A Methodology for Measuring FLOSS Ecosystems

Sadika Amreen, Dr. Bogdan Bichescu, Dr. Randy Bradley, Tapajit Dey, Yuxing Ma, Dr. Audris Mockus, Sara Mousavi, and Dr. Russell Zaretzki

Abstract

FLOSS ecosystem as a whole, is a critical component of world's computing infrastructure, yet not well understood. In order to understand it well, we need to measure it first. We, therefore, aim to provide a framework for measuring key aspects of the entire FLOSS ecosystem. We first consider the FLOSS ecosystem through lens of a supply chain. The concept of supply chain is the existence of series of interconnceted parties/affiliates each contributing unique elements and expertise so as to ensure a final solution is accessible to all interested parties. This perspective has been extremely successful in helping allowing companies to cope with multifaceted risks caused by the distributed decision making in their supply chains, especially as they have become more global. Software ecosystems, similarly, represent distributed decisions in supply chains of code and author contributions, suggesting that relationships among projects, developers, and source code have to be measured. We then describe a massive measurement infrastructure involving discovery, extraction, cleaning, correction, and augmentation of publicly available open source data from version control systems and other sources. We then illustrate how the key relationships among the nodes representing developers, projects, changes, and files can be accurately measured, how to handle absence of measures for user base in version control data, and, finally, illustrate how such measurement infrastructure can be used to increase knowledge resilience in FLOSS.

Sadika Amreen · Tapajit Dey · Yuxing Ma · Sara Mousavi
Graduate Students, Department of Electrical Engineering and Computer Science

Dr. Audris Mocus,
Ericsson-Harlan D. Mills Chair Professor, Department of Electrical Engineering and Computer Science ph: +1 865 974 2265, fax: +1 865 974 5483, e-mail: audris@utk.edu,

Dr. Bogdan Bichescu
Associate Professor, Haslam College of Business, e-mail: bbichescu@utk.edu

Dr. Randy Bradley
Assistant Professor, Haslam College of Business, e-mail: rbradley@utk.edu

Dr. Russell Zaretzki
Heath Faculty Fellow, Joe Johnson Faculty Research Fellow, Associate Professor, Haslam College of Business, e-mail: rzaretzk@utk.edu,
University of Tennessee, Knoxville, TN-37996, USA

1 Introduction

Open source is, perhaps, the least understood among the revolutionary inventions of the humankind. This is, perhaps, not very surprising because just two decades ago it was a mere curiosity, yet now, with its exponential growth, it has reached all corners of the society. This lack of understanding however, is not excusable, because much of the societies critical infrastructure and the ability to innovate depends on the heath of FLOSS (Free/Libre and Open Source Software).

Here, we attempt to alleviate this gap in understanding by proposing a measurement infrastructure capable of encompassing the entire FLOSS ecosystem in the large. To do that, we start from introducing conceptual framework of supply chains and adapting it to the unique features of the FLOSS ecosystem. In particular, we define software supply chain as the collection of developers and software projects producing new versions of the source code. This supply chain analogy provides us with key concepts of the abstract network involving nodes that represent developers, changes, projects, and files. The production process involves creating new versions of files via atomic increments that deliver specific value (commits). We then proceed to operationalize these and additional concepts from bottom up, i.e., from publicly available atomic records representing code changes.

The process of collecting and extracting this public data is involved due to lack of a single global registry of all FLOSS projects, the need to extract data from git database, need to store a petabyte of the data, and the need to covert it into a form so that the necessary measures could be calculated.

Before we can engage in the construction of the supply chain relationships, we need accurate identification of developers and projects and the relationships among them. Developers' identities are often misspelled, while projects may represent temporary forks of other projects. Both issues need to be addressed. Once the basic data has been cleaned and corrected in this way, we can engage in estimation of direct relationships that involve five basic types:

- Authorship links file(s) modified with the author and includes basic data in a commit: date, and commit message
- Version history links changes (and, therefore, versions of a file) trough a parent child relationship with each commit having zero or more parent commits.
- Static dependence links source code files via package use or call-flow dependencies.
- Project inclusion links projects (VCS repositories) with changes, and all versions of files contained therein.
- Code copy dependencies identify instances of code between specific versions of files and, in conjunction with version history, can be used to create Universal Version History that breaks project boundaries.

In combination, these dependencies induce additional networks, for example, the knowledge flow graph of developers connected trough files they modify in succession or upstream/downstream collaboration graph linking developers working on projects that have static dependencies.

Once the data for software supply chains is produced, the types of attributes that are directly available, are limited and we, typically, need to augment basic data with quantities that may reside in other data sources, for example, responsiveness that resides in projects' issue trackers or Q&A websites, or may be entirely unavailable, for example, the number of end users and, therefore, has to be obtained from models.

Finally, we illustrate how the constructed measurements can be used to increase resilience of the FLOSS ecosystem to the knowledge loss by assigning observers or maintainers to the strategically selected projects or source code files.

The remainder of this chapter is organized as follows: In section 2 the definition of FLOSS supply chains and general approaches used to optimize FLOSS supply chains' network are provided. In section 3 the

process of collecting and storing data from software projects hosted on various open source platforms is described. Section 4 composed the data extraction process, storage and cleaning through disambiguating author identities. Section 5 depicts operationalization of software supply chain by constructing code reuse, knowledge flow and dependency networks. In section 6 we provide a redundancy based approach to have more maintainers responsible for in-danger-files to reduce the the knowledge loss in the FLOSS ecosystem.

2 Supply Chains in FLOSS

The key output of software development activity is the source code. Therefore software development is reflected in the way the source code is created and modified. Although various individual and groups of projects have been well studied, it only gives partial results and conclusions on the propagation and reuse of source code in the large. As in traditional supply chains, the developers in FLOSS make individual decisions with some cooperative action, hence the analytical findings from traditional supply chains may help in FLOSS. Second, we have a complicated network of technical dependencies with code and and knowledge flows akin to traditional supply chains, making the analogies less complicated. Third, the emerging phenomena, for example, the lack of transparency and visibility, appear to be as, or more, important in FLOSS as in traditional supply chains. Fourth, unlike traditional supply chains, FLOSS has very detailed information about the production and dependencies. We, therefore, hope that detailed data with supply chain analytical framework may bring transformative insights not just for FLOSS supply chains, but for all supply chains generally. We, therefore, would like to systematically analyze the entire network among all the repositories on all source forges, revealing upstream to downstream relations, the flow of code and the flow of knowledge within and among projects.

2.1 Defining FLOSS Supply Network

La Londe *et. al.* proposed a supply chain as a set of firms that pass materials forward [20], Lambert *et. al.* define a supply chain as the alignment of firms that brings products or services to market [21], Christopher [8] described supply chain as the network of organizations that are involved, through upstream and downstream linkages, in the different processes and activities that produce value in the form of products and services delivered to the ultimate consumer. A common comprehension is that the supply chain is a set of three or more companies directly linked by one or more of the upstream or downstream flows of products, services, finance, and information from a source to a customer.

