that had a tuple with each commit SHA1 and developer ID and another table that mapped developer ID (internal to that database) to developer name (as found in a commit). The Bitergia data had only 10,344 unique developer IDs that were mapped to 8,840 developers (internal database IDs). We first restricted the set of commits in our data set to the set of commits that were in the Bitergia database and selected the relevant subset of developers (10,344 unique developer IDs) from our data for comparison to ensure that we are doing the comparison on exactly the same set of developers. Bitergia algorithm missed 17,587 matches predicted by our algorithm and introduced six matches that ALFAA does not predict. In fact, it only detected 1,504 matches of over 22K matches (under 7%) predicted by ALFAA. Bitergia's matching predicted 8,840 distinct developers, or 41% more than ALFAA which estimated 6,271 distinct developers from the 10,344 distinct developer IDs.

Bitergia VS ALFAA

10,344 distinct developer IDs mapped to 8,840 developers by Bitergia.

The same developer IDs mapped to 6,271 developers by ALFAA.

As shown in Table 5.1, it has almost 50 times higher splitting error than manual classification, though it almost never lumps two distinct authors. We, therefore, conclude that the prediction done by the commercial effort has substantially lower accuracy.

[2]https://bitergia.com/

5.3.3 Comparison with a Research Study

We also compare our method to a recent research method[3] that was applied on data from 23,493 projects [66] from GHTorrent to study social diversity in software teams. From this point forward, we refer to that method as "`Recent`". Method `Recent` starts by creating a record containing elements of the name and email address as shown on the left of Figure 5.1. It then forms candidate pools of identities linked by matching name parts, then uses a heuristic to accept or reject each pool based on counts of different similarity "clues". The authors then iteratively adapted this automatic identity matching by manually examining the pools of matched emails and adjusting the heuristic. The authors of `Recent` limit the false positives by being as conservative as possible when merging/linking two IDs. For example, if multiple aliases share the same well-formed and non-fictitious email address, the aliases are merged based on the assumption that email addresses are individual. To ensure that the heuristics in `Recent` were applied in a way consistent with their prior use, the first author of the original paper [66] was asked to run it on the data set used in this study and adjust it analogously to how he had adjusted for his own studies[4].

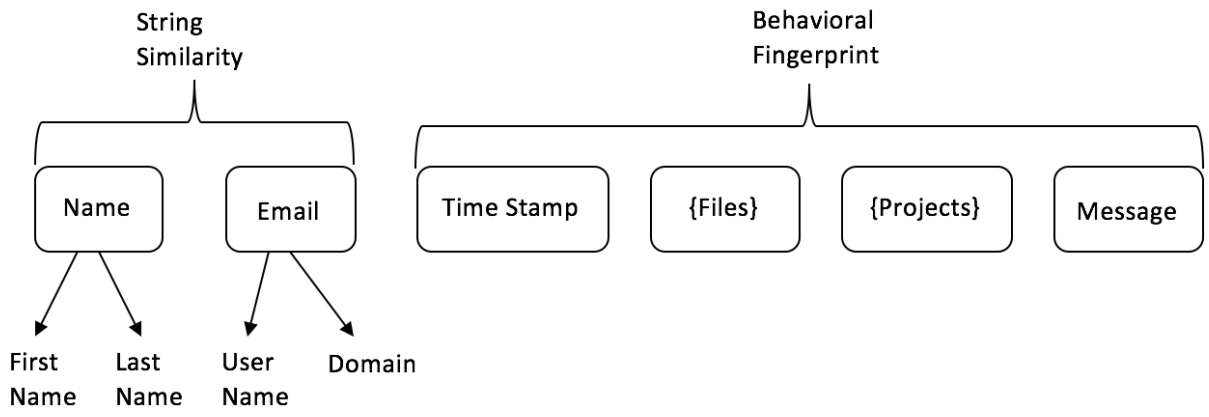

Figure 5.1: Extracted components of commits that are used for string similarity and behavioral fingerprint

[3] https://github.com/bvasiles/ght_unmasking_aliases

[4] The author got much better results than we could obtain using their published code without modifications.

We first compare the results of `Recent` to the results generated by ALFAA on the entire set of over 16,000 OpenStack developers and then on a larger data set described in the next subsection. We design an experiment described next, to compare the accuracy of `Recent` and ALFAA.

Manual validation of results from `Recent` and ALFAA

ALFAA produced 1,876,595 matches that were not identified by `Recent`. `Recent` produced 363,818 matches that were not matched by ALFAA. In order to understand and validate the differences, an experiment was designed to compare the results of `Recent` to ALFAA generated from a set of 16,007 OpenStack developers. As the size of the full data set is too large for manual validation (over 256M pairs), a small section of the data for manual inspection was sampled. The protocol for sampling is defined as follows:

- Remove all self-matched observations from consideration i.e. where developer ID1 and developer ID2 are identical
- Remove all observations that have positive outcomes from both approaches i.e. the intersection of matches, for both ALFAA and `Recent`
- Create Set1: Randomly sample 500 observations where ALFAA yields a positive outcome and `Recent` does not
- Create Set2: Randomly sample 500 observations where `Recent` yields a positive outcome and ALFAA does not
- Combine Set1 and Set2 and randomly extract 500 observations from the combined set without replacement. Let's call this Sample1. The remaining observations will be in Sample2. This step ensures that no bias is introduced when using human raters while generating ground truth

To sample, random samples of 1000 pairs of IDs were generated until it had more than 50% cases where the email of ID1 was not identical to the email of ID2. This was done because `Recent` automatically matched IDs that had the same email, and therefore, we

wanted to keep a substantial number of observations in our experiment where the email addresses are different. We sampled 500 observations from each of these two sets and found that 418 observations out of 500 in Sample1 and 312 out of 500 in Sample2 did not have identical emails. We used two raters (both PhD students in Computer Science,) to label each observation as a match (1), a non_match(0), and uncertain (0.5) keeping the following criteria in mind besides the individual rater's judgment.

- Label as a match (1) if first names and last names match and neither are Homonyms
- Label as a match (1) if names are somewhat similar (case difference, partial match, abbreviated etc.) and emails indicate personal and work (deduced from email domains)
  Example: jonathanramirez <jonathanramirezmeza@gmail.com>, Jonathan Ramirez <jonathan@Jonathan-BMP.iptv.microsoft.com>
- Label as 1 if name in ID1 matches username in ID2
  Example: Elliot Fehr <elliot@weebly.com>, Elliot <elliotfehr@gmail.com>
- Label as unsure (0.5) If a name has high empirical frequency (appears frequently in the data set)
- Label as unsure (0.5) if a name has high cultural frequency (known to be common in a nationality or culture)
- If one or both of the ID(s) is/are Homonym(s) label as 0.5
  Example: coffeemug <coffeemug@dell-desktop.example.com>, Slava Akhmechet <coffeemug @Elyas-MacBook-Pro.local>
- If there are multiple names present in the IDs, then label 0.5
  Example: Jonathan Berkhahn and Raina Masand <rmasand@pivotal.io>, Ruth Byers and Raina Masand <rmasand@pivotal.io>

We selected all instances that were labeled as a match (1) or a non-match (0) by both raters. In other words, we dropped all instances that were labeled a 0.5 by any one rater. We extracted 534 such observations out of the 1,000 labeled instances. There were 47

cases of disagreement in the 534 cases (8.8% disagreement and 91.2% agreement) between the two raters. We also calculate Cohen's Kappa (κ) which is a robust measure for for inter-rater agreement for categorical items and get $\kappa = 0.722$. We conclude that this is satisfactory as $0.6 \leq \kappa \leq 0.8$ is considered to be substantial agreement according to [25]. We found 184 pairs from results produced by ALFAA (34.5%) and 350 pairs from results produced by `Recent` (65.5%) in this set.

Rater 1 validation results for ALFAA: Out of the 178 pairs that were matched by ALFAA, Rater 1 found 7 instances where ALFAA wrongly matched a pair of IDs and 177 instances where ALFAA correctly matched a pair of IDs. This means that ALFAA achieved 3.8% False Positive rate and 96.2% True Positive rate from Rater 1.

Rater 1 validation results for `Recent`: Out of the 350 pairs that were matched by `Recent`, Rater 1 found 80 instances where `Recent` wrongly matched a pair of IDs and 270 instances where `Recent` correctly matched a pair of IDs. This means that `Recent` achieved 22.8% False Positive rate and 77.14% True Positive rate from Rater 1.

Rater 1 validation:
ALFAA: FP = 3.8%, TP = 96.2%
Recent: FP = 22.8%, TP = 77.14%

Rater 2 validation results for ALFAA: Out of the 178 pairs that were matched by ALFAA, Rater 2 found 28 instances where ALFAA wrongly matched a pair of IDs and 156 instances where ALFAA correctly matched a pair of IDs. This means that ALFAA achieved 15.2% False Positive rate and 84.8% True Positive rate from Rater 2.

Rater 2 validation results for `Recent`: Out of the 350 pairs that were matched by `Recent`, rater 2 found 94 instances where `Recent` wrongly matched a pair of IDs and 256 instances where `Recent` correctly matched a pair of IDs. This means that `Recent` achieved 26.8% False Positive rate and 73.14% True Positive rate from Rater 2.

Rater 2 validation:
ALFAA: FP = 15.2%, TP = 84.8%

Recent: FP = 26.8%, TP = 73.14%

In conclusion, ALFAA and `Recent` achieved an average false positive rate of 9.5% and 24.85% respectively. This shows that `Recent` is 2.61 times more prone to lumping error than ALFAA. To be able to report the splitting error, we would need to conduct a similar experiment by sampling from the set containing pairs that were labeled as a non-match (0) by ALFAA and the set containing pairs that were labeled as a non-match (0) by `Recent`, which is both time and labor intensive, and therefore, we keep it beyond the scope of this work.

Summary:
ALFAA: Average FP = 9.5%
Recent: Average FP = 24.85%
`Recent` is **2.61** times more prone to lumping error than ALFAA

Table 5.1 provides summary of the comparisons where the labels generated using method in the left column to be the ground truth. In particular, there is a good agreement between the raters: splitting error of 0.0139 and lumping error of 0.0139. The average ten-fold cross-validation error of ALFAA on the Rater 1 labeled data is 4.63 (0.0139/0.003) times and 13.9 (0.0139/0.001) times lower, though. The agreement between ALFAA and Rater 2 is similar to that between Rater 1 and Rater 2, suggesting that ALFAA has captured the heuristics implicit in the training set. Commercial effort has a much higher split error but it almost never lumps distinct individuals together. The last comparison of ALFAA vs `Recent` shows that ALFAA is 7.97 times more accurate than `Recent` with respect to splitting (0.1109/0.0139), and 3.5 times with respect to lumping (0.0487/0.0139).

5.3.4 Evaluation on a Large Set of Identities

To evaluate the feasibility of ALFAA on large scale we created a list of 1.8 million identities from commits to repositories in Github, Gitlab and Bioconductor for packages in 18

Table 5.1: Comparison of ALFAA against commercial and research methods

Set	Assumed Ground Truth	Comparison	Precision	Recall	Split	Lump
Training	R1	R2	0.9861	0.9861	0.0139	0.0139
Set	ALFAA	ALFAA	0.9990	0.9970	0.0030	0.001
	ALFAA	R2	0.9936	0.9823	0.0177	0.0063
Full	ALFAA	Bitergia	0.9991	0.4688	0.5312	0.0004
OpenStack	ALFAA	Recent	0.9480	0.8891	0.1109	0.0487

software ecosystems. The repositories were obtained from libraries.io data [47] for the Atom, Cargo, CocoaPods, CPAN, CRAN, Go, Hackage, Hex, Maven, NPM, NuGet, Packagist, Pypi, and Rubygems ecosystems; extracted from repository websites for Bioconductor[5], LuaRocks[6] and Stackage[7], and from Github searches for Eclipse plugins.

The application of the `Recent` algorithm mapped the 1,809,495 developer IDs to 1,411,531 entities as the algorithm was originally configured (1.28 aliases per entry), or 1,052,183 distinct entities after the heuristic was adjusted by its author, identifying an average of 1.72 aliases per entry. Upon applying ALFAA to this data set, we mapped the set to 988,905 – identifying an average of 1.83 aliases per entity. It is important to note that we did not incorporate any additional training beyond the original set of manually marked pairs and we expect the accuracy to increase further with an expanded training data set.

Recent: 1,809,495 developer IDs mapped to 1,052,183 developers

ALFAA: 1,809,495 developer IDs mapped to 988,905 developers

Notably, to apply ALFAA for 1.8M IDs, we need 3.2×10^{12} string comparisons for each field (first name, last name, etc) and the same number of comparisons for each behavioral fingerprint. To compare strings we used an allocation of 1.5 million core hours for the Titan supercomputer at Oak Ridge Leadership Computing Facility (OLCF) [8]. The entire calculation was done in just over two hours after optimizing the

[5]https://www.bioconductor.org
[6]https://luarocks.org
[7]https://www.stackage.org/lts-10.5
[8]https://www.olcf.ornl.gov/

Table 5.2: Precision, Recall, Splitting and Lumping derived from models with and without behavioral fingerprints

Metric	Without Behavioural Fingerprint (%)	With Behavioural Fingerprint (%)
Precision	99.3	99.9
Recall	99.5	99.7
Splitting	0.4	0.3
Lumping	0.6	0.1

implementation in pbdR [49] on 4096 16-core nodes. The approach can, therefore, scale to the entire set of over 23M author IDs in over 1B public commits. To compare behavioral fingerprints we exploited network properties (developers modify only a small number of all files) to reduce the number of comparisons by several orders of magnitude. Finally, it took us approximately two weeks to train a Doc2Vec model (for text embedding) on approximately 9M developer identities and 0.5B commits using a Dell server with 800G RAM and 32 cores.

5.3.5 Impact of Using Behavioral Fingerprints

To understand whether behavioral fingerprints have an impact on the accuracy of the algorithm we tried building a basic model without incorporating fingerprints. The results in Table 5.2 report the prediction performance where the prediction model was fit with and without the behavioural metrics. The table allows us to answer whether behavioural metrics increase accuracy. We obtained seven times higher precision error that increases from 0.1% to 0.7% and recall error increases 0.7 times from 0.3% to 0.5% when behavioural metrics are dropped from the models.

5.4 Impact on Developer Collaboration Network

In this section, we discuss the impact of identity errors in a real world scenario of constructing a developer collaboration network. More specifically, we measure the impact of disaggregation (or split) errors by comparing the raw network to its corrected version. To create the collaboration network (a common network used in software

engineering collaboration tools [9]), we start from a bipartite network of OpenStack with two types of nodes: nodes representing each developer ID and nodes representing each file, we refer to as G. The edges connecting a developer node and a file node represent the files modified by the author. This bipartite network is then collapsed to a regular author collaboration network by creating links between authors that modified at least one file in common. We then replace multiple links between the authors with a single link and remove authors' self links as well. The new network, which has 16,007 author nodes, depicts developer collaboration, we refer to as G′. We apply our disambiguation algorithm on G′ and aggregate author nodes that belong to the same developer and produce a corrected network which we refer to as G″. The network and its transformations are illustrated in Figure 5.2.

To evaluate the impact of correction from G′ to G″, we follow prior work investigating the impact of measurement error on social network measures [70]. We look at four node-level measurements of network error, i.e. degree centrality [19], clustering coefficient [73], network constraint [8] and eigenvector centrality [7]. For each node-level measure we compute a vector M. For example, in Figure 2, vector M has the degree centrality of G, Gand G″. Similar to the approach discussed in [70] we compute two vectors M2 and M2′ for graph G″ using each node level measure and compute Spearman's rho between these two vectors.