As software product developers increasingly depend on external suppliers, supply chains emerge, as in other industries. Upstream suppliers provide assets downstream to as more complex products emerge. As open source software proliferates, developers of new software tend to build on top of mature projects or packages with only a small amount of modifications, which leads to the emergence of software supply chain in OSS.

A supply chain with individual developers and groups (software projects or packages) representing “companies” producing new versions of the source code (e.g., files, modules, frameworks, or entire distributions). The upstream and downstream flow from projects to end users is represented by the dependencies and sharing of the source code and by the contributions via patches, issues, and exchange of information. This is our definition of software supply chain. Supply chains lead to two important concepts.

Visibility is information that developers have about the inputs, processes, sources and practices used to bring the product to consumers/market. This includes complete supply chain visibility including traceability for the entire supply chain. Visibility is, generally, inwardly/developer focused. Visibility refers to how far you can see upward beyond direct upstream, i.e. how many layers of dependency you can see from a software in supply chain.

Transparency is information that developers share with their consumers about the inputs, processes, sources and practices used to bring the product to the consumer. It is more outwardly focused/from the consumer perspective than visibility. How much each the developer or project is providing publicly (including the ability to interpret that information by others) is a form of transparency.

2.2 Notation Used for FLOSS Supply Network

In traditional supply chain, producers are considered as nodes of a graph and the flow of information or materials as links. Based on the definition of software supply chain and the ability to measure it, we use the following notation for key concepts of software supply chain throughout the chapter:

1. A Node is a
   - Developer - an individual producer, will be denoted as d. Developer author commits c with each commit having a single author $d = A(c)$.
   - A version of a file - a component work/information inserted into a project, will be denoted as $f_v \in p$. File versions are produced by commits with each commit producing zero or more file versions.
   - Project - a group of commits (a composition of work by individual developers) in the same repository. Will be denoted as $p = \{c : c \in p\}$. Since each commit produces a set of file versions, a project is also associated with all these file versions: $p = \{f_v : f_v \in c, c \in p\}$ and all authors of the commits.
2. There are different types of links
   - A technical dependence (upstream/downstream project $l_d(p_1, p_2)$)
   - Code flow file, *i.e.* file that has been copied in the past but is now being maintained in parallel $l_c(f, f1)$: $\exists f1, v_i, v_j$ *such that* $f1_{v_j} = f_{v_i}$.
   - Authorship: $l_a(d, f)$: $\exists c$ *such that* $d = A(c) \ \wedge f_v \in c$

3 Computing Infrastructure for Measuring FLOSS Supply Chains

FLOSS projects are not only scattered around the world, they also tend to be scattered around the web, hence, in order to collect data for measurement we need to discover where the relevant data sources are located [27, 26]. Historically, a variety of version control and issue tracking tools were used, but many of the projects can be now found on a few large platforms like GitHub, BitBucket, SourceForge, and GitLab and most projects have converged to Git as their version control system.

3.1 Discovery

While many projects have moved to (or at least are mirrored on) the main forges such as GitHub, a sizable number of projects are hosted on other forges. The number of such forges is not small. Some of these do not have stable APIs, and the rest each requires an unique API to discover all public projects on that forge. This makes the task of gathering information from these forges fairly challenging. However, although collecting information from these sources require slightly different approaches (which makes it difficult to use one single script for mining), the task itself isn't complicated and the only result required is the list of git URLs that could be used to mirror the data as described below. This makes the task an excellent candidate for crowdsourcing [27, 24]. Table 1 (from [24]) lists the active forges and an estimate of how many projects are hosted in each of them at the time of the study.

Table 1 Active Forges (other than GitHub and BitBucket) with Public Repositories [24]

Forge Name	**Forge URL**	**API**	**Repositories Retrieved**
CloudForge	cloudforge.com	Private API	42
SourceForge	sourceforge.net	REST API	48,000 - 50,000
launchpad	launchpad.net	API	36,860
Assembla	assembla.com/home	No	about 70,000
CodePlex	codeplex.com	REST API	100,000
Savannah	savannah.gnu.org	No	3613
CCPForge	ccpforge.cse.rl.ac.uk/gf	No	126
Jenkins	ci.jenkins-ci.org	REST API	106,336
Repository Hosting	Respositoryhosting.com	No	$<$88
KForge	pythonhosted.org/kforge	API	81,000
Phabricator	phabricator.org	Conduit API	about 10,000
Fedorahosted	fedorahosted.org/web	No	914
JavaForge	javaforge.com	No	7672
Kiln	fogcreek.com/kiln	No	43
SVNRepository	SVNRepository.com	No	15
Pikacode	pikacode.com	No	2
Planio	plan.io	No	26
GNA!	gna.org	No	1326
JoomlaCode	joomlacode.org/gf	REST API	971
tuxfamily	tuxfamily.org	No	209
pastebin	pastebin.com	No	about 1800
GitLab	gitlab.com	No	about 57,000
Eclipse	eclipse.org/home/index.php	No	214
Turnkey GNU	turnkeylinux.org/all	No	100
JavaNet	home.java.net/projects/alpha	No	1583
Stash	atlassian.com/software/bitbucket/server	REST API	5400
Transifex	transifex.com	No	5400
Tigris	tigris.org	No	678

Apart from discovering open source projects from forges that host VCS, software projects information can also be found in the metadata of popular Linux distributions. In particular [27], Gentoo, Debian, Slackware, OpenSuse, and RedHat distributions and package repositories such as rpmforge, provided a list of popular packages. Moreover, there are directories of open source projects that list home page URLs and other information about the projects. RawMeat (no longer in operation) and Free Software Foundation were

two prominent examples of such directories. While they do not host VCSs or even provide URLs to a VCS, they do provide pointers to source code snapshots in tarballs and project home pages.

3.2 Retrieval, Extraction, and Schema for Analytics

Source code changes in software projects are recorded in a VCS (version control system). Git is presently the most common version control system, sometimes with historic data imported from SVN or other VCS used in the past. Code changes are typically organized into commits that make changes to one or more source code files. Git repositories hosted on open source platforms can be retrieved by cloning them (functionality provided by `git clone --mirror`) to local servers.

The retrieved git database stores the full history of changes/commits made to a project. A Git commit records author, commit time, a pointer to the projects' file system, a pointer to the parent change, and the text of the commit message. Internally, the Git database has three primary types of objects: commits, trees, and blobs. Each object is represented by its sha1 value that can be used as a key to find its content. The content of a blob object is a content of a specific version of a file. The content of a tree object is a folder in a file system represented by the list of sha1s for the blobs and the trees (subfolders) contained in it. A commit contains sha1 for the corresponding tree, a list of parent commit sha1s, an author id, a committer id, a commit timestamp, and the commit message.

We extract git objects from each project and store them in the common database. This reduces the amount of storage needed approximately 100 times (which is an average number of projects a git object belongs to), and allows us to conduct analysis of the relationships. We have 2.8B blobs, 3.1B trees, and 0.8B commits collected from 40 million projects.