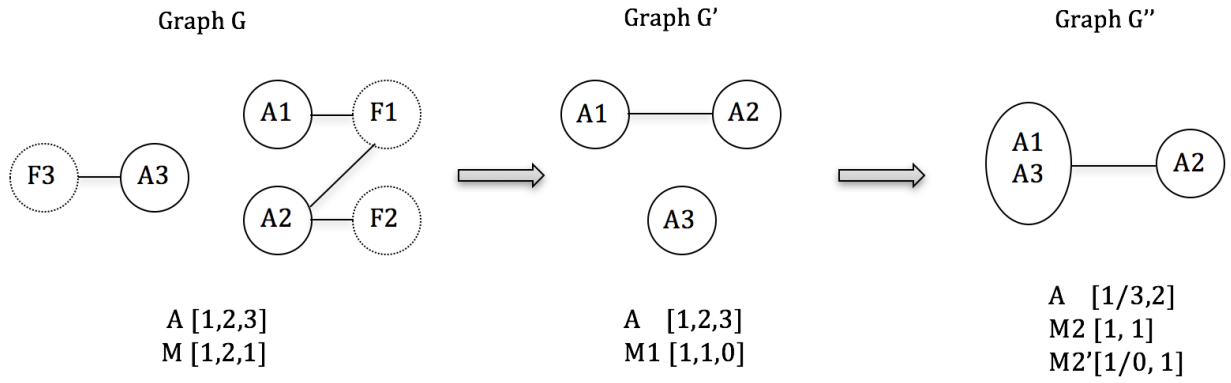

Figure 5.2: Understanding the impact of errors on a developer collaboration network

We obtain Spearman's rho for degree centrality to be 0.8619, clustering coefficient to be 0.8685, network constraint to be 0.8406 and eigenvalue centrality to be 0.8690. The correlations below 0.95 for any of these measures are considered to indicate major disruptions to the social network [70]. In our case all of these measures are well below 0.95. We can also look at the quantiles of these measures: for example one quarter of developers in the corrected network have 210 or fewer peers, but in the uncorrected network that figure is 113 peers. The eigen-centrality has an even larger discrepancy: for one quarter of developers it is below 0.024 for the corrected and 0.007 (or more than four times lower) for the uncorrected network.

5.5 Limitations

Our findings have a number of limitations. First, in the `Homonym` discovery phase, it is not clear if the proposed discovery process can identify all `Homonyms`. In fact, it is not clear what proportion of `Homonyms` may be left undiscovered, especially `Homonyms` associated with relatively few commits. Despite that, we feel it is important to have some systematic approach for `Homonym` discovery. Future research may be needed to determine how complete such process is.

Second, the proposed behavioural fingerprints may not be optimal for authorship assignment and other types of fingerprints may lead to a more accurate disambiguation. The results we got, however, show fairly high accuracy, suggesting that the specific choices are reasonable.

Third, the evaluation based on a sample of commits where developer ID was known but removed, may be different from commits where the developer ID was absent or was a `Homonym` representing multiple individuals. These more realistic scenarios, however, do not have the authorship data to validate our predictions. We, therefore, chose to validate the method on data where authorship is not in doubt. The validation of authorship on real `Homonyms` presents several challenges that we do not see how to overcome. First, a `Homonym`, unlike a regular ID, does not have valid contact information, second, it may be not easy for an actual author to remember/recognize the commit as their own. Finally, in

cases where authors seek anonymity, they may not desire to be contacted or may even be distressed at the ability of the algorithm to de-anonymize authorship.

To begin our research, the text we consider as an alias for identity are the commit messages of individuals from a very large collection of open source projects, in order to identify developers in open source software communities. While using text to identify individuals is a general method that can enable us to match users across different sources or platforms, it also has challenges and drawbacks such as the fact that many people may write similarly. This problem is even greater for the data set of commit messages we use in this study, as commit messages are typically concise and are heavily influenced by technical jargon pertaining in software projects. To solve this problem, we use a paragraph embedding technique by – Doc2Vec [34] which is an extension of Word2Vec by [42]. Doc2Vec produces vectors that embed semantics of the text in paragraphs or documents and therefore will be indicative of the behavioral pattern unique to an individual. Through comparing and matching these vectors we hope to identify individuals using different aliases.

Chapter 6

Overview, Discussion, and Conclusions

6.1 Overview

The objective of this work "Methods of Disambiguating and De-anonymizing Authorship in Large Scale Operational Data" was to correct operational data using identity disambiguation and de-anonymization by means of string comparison of developer IDs in conjunction with individual-specific activity-based fingerprints. Developer activities during the process of software development are recorded in a version control system. Data extracted from these tools (known as operational data) and has deficiencies, e.g., (1) missing data such as missing names and/or email address (2) incorrect data such as misspelled or inconsistent developer names and emails, all of which profoundly affect social network analysis.

In this thesis, we have proposed a framework named ALFAA (Active Learning Fingerprint Based Anti-Aliasing), and the implementation details of a prototype are outlined, which was built based on the framework. We also looked at errors that occur in operational data and found two broad categories using a qualitative approach known as open card sort. We refer to these errors as `Synonyms` and `Homonyms` errors in developer identity (IDs) that exist in operational data for a multitude of reasons. These two radically distinct types of errors need to be be treated separately – in particular, the second type of error, the `Homonyms`, cannot be resolved with traditional record matching techniques. Therefore, ALFAA offers solutions that augments "behavioral fingerprints"

to traditional string comparisons used in existing literature. These fingerprints include commit message text as they encode natural language used by the specific individual, times of a commit as they indicate the individual's work schedule, and files and projects modified during the commit as they indicate the scope of the individual's work. We use the string similarity between developer IDs and "fingerprint" the similarities of commits made by those developers as features that a supervised classifier can use to predict whether a pair of syntactically dissimilar IDs is a match. We find through detailed evaluation that our approach performs significantly better than existing state-of-the-art methods. In the following sections we review the research questions we started with, the answers we found through our work, and a summary of the contributions we made as a result. Finally, we discuss some potential future work in this domain.

6.2 Discussion: Primary Findings

In this section we discuss the primary findings of this thesis by answering each research question.

1. What are the most common reasons for identity errors in version control data?

   We found that identity errors occur for a number of reasons, and we broadly classified them into `Synonym` and `Homonym` errors. The reasons for identity errors include misspellings, case differences, name changes, changes in the order of names, abbreviations of names, transliteration and additions or deletions of characters such as spaces, periods etc., in the case of `Synonyms`. `Homonym` errors are introduced when IDs include generic roles of developers, names of projects, organizations, or, in certain cases, individuals trying to preserve anonymity. Our categorization of errors was done manually and was significant in helping us to design the corrective solutions introduced in ALFAA, as each type requires distinct approaches for correction.

2. Can we encode developer behavior from version control data to help correct identity errors?

Yes, our research shows that we can encode developer behaviour from version control data, that can help correct identity errors. Commit blobs extracted from version control data encapsulates a great deal of information about the individual making the commit. Among these information are names and email addresses associated with the commit (the combination of these two is referred to as the developer ID), the time during which the commit was made, and the files and projects that were modified in the commit. Traditional record matching techniques in software use comparisons of developer IDs (strings) to merge identities. Here we demonstrated that in fact other information found in a message can be used to encode developer behaviour that distinguishes the developers. We refer to this as "behavioural fingerprints" and our experiments showed that using these information was crucial especially in cases where developer IDs were inadequate. Furthermore, we find higher precision and recall errors when fingerprints metrics were added as features to the learner. This information plays a critical role in constructing ALFAA.

3. Does our approach improve upon existing matching techniques applied in the software engineering domain that is used in research and in commerce?

   We compared the results produced by the prototype implementation of ALFAA against current state-of-the-art research and commercial efforts. We found that ALFAA had 9.5% false positive rate compared to the 24.85% rate yielded by the recent research method. This was manually validated by checking a random sample of one thousand IDs. Thus, the recent research method to which we compared ALFAA was 2.61 times more prone to lumping error. We also evaluated our results against a commercial effort which mapped 10,344 distinct developer IDs to 8,840 developers. We mapped the same number of IDs to 6,271 developers using ALFAA.

4. What is the impact of identity errors on actual collaboration networks among developers?

   We found that identity errors have a significant impact on identity networks on collaboration networks. This conclusion was reached by means of the experiment

outlined in [70]. We looked at four node-level measurements of network error, i.e., degree centrality, clustering coefficient, network constraint and eigenvector centrality. We calculate correlations between vectors each of the raw and corrected networks for each of these measures. We found Spearman's Rho for degree centrality to be 0.8619, clustering coefficient to be 0.8685, network constraint to be 0.8406, and eigenvalue centrality to be 0.8690. According to [70], correlations below 0.95 for any of these measures are considered to indicate major disruptions to the network and in our case all measures were well below the threshold.

6.3 Summary of Contributions

A summary of the contributions separated into theoretical and practical contributions is provided below.

6.3.1 Theoretical Contributions

1. We propose ALFAA – a framework that uses behavioral fingerprints alongside traditional string similarity for identity correction

   Traditional methods of identity merge mostly use string comparisons of components found in an individuals ID and other on-the-surface attributes to merge identities. This requires preset heuristics and theoretical assumptions which is inadequate because data sets may differ between domains and practices may change over time. Through ALFAA, we propose a new approach that uses "behavioral fingerprints" for identity merge by using the information found in a commit blobs such as time stamps, files, and projects modified in a commit as well as the commit messages, along with string comparisons, to infer whether IDs associated with the commits belong to the same developer. Standalone behavioral fingerprints can also be used in the case of IDs that may have been used by multiple developers as in the case of `Homonym` errors for de-anonymization, since string comparisons in these cases are not meaningful. This concept can be extended to

identity merge in other domains by defining "fingerprint" metrics that apply to that domain. We believe and demonstrate in a detailed evaluation that incorporating "fingerprint" metrics had a significant positive impact on results.

2. Classification of errors in developer identity – i.e., `Synonym` and `Homonym` errors

   To be able to correct any error, it is important to understand the nature of the errors. In this work, we manually went through a list of developer identities and classified them into two large categories - `Synonym` and `Homonym` errors using a qualitative approach called the open-card sort method (Figure 3.4). `Synonym` errors occur when multiple aliases are used by the same individual and usually contain information about the identity of the individual in each of the aliases. For example, there might be similarities in names or user names among IDs. Homonym errors occur when multiple individuals/bots etc., use the same ID. These IDs almost never contain information about the identity of the individuals such as the name or user name so a user name cannot be assumed to represent an individual. For example, multiple individuals working for an organization may be using the same ID. Identifying these errors and understanding them helps in designing corrective algorithms focusing on the nature of the errors. Previous works on record linkage have not dealt with errors using any systematic classification and therefore, do not address some of the issues dealt with in this work; specifically, solutions have not been designed for `Homonyms`.

3. A systematic approach to discovering `Homonyms` in a large data set

   While `Synonyms` are relatively simple to detect using string similarity, it is much harder to detect and extract homonyms from a large collection of IDs. In this work, we have proposed a frequency based approach to detecting `Homonyms` and include a list of "`Homonym` keywords (Table 4.7)" as the most frequent Components of IDs were found to be the result of homonym errors. This keyword list can be used to extract `Homonyms` from other data sets by incorporating desired heuristics. For example, one can use a simple heuristic such as extract all IDs that have all components belonging

to the keyword list. Frequency analysis can also be used on other data sets to append to the existing list of keywords.

4. Using supervised learning in linking identities in software development context.

   Traditional identity resolution methods used in software engineering uses string similarity to compare names of individuals. Comparisons between components of names (first and last), email addresses and other manually designed heuristics are used in merging IDs. This can be problematic because names (or combination of first names, last names and usernames) almost never work as unique identifiers in large datasets. Furthermore, data from software tools often do not have meaningful strings to begin with as discussed in Chapter 3. Therefore, this work also includes exploration into other domains that have researched identity merge. For example, a research done on United States Patent and Trademark Office (USPTO) used supervised learning to disambiguate inventor records on patents [68] and attained error rates below 3% across all of the samples available to them. However, it is important to note that such techniques require a plethora of hand labeled records for training which is extremely hard, if not impossible, to obtain for our dataset. Another research used *Active Learning* [54] in disambiguating citation records - an efficient way of extracting observations for training set that is effective for the learner and requires minimum manual effort. In this study, we combined these approaches from a variety of research works previously done in other domains and applied them to our context.

5. Experiments to show that developer identities have impact on networks

   We conducted an experiment to prove that developer identities have impact on networks. We did this by comparing a network with errors to a corrected one as described in Section 5.4 following an approach outlined in [70]. We calculated correlations between vectors of the raw and corrected network for four node level network measure. We found that all of the measures have rho much lower than 0.95 – a threshold which indicates that the errors have significant impact on the network.

6.3.2 Practical Contributions

1. A very large hand-labeled training dataset -

   The prototype built using over 16,000 OpenStack developers employed a hand labeled training data set that has a high rate of agreement (about 98%) between two raters (both PhD student in CS). This required many hours of careful and labor intensive work including information gathering from various sources to make decisions on whether a pair of IDs should be labeled as a match or a non-match. This training data set was used to create models to predict matches/non-matches for every pair of developers in the data set as described in Chapter 4. The training data is available in a open source repository and can be used as is or added to in order to train new models.

2. Trained models

   We trained various models for the active learning phase. The pre-trained models available in the repository[1] can be used for extracting more effective training data (observations that are likely to be confused by a classifier) for the final classification phase.

3. Scripts for creating models using other data

   All codes used to build the prototype based on the proposed framework in this thesis is available in a public repository that can be found at https://github.com/ssc-oscar/fingerprinting. These scripts can be used on any set of IDs to build custom models provided with other training data.

4. A set of corrected developer identities

   Finally, we have provided a set of corrected developer IDs from the OpenStack project and a set of 18 other ecosystems as discussed in Chapter 5. These can serve as a much larger training set to build more robust models for any future work.

5. Disambiguation code for distributed computing implementation

[1] https://github.com/ssc-oscar/fingerprinting

We ran the proposed algorithm on a large set constituting over 2 Million IDs. This required us to learn and implement code using pbdR which is a set of R packages for the distributed environment. We ran this code on the Titan supercomputer (hosted at Oak Ridge Leadership Computing Facility) which is a hybrid architecture using about 1.5 million core hours of computation time. These scripts are also available in the public repository mentioned above.

6. Trained Doc2Vec models on over a billion of commit messages

   For this work, we have trained a number of models to extract document vectors using different hyper-parameter combinations as outlined in Appendix A. These were trained on over 1 billion commit messages and required many days worth of computation time. We believe that these pre-trained models will save time and aid research related to software repositories.

6.4 Future Work

As acknowledged in Section 5.5, we are not confident that the proposed homonym discovery process can identify all homonyms. Manual checks have shown that there are homonyms that occur more infrequently and may include no keyword, but only special characters such as "!!!! <!!!!>". We believe that the homonym correction process can be further refined and that various combinations of the "fingerprint" can be experimented with to create models. In this work we propose using text similarity, files and projects modified to correct homonym errors. Models built from other combinations can be compared to the current results to empirically derive the best set of "fingerprints" for version control data. There may also be room for improvement in building the Doc2Vec models used for embedding commit messages, by tuning hyper-parameters as these models have significant roles in the homonym resolution process.