Git is not a system that stores data in a way that makes analysis easy. We, therefore, re-organize and re-structure it in an efficient way to facilitate various analytics related to the above described concepts of software supply chain. The data must be stored in a way that allows fast and efficient data lookup for billions of objects. An appropriate structure for that is a hashtable or a key-value database optimized to retrieve fast by exact value of a key. For example, a developer as the key and the list of commits authored by the developer as value. This allows a fast response when a requesting what commits one specific developer made. Another example is the link between a commit and files modified by the commit. This is accomplished by comparing the tree (and all subtrees) of the commit with the tree of the parent commit. The new blobs created indicate new f_vs. Since the complete tree and subtrees can be fairly large, the operation is computationally non-trivial and, because such relationships are commonly needed, is worth precomputing.

We compared the performance of several key-value databases and found that TokyoCabinet to be the most competitive one in terms of tradeoffs between speed and storage needs. We break the keys by part of their sha1 into up to 128 different databases to facilitate parallel (hadoop-like) processing when we need to iterate over the entire database and to reduce the size of each individual database. These key-value maps are constructed to map developers to authored commits and files, commits to projects (and back), commits to their children commits, blobs created (and back to commit), and other lookup tables needed to construct the software supply chain.

The overall diagram of the data workflow is shown in Figure 1. The calendar time goes down, while the data layers from raw to analytics go from left to right.

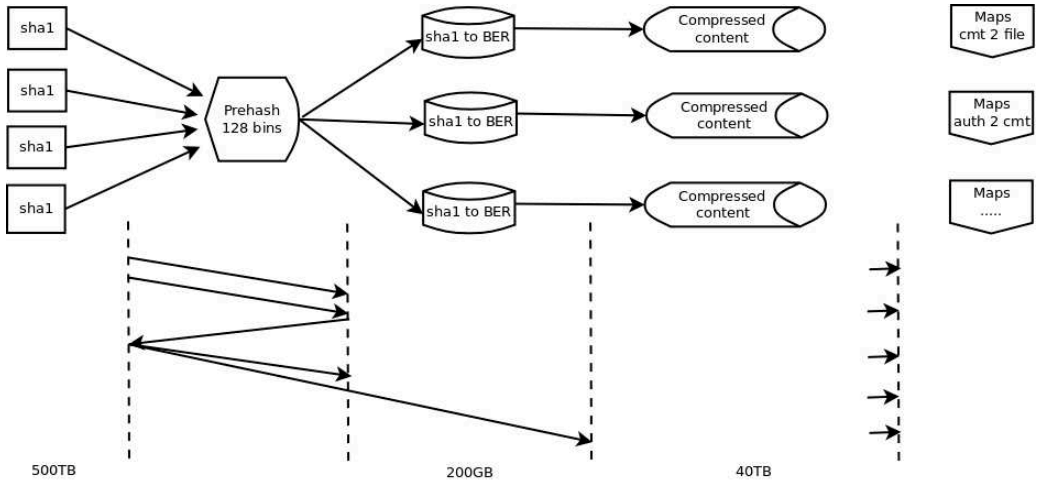

Fig. 1 Data retrieval diagram

4 Correction and Augmentation

Operational data extracted from software repositories [30] often contain incorrect and missing values. For example, and most importantly, primary author id, a key field for many analyses, often suffers from errors such as multiple or erroneous spellings, identity changes that occur over time, group identities, and other issues. These problems arise because primary author information in a Git commit (which we study here) depends on an entry specifying user name and email in a Git configuration file for the specific computer a developer is using at the moment. Once the Git commit is recorded, it is immutable like other Git objects, and can not 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 and email. It is also not uncommon to see the commits done under an organizational alias, thus obscuring the identity of the author.

Since developers serve as nodes in the supply chain network, it is of paramount importance to determine developer identities accurately. Erroneous data in developer identifiers can result in a misrepresented network undermining the value of constructing an OSS supply chain network. These issues have been recognized in software engineering [12, 2] and beyond [9]. However, identity resolution to identify actual developers based on data from software repositories is non-trivial mainly due to

1. Lack of ground truth - absence of validated maps from the recorded to actual identities. Similar disambiguation approaches have been applied on census data [9] or patent data [41] whereby over 150,000 samples of ground truth data was available.
2. Data Volume - millions of developer identities in hundreds of millions of code commits.

To avoid these challenges, studies in the software engineering field tend to focus on individual projects or groups of projects where the number of IDs that need to be disambiguated is small enough for manual validation. 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 [40, 2] to supervised learning based on string similarity [41] have been used to solve the identity problem in the past. However, such methods do not resolve all issues that are particular to data generated by version control systems. Therefore, in order to propose solutions or to tailor existing identity resolution approaches, we need a better understanding of the nature of the errors associated with the records related to developer identity.

4.1 Problems with the data

We inspected the collection of more than nine million author strings collected from over 500M Git commits and looked at random subsets of author IDs to understand how or why these errors occur. We identified these errors and broadly categorized them into the following three kinds - synonyms, homonyms and missing data and determined the common reasons causing errors to be injected into the system.

1. Synonyms: These kinds of errors are introduced when a person uses different strings for names, usernames or email addresses. For example, 'utsav dusad <utsavdusad@gmail.com>' and 'utsavdusad <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'. Developers may change their name over time, for example, after marriage, creating a synonym.
2. Homonyms: Homonym errors are introduced when multiple people use the same organizational email address. For example, the Id 'saper <saper@saper.info>' may be used by multiple entities in the organization. For example 'Marcin Cieslak <saper@saper.info>' is an entity who may have committed under the above organizational alias.
   Template credentials from tools is another source that might introduce homonym errors in the data as some users may not enter values for name and/or an email field. For example, 'Your Name <vponomaryov@mirantis.com>' which may belong to author 'vponomaryov <vponomaryov@mirantis.com>'. 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 devnull@localhost
3. Missing Data: Errors are also introduced when a user leaves the name or email field empty, for example, 'chrisw <unknown>'.

A look at the most common, names and user names shows that many of them were unlikely to be names of individuals. For example, the most frequent names in the dataset such as 'nobody', 'root', and 'Administrator' are a result of homonym errors as shown in Table 2.

4.2 Disambiguation Approach

Traditional record linkage methodology and identity linking in software [2] splits identity strings into several parts. Our approach splits the information in the author string into several fields representing the structure of that string and defines similarity metrics for all author pairs. We also incorporate the term frequency measure for each of the attributes in a pair. Finally, we add similarity between behavioral fingerprints for all pairs of authors in the dataset.