This framework can be used to build a tool for disambiguation. For example,an user can provide a list of IDs and have the tool to extract all commits associated with the provided list of IDs, compute string similarity and behavioral fingerprint similarity and

feed the information to a classifier for training. The classifier can then predict an outcome (match or non-match) for each pair of IDs in the list. Finally, transitive closure can be calculated to get the final clusters of IDs representing each individual. While all of these have been done in various scripts to create the prototype, work remains in making this process seamless and into a functional software.

Bibliography

[1] Amreen, S. and Mockus, A. (2017). Position paper: Experiences on clustering high-dimensional data using pbdr. In *Proceedings, International Workshop on Software Engineering for High Performance Computing in Computational and Data-enabled Science and Engineering*, Denver,Colorado, uSA. 114

[2] Badashian, A. S., Esteki, A., Gholipour, A., and Abram Hindle, E. S. (2014). Involvement, contribution and influence in github and stack overflow. In *CASCON '14 Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, pages 19–33, Markham, Ontario, Canada. 6, 9

[3] Baltes, S., Kiefer, R., and Diehl, S. (2018). Usage and attribution of stack overflow code snippets in github projects. In *Empirical Software Engineering*. Springer. 9

[4] Basili, V., Rombach, D., Schneider, K., Kitchenham, B., Pfahl, D., and Selby, R. (2007). *Empirical Software Engineering Issues. Critical Assessment and Future Directions: International Workshop, Dagstuhl Castle, Germany, June 26-30, 2006, Revised Papers*, volume 4336. Springer. 6

[5] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A. (2006). Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 137–143, New York, NY, USA. ACM. 7, 18, 19, 22, 46, 48, 61

[6] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R. C. (1997). *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA. 114

[7] Bonacich, P. (1987). Power and centrality: A family of measures. *American Journal of Sociology*, 92(5):1170–1182. 82

[8] Burt, R. S. (1992). *Structural Holes*. Harvard University Press. 82

[9] Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. (2006). Identification of coordination requirements: implications for the design of collaboration and

awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 353–362. ACM. 82

[10] Chen, H.-M., Kazman, R., and Haziyev, S. (2016). Strategic prototyping for developing big data systems. In *IEEE Software*, volume 33, pages 36–43. IEEE. 109

[11] Christen, P. (2006a). A comparison of personal name matching: Techniques and practical issues. *Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06)*, pages 290–294. 22

[12] Christen, P. (2006b). A comparison of personal name matching: Techniques and practical issues. In *Sixth IEEE International Conference on Data Mining - Workshops (ICDMW'06)*, pages 290–294. 37

[13] Cohen, W. and Richman, J. (2001). Learning to match and cluster entity names. In *In ACM SIGIR-2001 Workshop on Mathematical/Formal Methods in Information Retrieval*. 28

[14] Cohen, W. W., Ravikumar, P., and Fienberg, S. E. (2003). A comparison of string metrics for matching names and records. In *KDD WORKSHOP ON DATA CLEANING AND OBJECT CONSOLIDATION*. 7, 9, 17

[15] Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176. 47

[16] Dunn, H. L. (1946). Record linkage. *American Journal of Public Health and the Nations Health*, 36(12):1412–1416. PMID: 18016455. 3, 15

[17] Fellegi, I. P. and Sunter, A. B. (1969). A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210. 4, 12, 16

[18] Fleming, L., King, C., and Juda, A. I. (2007). Small worlds and regional innovation. *Organization Science*, 18(6):938–954. 17, 23

[19] Freeman, L. C. (1978). Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215 – 239. 82

[20] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary. 114

[21] German, D. and Mockus, A. (2003). Automating the measurement of open source projects. In *Proceedings of the 3rd workshop on open source software engineering*, pages 63–67. University College Cork Cork Ireland. 7, 18

[22] Gharehyazie, M., Posnett, D., Vasilescu, B., and Filkov, V. (2015). Developer initiation and social interactions in oss: A case study of the apache software foundation. *Empirical Software Engineering*, 20(5):1318–1353. 18

[23] Goeminne, M. and Mens, T. (2013). A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971 – 986. 18, 20

[24] Gorton, I., Bener, A. B., and Mockus, A. (2016). Software engineering for big data systems. *IEEE Software*, 33(2):32–35. 10, 32

[25] Hallgren, K. A. (2012). Computing inter-rater reliability for observational data: an overview and tutorial. *Tutorials in quantitative methods for psychology*, 8(1):23. 78

[26] Holmes, D. and McCabe, M. C. (2002). Improving precision and recall for soundex retrieval. In *Proceedings. International Conference on Information Technology: Coding and Computing*, pages 22–26. IEEE. 38

[27] Iyengar, V. S., Apte, C., and Zhang, T. (2000). Active learning using adaptive resampling. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 91–98, New York, NY, USA. ACM. 18

[28] Jaro, M. A. (1989). Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420. 47

[29] Kouters, E., Vasilescu, B., Serebrenik, A., and van den Brand, M. G. J. (2012). Whos who in gnome: using lsa to merge software repository identities. In *28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 22, 38, 43, 48

[30] Kukich, K. (1992). Techniques for automatically correcting words in text. *Acm Computing Surveys (CSUR)*, 24(4):377–439. 38

[31] Lait, A. and Randell, B. (1996). An assessment of name matching algorithms. *Technical Report Series-University of Newcastle Upon Tyne Computing Science*. 38

[32] Lau, J. H. and Baldwin, T. (2016). An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv preprint arXiv:1607.05368*. 108

[33] Lawrence, S., Giles, C. L., and Bollacker, K. (1999). Digital libraries and autonomous citation indexing. *Computer*, 32(6):67–71. 17

[34] Le, Q. and Mikolov, T. (2014). Distributed representation of sentences and documents. In *Proceedings of the 31 st International Conference on Machine Learning*, volume 32, Beijing,China. JMLR. 52, 84, 105

[35] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. 47

[36] Li, G.-C., Lai, R., DAmour, A., Doolin, D. M., Sun, Y., Torvik, V. I., Yu, A. Z., and Fleming, L. (2014). Disambiguation and co-authorship networks of the u.s. patent inventor database (19752010). *Research Policy*, 43(6):941 – 955. 23

[37] Li, Y., Peng, Y., Zhang, Z., Xu, Q., and Yin, H. (2017). Understanding the user display names across social networks. In *International World Wide Web*, Perth, Australia. 21

[38] Luhn, H. P. (1957). A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of research and development*, 1(4):309–317. 50

[39] Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., and Mockus, A. (2019). World of code: An infrastructure for mining the universe of open source vcs data. In *Proceedings of the 2019 International Conference on Mining Software Repositories*, MSR '19. 32, 34, 36

[40] Martinez-Romo, J., Robles, G., Gonzalez-Barahona, J. M., and Ortuño-Perez, M. (2008). Using social network analysis techniques to study collaboration between a floss community and a company. In Russo, B., Damiani, E., Hissam, S., Lundell, B., and Succi, G., editors, *Open Source Development, Communities and Quality*, pages 171–186, Boston, MA. Springer US. 6

[41] McCallum, A. and Wellner, B. (2003). Object consolidation by graph partitioning with a conditionally-trained distance metric. In *KDD Workshop on Data Cleaning, Record Linkage and Object Consolidation*. Citeseer. 5

[42] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*. 84, 105

[43] Mockus, A. (2009). Succession: Measuring transfer of code and developer productivity. In *2009 International Conference on Software Engineering*, Vancouver, CA. ACM Press. 51

[44] Mockus, A. (2014). Engineering big data solutions. In *ICSE'14 FOSE*, pages 85–99. 5

[45] Narayanan, A., Paskov, H., Gong, N. Z., Bethencourt, J., Stefanov, E., Shin, E. C. R., and Song, D. (2012). On the feasibility of internet-scale author identification. In *Security and Privacy (SP), 2012 IEEE Symposium*, San Francisco, California, USA. 25, 52

[46] Navarro, G. (2001). A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88. 20, 38

[47] Nesbitt, A. and Nickolls, B. (2017). Libraries.io Open Source Repository and Dependency Metadata. 80

[48] Newcombe, H. B., Kennedy, J. M., Axford, S. J., and James, A. P. (1959). Automatic linkage of vital records. *Science*, 130(3381):954–959. 4, 16

[49] Ostrouchov, G., Chen, W.-C., Schmidt, D., and Patel, P. (2012). Programming with big data in r. 81

[50] Pollock, J. J. and Zamora, A. (1984). Automatic spelling correction in scientific and scholarly text. *Communications of the ACM*, 27(4):358–368. 48

[51] Poncin, W., Serebrenik, A., and v. d. Brand, M. (2011). Process mining software repositories. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 5–14. 22

[52] Řehůřek, R. and Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta. ELRA. 27, 109

[53] Robles, G. and Gonzalez-Barahona, J. M. (2005). Developer identification methods for integrated data from various sources. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA. ACM. 18, 22

[54] Sarawagi, S. and Bhamidipaty, A. (2002). Interactive deduplication using active learning. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 269–278, New York, NY, USA. ACM. 4, 24, 28, 54, 55, 90

[55] Sariyar, M. and Borg, A. (2010). The recordlinkage package: Detecting errors in data. *The R Journal*, 2(1):61–67. 28, 48

[56] Silvestri, G., Yang, J., Bozzon, A., and Tagarelli, A. (2015). Linking accounts across social networks: the case of stackoverflow, github and twitter. In *International Workshop on Knowledge Discovery on the WEB*, pages 41–52. 13, 21

[57] Singh, J. (2005). Collaborative networks as determinants of knowledge diffusion patterns. *Management Science*, 51(5):756–770. 23

[58] Smalheiser, N. R. and Torvik, V. I. (2009). Author name disambiguation. *Annual review of information science and technology*, 43(1):1–43. 70

[59] Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21. 50

[60] Steorts, R. C., Ventura, S. L., Sadinle, M., and Fienberg, S. E. (2014a). A comparison of blocking methods for record linkage. In Domingo-Ferrer, J., editor, *Privacy in Statistical Databases*, pages 253–268, Cham. Springer International Publishing. 9

[61] Steorts, R. C., Ventura, S. L., Sadinle, M., and Fienberg, S. E. (2014b). *A Comparison of Blocking Methods for Record Linkage*, pages 253–268. Springer International Publishing, Cham. 24

[62] Su, J., Shukla, A., Goel, S., and Narayanan, A. (2017). De-anonymizing web browsing data with social networks. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1261–1269. International World Wide Web Conferences Steering Committee. 24

[63] Tejada, S., Knoblock, C. A., and Minton, S. (2002). Learning domain-independent string transformation weights for high accuracy object identification. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 350–359, New York, NY, USA. ACM. 5

[64] Thung, F., Bissyand, T. F., Lo, D., and Jiang, L. (2013). Network structure of social coding in github. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 323–326. 9

[65] van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*. 109

[66] Vasilescu, B., Serebrenik, A., and Filkov, V. (2015). A data set for social diversity studies of github teams. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 514–517. ACM. 4, 18, 75

[67] Venkataramani, R., Gupta, A., Asadullah, A., Muddu, B., and Bhat, V. (2013). Discovery of technical expertise from open source code repositories. In *Proceedings*

of the 22Nd International Conference on World Wide Web, WWW '13 Companion, pages 97–98, New York, NY, USA. ACM. 9

[68] Ventura, S. L., Nugent, R., and Fuchs, E. R. (2015). Seeing the non-stars: (some) sources of bias in past disambiguation approaches and a new public tool leveraging labeled records. *Elsevier*. 4, 5, 7, 9, 13, 17, 23, 70, 90

[69] Virtanen, T. (2016). Git for computer scientists. 33

[70] Wang, D. J., Shi, X., McFarland, D. A., and Leskovec, J. (2012). Measurement error in network data: A re-classification. *Social Networks*, 34(4):396–409. 6, 9, 82, 83, 88, 90

[71] Wang, M., Tan, Q., Wang, X., and Shi, J. (2018). De-anonymizing social networks user via profile similarity. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pages 889–895. IEEE. 53

[72] Wasserman, S. and Faust, K. (1994). *Social network analysis: Methods and applications*, volume 8. Cambridge university press. 9

[73] Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393:440–442. 82

[74] W.Cohen, W. and Richman, J. (2002). Learning to match and cluster large high-dimensional data sets for data integration. In *SIGKDD*, volume 32, Edmonton,Alberta,Canada. ACM. 10, 21

[75] Wiese, I. S., d. Silva, J. T., Steinmacher, I., Treude, C., and Gerosa, M. A. (2016). Who is who in the mailing list? comparing six disambiguation heuristics to identify multiple addresses of a participant. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 345–355. 18

[76] Winkler, W. E. (1999). The state of record linkage and current research problems. In *Statistical Research Division, US Census Bureau*. Citeseer. 18

[77] Winkler, W. E. (2006). Overview of record linkage and current research directions. Technical report, BUREAU OF THE CENSUS. 9, 10, 17

[78] Winkler, W. E. (2014). Matching and record linkage. *WIREs Comput Stat*, 6:313–325. 17, 18

[79] Wu, S. (2013). A review on coarse warranty data and analysis. *Reliability Engineering & System Safety*, 114:1 – 11. 5

[80] Xiong, Y., Meng, Z., Shen, B., and Yin, W. (2017). Mining developer behavior across github and stackoverflow. In *The 29th International Conference on Software Engineering and Knowledge Engineering*, pages 578–583. 18

[81] Yancey, W. E. (2005). Evaluating string comparator performance for record linkage. *Statistics*, 5. 38

[82] Zheng, Q. and Mockus, A. (2015). A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ESEC/FSE'15, pages 637–648, New York, NY, USA. ACM. 29

Appendices

Appendix A

Testing Feasibility of Using Doc2Vec on Commit Message Text

Experiment 1: Training and Testing Doc2Vec Models

Before we conclude that we can use the Doc2Vec algorithm for our research, we conducted an experiment to test its viability for our data set. We use Gensim's implementation of Doc2Vec to carry out this experiment. Doc2Vec, a paragraph embedding technique [34], uses an unsupervised approach to learn fixed-length feature representations from variable length pieces of texts, such as sentences, paragraphs or even documents, thus, producing what is referred to as *Paragraph Vectors*. This is an extension of an earlier work done by Mikolov et. al. on a word embedding technique called Word2Vec [42].

Training

In the training phase, we create a number of subsets of commit messages made by each user and replace the IDs associated with them with synthetic dummy IDs. The Doc2Vec algorithm generates vectors for each of the IDs and our goal is to see whether vectors generated from different synthetic IDs that map to the same real ID are highly similar – and can group IDs with highly similar vectors which would represent an individual. This would mean that the document embedding algorithm is able to learn from the style of commit message text of each individual and is able to differentiate it from others.

To conduct this experiment, we randomly selected 21 users and approximately 70,000 commit messages generated by them. We grouped 20 commit messages from the same developer into a single document and provided a synthetic label for it. A map of the synthetic label to the actual label is kept in a separate file. Therefore, we have 3500 documents (and the same number of synthetic labels) for training the Doc2Vec model. The label for each document is the developer's ID – a name <email> combination. Gensim's implementation of Doc2Vec has 3 models as shown in Table A.1, where two of the models are a variant of *Distributed Memory* (PV-DM) and the other is *Distributed bag-of-words* (PV-DBOW). We conduct our experiment on all 3 models to understand which model yields the best result so that we can generate document vectors on our whole data set using one particular model.

We use the functions shown in Table A.2 from Gensims Doc2Vec API to conduct the experiment.

Testing

To test the model, 30 documents are randomly selected from each user (each user has a varying number of documents therefore, selection range is adjusted accordingly). Thus, we have 630 documents for testing. Once we have the trained Doc2vec model, we input a test document and it produces the ID of the most similar document (based on the cosine similarity of the document vectors) to the input. We train each model with 5000 epochs and repeat this process for the three Doc2Vec models described above. We output the number and percentage of documents each of the Doc2Vec models can identify correctly. Figure A.1 illustrates the experimental setup and how the results are evaluated.