Table 2 Data Overview: The 10 most frequent names and emails

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

1. **Author Distances Based on String Similarity:** Each author string is stored in the following format - "name <email>", e.g. "Hong Hui Xiao <xiaohhui@cn.ibm.com>". For our analysis, we define the following attributes for each user.

   a. Author: String as extracted from source as shown in the example above
   b. Name: String up to the space before the first '<'
   c. Email: String within the '<>' brackets
   d. First name: String up to the first space, '+', '-', '_', ',', '.' and camel case encountered in the name field
   e. Last name: String after the last space, '+', '-', '_', ',', '.' and camel case encountered in the name field
   f. User name: String up to the '@' character in the email field

   Additionally, we introduce a field 'inverse first name' whereby the last name of the author is assigned to this attribute. 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 first, last and the name field.
   In order to measure the distance between strings, we tested two common measures of string similarity, the Levenshtein score and the Jaro-Winkler score [43]. Our experiments indicated that the Jaro-Winkler similarity produces scores that are more reflective of actual similarity as verified by human experts than the Levenshtein score. Therefore, we implemented the Jaro-Winkler score as the measure of similarity throughout the rest of this study.
   The Jaro Similarity 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 i, m is the number of matching characters and t is half the number of transpositions.
   The Jaro-Winkler Similarity modifies the Jaro similarity so that differences at the beginning of the string have 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.

2. **Author Distance Based on String Frequency:** We count the number of occurrences of the attributes for each author as defined in Section 1 i.e. name, first name, last name, user name and email for our dataset. We calculate the similarity between author pairs, authors a_1 and a_2, for each of these attributes as follows:

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

We generate a list of 200 common strings of names, first names, last names and user names and emails from the larger dataset of 9.4M authors (the first 10 shown in Table 2) and manually remove names that appear to be legitimate, i.e. Lee, Chen, Chris, Daniel etc. We set string frequency similarity of a pair of name or first name or last name or user name to -10 if at least one element of the pair belongs a string identified as not legitimate. 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 found that the value for other highly frequent terms were significantly greater than -10.

3. **Author Distances Based on Fingerprints** There are 4 additional distance measures we incorporate into our study which address the behavioral attributes of authors: 1) Author similarity based on files touched, when two authors identities have modified the same files there is a greater chance that they represent the same entity. 2) Author similarity based on time zone, two author identities committing in the same time zone indicate geographic proximity and, therefore, a higher similarity weight is given. 3) Author similarity based on text, similarity in style of text between two author identities may indicate that they are the same physical entity. 4) Gender, incorporating gender information helps us distinguish between highly similar author identity strings. Quantitative operationalizations are given below.

   a. **Author similarity based on files touched**: Each file is weighted using the number of authors that have modified it. The file weight is defined as the inverse of the number of distinct authors who have modified that file. The pairwise similarity between authors, a_1 and a_2, is derived by summing over the weights of the files touched by both authors. A similar metric was found to work well finding instances of succession (when one developer takes over the work of another developer) [28]. In this metric, we consider only the first 100 common authors for a given file.

$$file\_weight(W_f) = \frac{1}{A_f}, \text{where } A = |a_1, ..., a_n|$$

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

   b. **Author similarity based on time zone**: We discovered 300 distinct time zones strings from the commits and created a 'author by time zone' matrix that had the count of commits by an author in a given time-zone. All time zones that had less than 2 entries were eliminated from further study. Each author is therefore assigned a normalized time-zone vector (with 139 time zones) that represents the pattern of his 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 author i's time-zone vector as:

$$a_i = C_{a_i} . \frac{1}{A_T},$$

Here, C_{a_i} is the vector representing the commits of an author *i* in the different time zones and A_T is the vector representing the number of authors in the different time zones. The pairwise similarity metric between author a_1 and author a_2 is calculated as:

$$tzd_{a_1a_2} = cos\_sim(a_1, a_2)$$

where a_1 and a_2 are the authors' respective vectors.

c. **Text similarity**: We use the Gensim's [1] implementation of the Doc2Vec [22] algorithm to generate vectors that embed the semantics and style of the commits messages of each author. All commit messages for each individual who contributed at least once to one of the OpenStack projects were gathered from the collection described above and a Doc2Vec model was built. We obtained a 200 dimensional vector for each of the 16,007 authors in our dataset and calculated cosine similarity to find pairwise similarity between authors.

$$d2v_{a_1a_2} = cos\_sim(a_1, a_2)$$

d. **Gender Similarity**: We obtain the gender of the users as either Male, Female or Undetermined. The similarity between author pairs are determined as follows:

$$gs_{a_1a_2} = \begin{cases} 0.5, & \text{if } G_{a_1} \text{ or } G_{a_2} = \text{Undetermined} \\ 1, & \text{if } G_{a_1} = G_{a_2} \\ 0, & \text{if } G_{a_1} \neq G_{a_2} \end{cases}$$

where G_i represents the gender of author *i*.

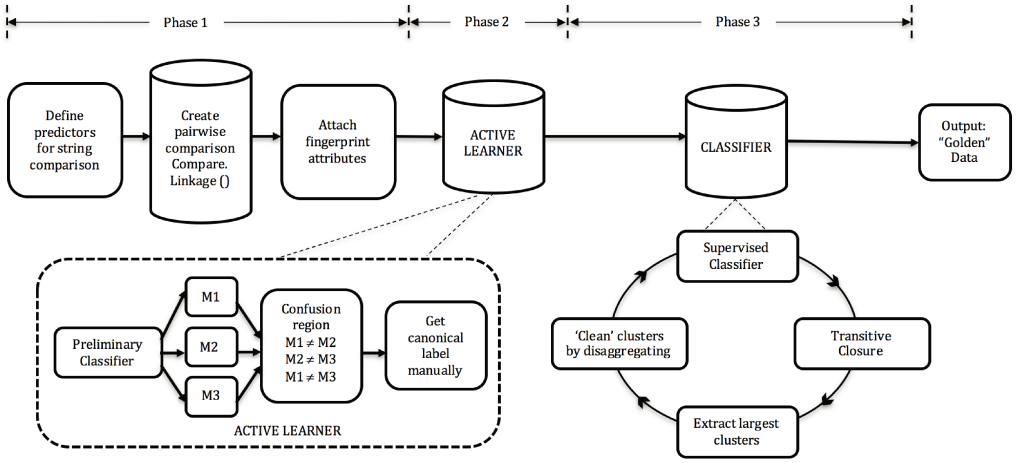

Fig. 2 Concept of the Disambiguation Process

4. **Data Correction** The data correction process can be divided into 3 broad phases as shown in Figure 2.

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

a. **Define predictors** - In this phase we compute the string similarity, frequency similarity and behavioral similarity. We use functions from the RecordLinkage library [38] to compute Jaro-Winkler similarities of the defined attributes (name, first name, last name, email, username). We compute string similarity between a pair of authors' name, first name, last name, user name, email and the first author's first name to the 2nd author's last name (we refer to this as the inverse first name). Based on our preliminary analysis we found many instances of developers using their names in both orders. In addition to the string similarities based on these fields, we also include the term frequency metric, as is commonly done in record matching literature. The high frequency values tend to carry less discriminative power than infrequent email addresses or names. Finally, we include three fingerprint metrics - author similarity based on files touched, time-zone similarity and commit log text similarity. This resulting matrix data is used as an input to the next phase, the active learning process.
b. Active learning - This phase uses a preliminary classifier to extract a small set from large collection of data and generate labels for further classification. Supervised classification requires ground truth data. As noted earlier, it is extremely time consuming and error-prone to produce a large set of manually classified data to serve as an input for a supervised classifier. Moreover, identifying a small subset of instances so that the classifier would produce accurate results on the remainder of the data is also challenging. A concept called *Active Learning* [37] using a preliminary classifier helps us extract a small set of author pairs that is viable for manual labeling, from the set of over 256M author pairs. To design the preliminary classifier, we partition the data into ten parts and fit bootstrap aggregation (bagging) models on three different combinations of nine parts and predict on one the ten parts. Each classifier learns from manually classified pairs and outputs links or non-links for each author pair in the prediction set. The three classifiers trained on different training subsets yield slightly different predictions (links and no-links for each pair). The mismatch between predictions of two such classifiers indicates instances where the classifier has large uncertainty (confusion regions). We conducted a probabilistic manual classification on the cases in the confusion region of the classifier and extracted pairs where links were assigned with full confidence i.e. probability = 1. Each pair was updated manually to include a canonical label chosen from among the existing author identities that had a proper name and email address. This produces a preliminary set of training data for supervised classification.
c. **Classification** - In this phase we discuss supervised classification suitable for disambiguation, transitive closure applied on classifier output, extraction of clusters to correct, and dis-aggregation of wrongly clustered individuals. Once the labeled dataset is created, we use it to train random forest models which are commonly used in record matching literature. A 10-fold cross validation using this method produced high precision and recall scores for the classifier. The final predictor involves a transitive closure on the pairwise links obtained from the classifier[2]. The result of the transitive closure is a set of connected components with each cluster representing a single physical entity. 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 if names/emails were not sufficient to assign them to a cluster with adequate confidence.

[2] 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

4.3 Handling Missing Data

In addition to the bad and/or incorrect data, the observational data collected for the different software ecosystems often do not have observations for all the relevant variables [30, 44]. Generally, the missing data problem focuses on cases where a few observation values are missing for an otherwise observed variable [23], however, when talking about missing data in this context, we have to take into consideration cases where a number of variables might be completely unobserved as well. For example, if we are trying to measure the popularity of a particular project in an ecosystem, the best possible measure would be the number of active users. However, the number of active users is a quantity very hard to measure in practice, and the second best measure, the number of downloads, is typically not tracked very accurately for most FLOSS software. At this point, our choices are to either find a proxy measure for the popularity of a project or find a way to estimate the unobserved variables.

As for the proxy measures, there a few options, e.g. the number of stars/watchers/forks for a GitHub project [34, 16, 42], however, although these measures should closely correlate with the actual popularity of the product, sometimes analyses done using these measures could end up finding some relationship that is an artifact related to that particular measure, and is not reflective of the actual popularity. Because these metrics are easily manipulated, they may also be deliberately biased and not representative.

A more appealing option, therefore, is to estimate the missing observations. In the more common case of missing data estimation, only a few observations are missing for a variable, and the estimation can be done by means of partial/full imputation and/or interpolation or extrapolation [23]. However, when a variable is completely unobserved for a dataset, such techniques can not be used. In such a scenario, a set of alternative methods are useful, as listed below:

- **Factor analysis** [11, 13, 25]: If we have measures for a set of variables that are likely to be affected by a common set of unobserved variables, we can perform a method called factor analysis on the observed variables to extract an estimate for the missing unobserved "factors". This method, however, depends on both a parametric probability model and assumes a particular relationship between the unobserved variables and the multivariate observation.
  With regard to the example of measuring the popularity (*i.e.* number of users) of a project, if we have measurements for a set of variables (hypothetically) directly affected by the number of users (*e.g.* number of crashes, downloads, or even forks or stars for a GitHub project), we can extract the maximum likelihood factors from those variables (*e.g.* by using the `factanal` function in R[3]), which, under the assumption that each observation is the sum of a linear combination of the underlying missing factors and a gaussian noise component, should give an accurate estimate of the number of users.
- **Prediction:** If the scenario is such that the values of a variable are available only in certain situations, a predictive model can be used for estimating the unobserved variable. For example, the number of users for a particular software might be available only for a specific subset of releases. In this case, we may use the complete observations for releases where the data is observed to train a model (*e.g.* linear regression model or Random Forest) that can be used to predict the number users in cases where this quantity is not observed.
- **Hidden node detection using graphical models:** If a graphical model is used for modeling the interrelationship among the variables, an unobserved variable might be represented by a hidden node in the graph and can be estimated using data from the variables that have connections to the hidden node [15, 17, 33]. Factor analysis may be viewed as a special case of this type of analysis.

[3] https://www.statmethods.net/advstats/factor.html

In order to measure the number of users for a software in this method, we first need to construct a graphical model of dependence among all of the observed variables. Two strategies are usually used to define the structure: 1) the graph represents dependencies obtained from domain experts, or 2) the graph may initially be based on prior distributions about the parameters of the overall model. The data is then used to calculate the posterior distribution and to make inference. The second approach makes minimal a-priori assumptions about the model and focuses on the search for the best graphical representation for a given dataset (structure learning). This is an NP-hard problem [7], but a number of different heuristic structure learning algorithms are available [39].
After the model is constructed, one or more hidden nodes can be added to it. The standard approach is adding one node at a time and optimizing its placement by optimizing the network score (generally BIC score in such situations) at each step [4, 14].
Graphical models models have several advantages over regression models. To be precise, regression analysis is a very simple graphical model allowing one directed link from each independent variable to dependent variable. Therefore, the more general approach of graphical models can help with multicollinearity (which is a common problem in the software due to many of of the observed variables being highly correlated) by linking independent variables.

5 Code and Knowledge Flow and Technical Dependencies

The most fundamental part of software supply chain or ecosystem is the networks of dependencies and code or knowledge flows. **The dependency network** is based on technical dependencies. These can be subdivided into several types. For example, a run-time dependency requires a library from another package to be available when the program is run. Package dependencies in Debian are an example of such relationship. A different type of dependency is build dependency, where a set of tools and include files may be needed in order to compile and build a package. Optional dependencies usually denote the potential extension in the functionality of a program if that dependency is satisfied. **The code flow network** represents the source code copying. **The knowledge flow network** represents implicit exchange of information as developers modify source code in sequence. A senior developer d_s creates (or modifies) a set of source code files. Another developer d_j modifies a subset of these files, thus having to understand design decisions made by d_s. This mentor-follower knowledge flow can be quantified [28].

5.1 Constructing technical dependencies

As discussed above, different types of technical dependencies exist. Major types are dependencies required to run software and dependencies required to build software. Each dependency may need to be obtained differently for projects that are inside package managers such as deb or npm (and, thus, have metadata in the package manager that explicitly specifies the dependencies) and projects outside package managers, where dependencies can only be extracted based on the actual content of the code, configuration, and build scripts.