Results

We evaluate each of the models in the following process:

1. Performance in finding the most similar document - As the most similar document returned by the most_similar() function returns the label of the same document (similarity = 1.0), we exclude that and consider the next most similar document

Table A.1: The 3 models implemented in Gensim's Doc2Vec

Model No.	Model Name	Description and Parameters
1	Distributed Memory	Doc2Vec(dm=1, dm_concat=1, size=200, window=10, negative=20, hs=0, min_count=5, workers=cores)
2	Distributed bag-of-words	Doc2Vec(dm=0, size=200, negative=20, hs=0, min_count=5, workers=cores)
3	Distributed Memory	Doc2Vec(dm=1, dm_mean=1, size=200, window=10, negative=20, hs=0, min_count=5, workers=cores)

[1] Dm_concat – If 1, use concatenation of context vectors rather than sum/average; default is 0 (off)
[2] Size - Dimensionality of the feature vector
[3] Window - Maximum distance between the predicted word and context words, optimal is 10 words
[4] Negative – If > 0, negative sampling will be used,
the int for negative specifies how many "noise words" should be drawn (usually between 5-20)
[5] hs - Hierarchical softmax will be used for model training
[6] Min_count - Ignore words with total frequency lower than this

Table A.2: Gensim's Doc2Vec API used

Function	Description
Model.build_vocab()	Build vocabulary from sequence of sentence
Model.reset_from()	Reuse sharable structure from other models
Model.train()	Update the models neural weight from a sequence of sentence
Model.docvecs.most_similar()	returns most similar documents to the one passed

Table A.3: Summary of results: Classifying 630 documents

Measure	Correctly	%Accuracy
Top 1	497	78.8
Top 1, 2 and 3	410	65.1
Any 3	469	74.4
All 5	219	34.8

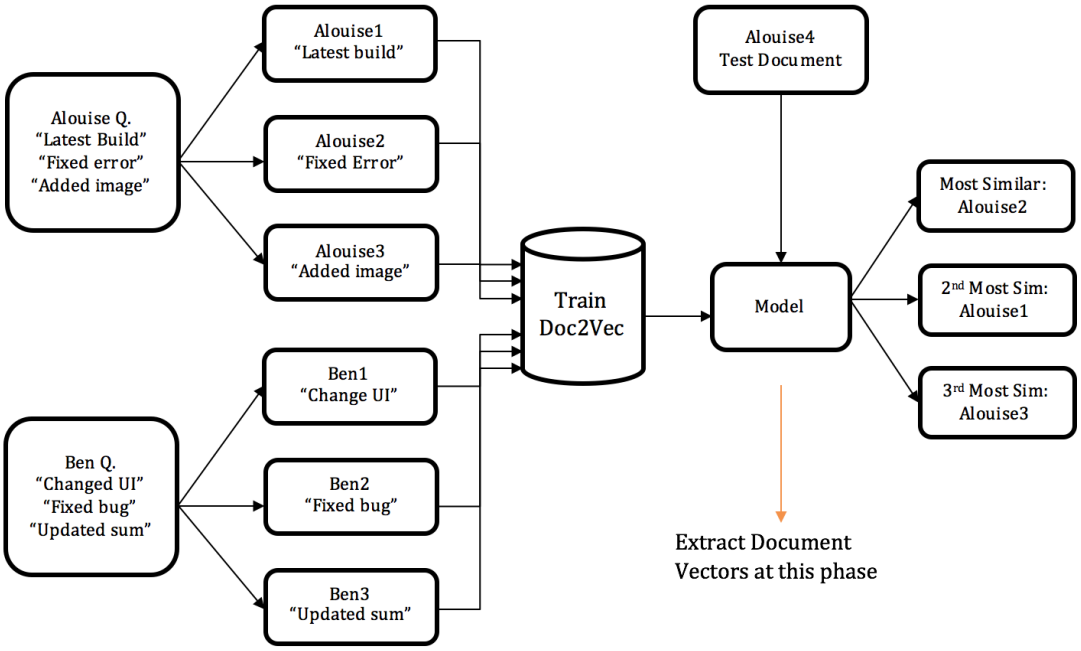

Figure A.1: Experimental setup to test Doc2Vec for our dataset

for this measure. Each model returned a synthetic label and it matched the correct ID for model 1 56%, model 2, 62% and model 3, 79% of the times respectively.

2. Performance in finding the most similar and second most similar documents - We also checked the performance of the three models on finding the second most similar document (again, excluding itself). Quite similar to the previous test, model 1 performed the worst (30%) and model 3 performed the best (65%).

3. Performance in finding 3 most similar documents- In this case, we only count an output as correct if all three labels returned were correct. Model 1 had an accuracy of 18%, model 2 had an accuracy of 39% and model 3 had an accuracy of 52%.

After performing this experiment, we learned that the PV-DM model, where the feature vectors are summed/averaged, yield better results than the model with feature vectors concatenated or the PV-DBOW model as shown in Table A.3. This finding contradicted with the findings of the an empirical study [32] where the authors found that

DBOW is a superior model to PV-DM. We also had a horizontal prototyping exercise [10] whereby we determined the viability of using the Doc2Vec algorithm on our data set of commit messages. For the rest of our work, we therefore use the PV-DM model to train and extract feature vectors.

Building Doc2Vec Models on the Large Dataset

Following the prototyping exercise described above, we built the Doc2Vec models on the entire data set (the filtered data set has 2.3 Million tags and 250 Million commit messages). Each commit message represents a single document and the label is the associated ID. An user writes multiple commit messages, as shown in Figure 4.2 (Chapter 3), therefore multiple documents will have the same label. We choose not to group the commit messages into a large document in order to retain the semantics of the texts as much as possible. Once Doc2Vec is trained using the parameters discussed below, it produces a single vector corresponding to each tag.

We build two distributed memory models and generate document vectors for each of the 2.3 Million tags. Model 1 uses the mean of the context vectors produced from each document to generate a vector for a tag, has a window size of 10 and ignores all words that has a frequency lower than 5. Model 2 (default) uses the sum of the context vectors and a window size 3 and ignores words that appear less than 15 times. Both the models yield vector size 200 (Vector size is short enough but long enough to hold the information, for example, [52] used vector size 300 for testing on a WIKI and AP-NEWS dataset) The results of Doc2Vec training can be sensitive to parameterization, and therefore, it is extremely important for us to understand how the quality of the vectors depend upon each parameter. For example, in Model 2 we have a higher threshold for sub-sampling as high frequency words often provide little information and filtering them out while training will result in higher training speed [65].

These vectors are stored in a compressed file in a ID;vector format. We refer to the size of these vectors as dimensions – we, thus, have 200 dimensional vectors.

Model 1:
mod = Doc2Vec (dm = 1, dm_mean = 1, size = 200, window = 10, negative = 20, hs = 0, min_count = 5, workers = cores)

Model 2:
mod = Doc2Vec (dm = 1, dm_mean = 0, size = 200, window = 3, negative = 20, hs = 1, min_count = 15, workers = cores-2)

Experiment 2: Validation using other attributes

We conducted an experiment considering the similarity between names (string similarity) and projects the user worked on. To carry out this test, we randomly sampled users from 20% of the users who have the highest message count. We then obtained the top 25 matches for each of the users generated by the Doc2Vec model based on their similarity score. From each of these sets, we manually selected user pairs that have similar IDs (string similarity) i.e., "pradeep:pradeep@arrayfire.com" and "Pradeep: pradeep@arrayfire.com" and found the number of project overlaps between them.