Dependencies within a specific package manager are recorded when a new package is added into package manager or its dependencies change. For example, the dependency information for packages hosted on NPM can be extracted from `PACKGE.json` file and is also stored in the NPM registry.

Different package managers may have different standards of defining dependencies, e.g. NPM has five types of dependencies: dependencies, devDependencies, peerDependencies, bundledDependencies, and optionalDependencies; while packages in R CRAN also have five (but not equivalent) types: depends, imports, suggests, linkingto, and enhances. Defining standards for the categorization of dependencies that are generally applicable to all package managers may not be possible.

We illustrate the procedures of constructing the dependency network by exploring R CRAN ecosystem. R package can be scraped from R CRAN official website which contains approximately 11K packages. We used data from METACRAN[4] which provides the latest R CRAN metadata containing the dependency information. As we have mentioned in section 4.2.1, there are five types of dependency keywords in R CRAN and we considered 'imports' and 'depends' as dependency, because packages listed in 'imports' must be installed in advance and 'depends'[5] is the old name for 'imports'.

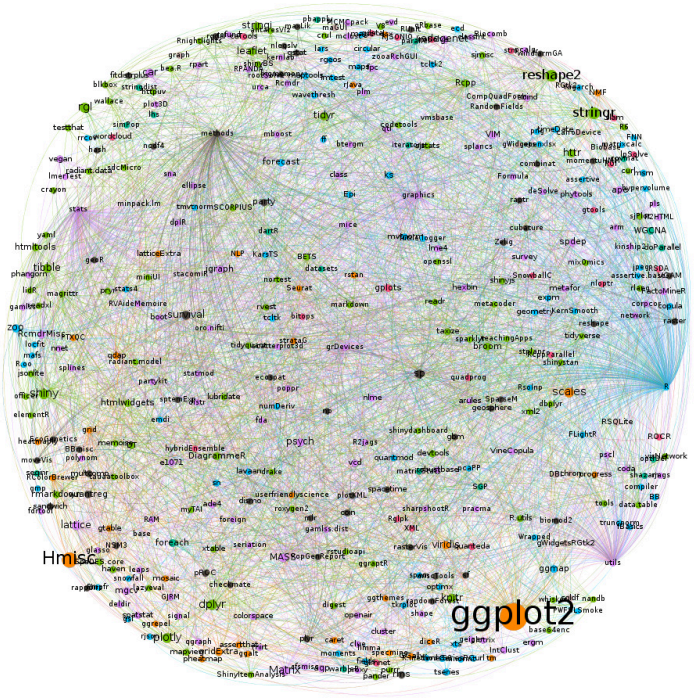

Fig. 3 R CRAN dependency network.

By creating a link from individual package to each dependency in its 'imports' and 'depends', we construct a dependency network for R CRAN in Figure 3. Packages with degree less than 20 are removed which ends up with 421 (1.9%) nodes and 3235 (6.6%) edges in Figure 3. Node size is proportional to its betweenness centrality value, and the color is based on modularization algorithm[6] of gephi. In Figure 3, numerous dependency links are revealed among popular R CRAN packages. In particular, 'ggplot2', 'Hmisc', 'reshape2', 'stringr' and 'Rcpp' are core packages based on betweenness centrality.

Unfortunately, projects that are not a part of the registries of package managers may have no metadata that allows easy identification of dependencies. Since such projects represent a bulk of projects, the dependencies need to be extracted directly from the source/configuration/build code. For example, import statements

[4] METACRAN is a collection of services around the CRAN repository of R packages. https://www.r-pkg.org/about

[5] Prior to the rollout of namespaces in R 2.14.0, Depends was the only way to 'depend" on another package. Now, despite the name, you should almost always use Imports, not Depends.

[6] https://github.com/gephi/gephi/wiki/Modularity

in Java or Python, use statements in Perl, include statement in C, or, as is the case for our study, library statements for the R language.

Below is an example workflow to determine dependencies for all R files in all projects:

1. Identify all R-language files by extension (.r or .R) in the complete list of all files in the file-to-commit map described above.

2. For each filename use filename to blob (file versions) map to obtain the content for all versions of the R-language files obtained in Step 1.

3. Analyze the resulting set of blobs to find a statement indicating an install or a use of a package:

- `install\.packages\(.*"PACKAGE".*\)`
- `library\(.*[\"']*?PACKAGE[\"']*?.*\)`
- `require\(.*[\"']*?PACKAGE[\"']*?.*\)`

4. Use blob to commit map to obtain all commits that produced these blobs and then use the commit to determine the date that the blob was created.

5. Use commit to project map to gather all projects that installed the relevant set of packages.

A similar approach can be applied to other languages and technologies with suitable modification in the dependency extraction procedures, since different package managers, different languages, or different frameworks might require alternative approaches to identify dependencies or the instances of use. Dependencies can typically be detected in a programming language or build system dependent manner [36]. For example, the dependency information of a python source file is listed in `import` statement; dependency information of a C project is listed in header files; package dependency in Debian can be extracted by *apt-cache depends package-name*

5.2 Constructing code flow networks

In FLOSS the code sharing is possible and welcome, unlike in proprietary software and is, perhaps, one of the key advantages that brings rapid innovation with new projects building from components or copied code of existing projects.

Code flow has been extensively investigated, albeit at a smaller scale. To determine instances of code flow several approaches may be taken:

- Compare the strings representing the content of a source code file in the potential source and the potential destination [18, 19, 1]
- Compare the strings representing the file name and the path [5, 6, 46]

Here we illustrate the first approach as it is largely language independent and allows detection for code and non-code flow. When two files have a matching content, i.e., $\exists v_1, v_2 : f^1_{v_1} = f^2_{v_2}$ and f^1 and f^2 are files from distinct projects, it is not unreasonable to assume that $f^1_{v_1}$ and $f^2_{v_2}$ were not created independently but the code was copied. This apply if the unit of code is not an entire file, but only a part of file. From the theoretical perspective we may produce false links (links where code flow does not exist, i.e., the content of both files was created independently of each other) and also miss links where information does flow, as in cases where the copied code was modified substantially before being committed to the repository.

We, therefore, need to quantify and minimize both of these potential errors. Whether we look at the file content or file pathname, the erroneous links may be introduced if the two linked strings are similar (or the same) purely by chance and the information was never shared. If we assume the string to be a random

sequence of characters, the chance that two strings of length n would match purely by chance is m^{-n} where m is the size of alphabet. We can easily eliminate false matches (make the chance of such matches negligibly small) by ensuring that the string is of nontrivial length. For example, a random string with ten characters (from alphabet of 26 letters) would match by chance with probability lower than 10^{-14}. By considering links that are based on strings exceeding such length we can ensure a very low probability of false matches.

Unfortunately the strings representing file content and file pathnames are not random for a variety of reasons [6, 46, 26]:

- file depth in a project is not random distributed (usually file depth varies between 2 to 5)
- filenames are not always related to file content, e.g. foo
- some filenames are quite common among projects, e.g. main.c
- the content may be generated by a tool, therefore anyone using a tool will have exactly the same content.
- the template may have been used and only small parts of the template have been modified.

We, therefore, have to add additional ways of eliminating false links from the supply chain network through other means. For example, by identifying the reasons for false positives and removing links that are similar to the identified reasons for false positives.

Once the presence of the link is established, the next question involves the direction of the code flow. *File creation time* may serve such purpose. For Case 1, if the creation time of file f_i precedes that of f_j the direction of flow should, in general, go from p_1 to p_2. For Case 2, if the matching version of file $f_i(v_i)$ was created prior to $f_j(v_j)$ the direction of flow should go from p_1 to p_2.

The rationale for such approach would be that if a file F is first created in project A and then copied to project B, the creation time of file F in project A is prior than that in project B, the project B is likely to be downstream to project A because file F was supplied to project B from Project A. It is possible that in some cases the primary maintenance of file F may be transferred to Project B and Project A gets updates of file F from project B, but such instances could be detected by a more in-depth analysis of version history of file F in both projects [5].

A detailed procedure to illustrate the constructing code flow network is discussed next.

5.2.1 Code flow network for ember.js

Front side web framework ember.js has been attracting many contributors over several years, which makes it suitable to illustrate how complicated code flow network may be.

To create the code flow network we first collect all file names f and file versions f_v in the form of their SHA1 digests from emberjs/ember.js project (E). We use project-To-filename map to obtain files $A = f \in E$ and project-To-blob map to obtain the file versions $B = f_v \in E$. For each file in A we then use filename-To-project map to find all other the projects that contains this file name. Similarly, for each $f_v \in B$ we use blob-To-project map to find all other projects that contains this blob. This procedure creates links from E to all projects that share a filename or a blob.

These initial links contain numerous false positives and need be filtered. Links created by file names that start with a period are often created by IDE tools or programming language/script, so they should be removed as they do represent code transfer from one project to another. It represents not code flow, but dependence on the tool.

Links that are created by forked projects of Emberjs/ember.js also need to be removed because they are a part of Emberjs/ember.js project. GitHub forks are created primarily to be able to contribute to the the main

project via pull requests, not to start a new project. Again, these projects represent private branches and do represent code flow but at a finer granularity than we consider at the moment.

In addition to the traditional definition of a forked project where the development is done in parallel with no intention to merge projects, the repository of the code for Ubuntu/Debian distribution does represent an example of downstream development done in order to maintain compatibility among the projects collected into a single distribution.

Once false positives are eliminated, 54 projects have code flow to pr from ember.js. To understand the patterns of code flow we categorized these projects into different groups.

- Build tools: rake — makefile for Ruby on Rails.
- Testing: qunit — a testing framework.
- Runtime: jQuery — a JavaScript library.
- Framework: epf — emberjs Persistence Foundation.
- Prior incarnations: SproutCore/Amber.js — early name for the emberjs project.
- Hard forks: innoarch/bricks.ui — a hard fork of emberjs that was then developed as a separate project.
- Tutorials: cookbooks/nodjs — early code examples.
- Package manager: package.json — a file for NPM package manager.

As we can see, these types of code flow have different causes: tools, libraries and frameworks, hard forks, documentation templates, and distribution templates. Each type of code flow appears, therefore, to represent a different phenomena and needs to be identified and investigated separately.

5.3 Constructing knowledge flow networks

Developer knowledge varies from developer to developer and depends on what they have worked on [31, 10]. A unit of work can be considered as experience atom [31] and approximated by developers' modifications to the source code. Each time a developer makes a change to a file, they have first to understand the design decisions that went to the code they modify and, at the same time, their modification (be it a code fix or additional functionality) implements their knowledge in the way the change is designed and implemented. Thus, the knowledge of earlier developers, through code, flows to developers who modify the code later. This observation can facilitate linking of developers trough the timing of the changes they make on files modified in common. Using notation introduced above, let d_s, d_j denote two developers. Let F_{d_s,d_j} be a set of files modified by both developers. Let S_2 be the strength of expertise transferred. Let N_{fd_s} be the number of changes developer d_s made to file f (changes are made and counted through commits). Let $FC(f,d_s)$ denote the time when developer d_s made his/her first change to file f. Then the challenge of measuring the strength of knowledge flow from senior developer to his/her subsequent developers can be approximated via the following expression [28]:

$$S_2(d_s,d_j) = \sum_{f:\begin{cases} f \in F_{d_s,d_j} \\ FC(f,d_j) > FC(f,d_s) \end{cases}} \frac{N_{fd_s} + N_{fd_j}}{\sum_i N_{fd_i}} \tag{1}$$

The formula can be interpreted as follows: the strength of expertise flow from developer d_s to developer d_j is based on the sum of their contribution ratio to files in which developer d_s's first change is earlier than developer d_j's. Files changed mostly by others where the two developers had contributed little would not

contribute much to the measure, but files where at least one developer made significant fraction of changes would contribute a lot.

For example, knowledge flow network in a popular web front side framework – emberjs is shown in Figure 4. The node size is proportional to its betweenness centrality value, and the color is based on modularization algorithm[7] of gephi. Note that several labels have been adjusted to fit the page size. The most productive developers are annotated via their name and email. More information on ember.js can be found in 5.2.1. In Figure 4, there are several big clusters of developers centered around each core developer. More specifically, 'Peter Wagenet' and 'Robert Jackson' are leading developers with vast number of successors. Clusters are linked by shared followers although the density of such links is low, indicating that majority of developers in ember.js tend to follow the work of a single core developer.