Table A.4 shows some of the selected tags from our data set. $P(T_n)$ represents the number of projects user T_n worked on. The SimScore column represents the cosine similarity between the document vectors of the users. We see that 23 out of 26 projects (88%) of user "Hanul:hanul@hanul.me" overlaps with projects of user "Hanul:mr@hanul.co" who had 81 projects. This gives us some confidence that both these tags belong to the same user. This exercise helped us build more confidence on identifying same users through using commit message similarity as an underlying identifier. It also gave us a plausible way of identifying the primary user name and email i.e. the tag of an entity.

Table A.4: Results from Experiment 2: IDs with high text similarity score had high overlap in projects, increasing confidence in the effectiveness of using Doc2Vec for commit messages

Tag(T_1)	P(T_1)	Tag(T_2)	P(T_2)	SimScore	Overlap
Hanul: hanul@hanul.me	26	Hanul: mr@hanul.co	81	0.807	23
Cesar Ferreira: cesar.manuel.ferreira@gmail.com	41	cesar: cesar.manuel.ferreira@gmail.com	5	0.610	3
Cesar Ferreira: cesar.manuel.ferreira@gmail.com	41	Csar: otymonn@gmail.com	3	0.581	0
pradeep: pradeep@arrayfire.com	57	Pradeep: pradeep@arrayfire.com	55	0.780	53
pradeep: pradeep@arrayfire.com	57	pradeep: pradeep.garigipati@gmail.com	12	0.551	11
Eva Shon: eshon@opengeo.org	112	eshon@opengeo.org: eshon@opengeo.org	42	0.724	41
Eva Shon: eshon@opengeo.org	112	Eva Shon: eshon@boundlessgeo.com	262	0.653	18

Transitive Closure Clustering

The first test we designed to analyze the document vectors is to group the most similar vectors together by using their similarity scores produced by the most_similar() API in Gensim's implementation. Through transitive closure clustering we hope to construct a structure that can answer connectivity related questions. For example, suppose Node A is connected to Node B and Node B is connected to Node C. After a network has been constructed through transitive closure, Node C will be connected to Node A. In the context of our work, if two nodes, each representing the alias of an individual, are connected it should signify that both the aliases belong to the same individual. Our goal for this experiment was to test whether similarity scores (given certain thresholds) yield reasonable (cluster sizes are not too large and have more than one element) groups or clusters of identities.

To perform this experiment we implemented two methods of clustering: directed and undirected transitive closure clustering. For both methods, we consider a high similarity threshold, i.e. greater than or equal to 0.9, between vectors.

1. Directed Transitive Closure Clustering

For each of the document vectors representing a tag, we get the tags that are most similar to it (similarity score >=0.9) and add to the cluster and remove the ID from the available pool of IDs. We then get the ID that is most similar to the previous ID and add it to the cluster and repeat this until no similar tags can be found. In other words, there are no tags in the remaining pool that has a similarity score greater than or equal to 0.9 to the current ID. Table A.5 shows the results of this experiment, where the total number of clusters formed were 390,310, 77% of which were single element clusters. This approach yielded too many stray tags i.e., large number of clusters with a single element. As this result was unsatisfactory, we repeated the experiment without directionality.

2. Undirected Transitive Closure Clustering

   This approach is very similar to the directed clustering technique with the except – tags are not removed from the pool of tags once they are drawn into a cluster. Therefore, a tag can belong to multiple clusters. The number of clusters formed with this approach is 89,396. In this case, we were able to find 57,527 small clusters of 2 elements (64%). However, this approach also formed a large single cluster that had 311,679 elements. Table A.6 shows the results of this experiment. Through a manual checking process (from randomly drawn 10 samples) we determined that 6 clusters had syntactically similar tag-pairs from the 2 element clusters. 5 clusters had syntactically similar tags (two or more) from the 3 element clusters and 3 clusters had syntactically similar tags (two or more) from the 4 element clusters. Our conclusion from this test was that transitive closure leads to either a large

Table A.5: Distribution of the transitive closure clusters – with directionality

Cluster Size	Number of Clusters
1 (smallest cluster)	300763
199 (third largest cluster)	1
504 (second largest cluster)	1
60664 (largest cluster)	1

Table A.6: Distribution of the transitive closure clusters – directionality removed

Cluster Size	Number of Clusters
2 (smallest cluster)	57527
3 (second smallest cluster)	15022
4 (third smallest cluster)	6533
352 (fourth largest cluster)	1
429 (third largest cluster)	1
861 (second cluster)	1
311679 (largest cluster)	1

number of extremely small clusters of a single element or a long chain (one very large cluster) of hundreds of thousands of elements. Therefore, transitive closure will not be effective with the current technique.

Clustering in Distributed Systems

Another approach we attempted was to cluster users based on the document vectors of each user. Our goal for this experiment was to test how traditional clustering methods such as K-Means perform on our highly dimensional data set. Such clustering algorithms are computationally intensive and require a large amount of memory for intermediate data structures i.e. the distance matrix. To be more specific, the data we attempted to cluster will result in a 2.3M X 2.3M (approximately 32 TB) distance matrix with a potential need for at least twice that amount of memory during computations.

These highly computational and space intensive tasks requires capacity that exceed that of commercially available clusters. Therefore, to enable ourselves to experiment efficiently, we obtained access to Titan[1]. This gave us access to 18,688 compute nodes and 710 TB of total system memory with 32 GB of memory per node on a supercomputing system with a hybrid architecture. To allow us to focus more on the experimentation and increase productivity, we chose pbdR - an effective middleware for using R in a distributed environment.

[1]https://www.olcf.ornl.gov/titan/

Using the pbdR packages

The programming with big data in R (pbdR) project is a set of highly scalable packages for distributed computing and profiling in data science. It allows a scientist to use the R programming language without, at least nominally, having to master many layers of HPC infrastructure, such as OpenMPI [20] and ScalaPACK [6]. The packages within this project interface with softwares for distributed systems such as MPI, scaLAPACK, PAPI and is designed to work best on large distributed platforms. In this chapter, we focus on the **pmclust** package which implements model-based and k-means clustering for high dimensional and ultra large data on a distributed environment. We also report our experience using pbdR through conducting a number of experiments discussed in [1].

Appendix B

List of Publications

Publications related to this research:

1. Amreen, S. and Mockus, A., Zheng, Y., Bogart, C., Zaretzki, R. (2019) ALFAA: Active Learning Fingerprint Based Anti-Aliasing for Correcting Developer Identity Errors in Version Control Data, arXiv preprint arXiv:1901.03363

2. Ma, Y., Bogart, C., Amreen, S., Zaretzki, R. and Mockus, A. (2019) World of Code: An Infrastructure for Mining the Universe of Open Source VCS Data. International Conference in Mining Software Repositories, Montreal, QC, Canada.

3. Amreen, S. and Mockus, A. (2017).Position paper: Experiences on clustering high-dimensional data using pbdr. Proceedings of the 2017 International Workshop on Software Engineering for High Performance Computing in Computational and Data-enabled Science and Engineering, Denver,Colorado, USA.

4. Agrawal, K., Amreen, S. and Mockus, A. (2015). Commit quality in five high performance computing projects. Proceedings of the 2015 International Workshop on Software Engineering for High Performance Computing in Science.

Vita

Sadika Amreen was born in Dhaka, Bangladesh. She obtained her Bachelor's degree in Computer Science and Engineering from BRAC University, Bangladesh in 2012. She enrolled into a Bachelor's to PhD program in 2013 and obtained her Masters in Computer Science along the way at the University of Tennessee. During the summers of graduate school, Sadika has worked at various organizations including Oak Ridge National Lab as a research intern, Pilot Flying J as a business intelligence intern, and at eBay as an applied research intern. She is also co-founded a student organization at the Department of Electrical Engineering and Computer Science whose mission is to recruit, mentor and retain women entering undergraduate and graduate programs at the department. She received her Doctorate in Philosophy in Computer Science from the Department of Electrical Engineering and Computer Science at the University of Tennessee in April 2019.