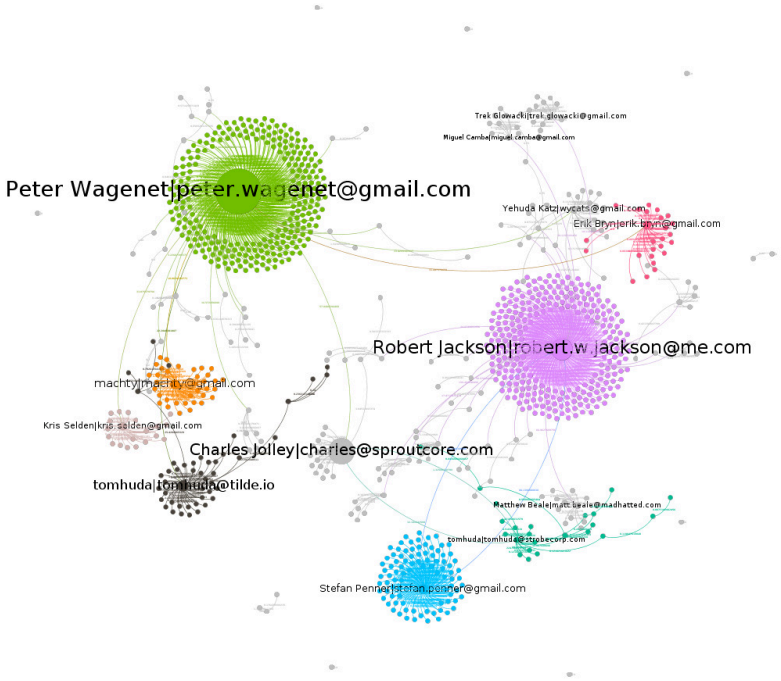

Fig. 4 Knowledge Flow Network for Emberjs

6 Example Application: Increasing Knowledge Redundancy

As developers who author source code become experts for that code, what happens if they, for some reason, stop maintaining the project? Fixing bugs or adding functionality to such code will become harder and fault prone [32, 3, 29, 28]. If an organization can identify the files that are likely to be left with no maintainer in the future, it may choose to assign their employees as additional maintainers to reduce the risk. It would seem that increasing the number (redundancy) of maintainers may reduce the risk. It may be possible to increase this knowledge redundancy by borrowing ideas from data redundancy. Erasure codes are forward error corrections codes used to prevent stored data from being lost work by increasing data redundancy. The

[7] https://github.com/gephi/gephi/wiki/Modularity

main idea of the erasure codes is that the data of size B is divided into k segments and then the k segments are further converted into n segments such that $n = k + m$, where m is the amount of redundancy added to the original data. As a result, up to m segment failures can be tolerated.

Adding redundancy to the existing knowledge for each file can help mitigate the risk of lost knowledge resulting from developer turnover [35, 45]. Let's denote files as f, assigned maintainers as m, and original developers as d. Lets assume that we have I files, J developers, and Z backup maintainers.

We represent developer d_j and file f_i relationship as a developer matrix

$$D_{d_j f_i} = \begin{Bmatrix} 1 \; if \; d_j \; maintains \; f_i \\ 0 \qquad otherwise \end{Bmatrix}$$

For illustration, we include an example below, where developer d_1 is responsible for file f_1 but is not responsible for file f_2. Then M is:

$$D = \begin{bmatrix} & f_1 & f_2 & f_3 & \cdots & f_I \\ d_1: & 1 & 0 & 1 & \cdots & 1 \\ d_2: & 0 & 1 & 1 & \cdots & 0 \\ & & & & \vdots & \\ d_J: & 0 & 0 & 1 & \cdots & 0 \end{bmatrix}$$

Then, the number of developers responsible for file f_i is the sum of the column corresponding to file f_i, which in the example above is:

$$R = \begin{bmatrix} f_1 & f_2 & f_3 & \cdots & f_I \\ 1 & 1 & 3 & \cdots & 1 \end{bmatrix}$$

We refer to this vector R as knowledge redundancy.

Once the risky files (with low knowledge redundancy) are identified, the next step in our risk-mitigation approach is to assign the files at risk to backup maintainers. Let's define a threshold t that represents the maximum number of files that each backup maintainer is capable of being responsible for. We also define r as the minimum amount of redundancy that can be tolerated. Similar to matrix M above, we construct matrix M' for the backup maintainers, in which each row is a vector representing the files that the corresponding backup maintainer is responsible for.

$$M' = \begin{bmatrix} & f_1 & f_2 & f_3 & \cdots & f_I \\ m_1: & 1 & 1 & 0 & \cdots & 1 \\ & & & & \vdots & \\ m_Z: & 0 & 1 & 1 & \cdots & 1 \end{bmatrix}$$

Files with fewer than r developers need one or more backup maintainers, but the sum of each row of M' can not exceed the maintainer capacity threshold t. We first calculate the number of file/maintainer slots that need to be assigned to the backup maintainers. That number is $slots = rI - \sum_{i=1\ldots I} \min(R_i, r)$. Obviously, we need at least $\max(r - \min_i r_i, slots/t)$ maintainers. To minimize the number of backup maintainers, we can always target the current maintainers to be responsible for some of the files that they are not currently in charge of, or count on volunteers.

The problem of optimally assigning files to backup maintainers can be cast as a mathematical integer program. Below is a possible formulation, that can be solved with readily available solvers such as *CPLEX* or *Gurobi*.

$$\text{maximize} \sum_{i=1}^{I} R_i$$
$$\text{subject to} \sum_{i=1}^{J} M[i][j] + \sum_{z=1}^{Z} M'[z][j] \geq r,\ j = 1,...,I$$
$$\textit{and}$$
$$\sum_{i=1}^{I} M'[z][i] \leq t,\ z = 1,...,Z$$

We can refine the objective to minimize the average or maximum risk resulting from discontinued contribution r developers together by adding conditions that are based on the structural properties of M. Using this approach it is possible increase the knowledge redundancy of each file to at least r (e.g. some file might already have more that r maintainers).

7 Conclusions

The ability to understand software ecosystems is limited by the ability to measure the relevant properties of these ecosystems and the conceptual framework needed to do the measurement. Many of the modeling or intervention techniques described in this book need a sound measurement framework. We have argued for the need to look at FLOSS from a global perspective and through the supply chain conceptual framework. We describe a concrete way to obtain highly detailed data of the entire FLOSS ecosystem, described ways to clean, correct, and augment basic version control data with metrics needed to produce knowledge and code flow networks and create models that, through increased visibility, can help developers and organizations make better decisions resulting in a healthy and productive FLOSS ecosystem.

References

1. Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.
2. Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 137–143, New York, NY, USA, 2006. ACM.
3. Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Putting it all together: Using socio-technical networks to predict failures. In *17th International Symposium on Software Reliability Engineering (ISSRE 09)*, Bengaluru-Mysuru, India, 2009.
4. Wray L Buntine. Operations for learning with graphical models. *Journal of artificial intelligence research*, 2:159–225, 1994.
5. Hung-Fu Chang and Audris Mockus. Constructing universal version history. In *ICSE'06 Workshop on Mining Software Repositories*, pages 76–79, Shanghai, China, May 22-23 2006.
6. Hung-Fu Chang and Audris Mockus. Evaluation of source code copy detection methods on FreeBSD. In *5th Working Conference on Mining Software Repositories*. ACM Press, May 10–11 2008.
7. David Maxwell Chickering. Learning bayesian networks is np-complete. *Learning from data: Artificial intelligence and statistics V*, 112:121–130, 1996.
8. Martin L. Christopher. *Logistics and Supply Chain Management*. London: Pitman Publishing, 1992.
9. William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string metrics for matching names and records. In *KDD WORKSHOP ON DATA CLEANING AND OBJECT CONSOLIDATION*, 2003.

10. Thomas Fritz, Gail C Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):14, 2014.
11. Benjamin Fruchter. Introduction to factor analysis. 1954.
12. Daniel German and Audris Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd workshop on open source software engineering*, pages 63–67, 2003.
13. Richard L Gorsuch. Common factor analysis versus component analysis: Some well and little known facts. *Multivariate Behavioral Research*, 25(1):33–39, 1990.
14. David Heckerman. A tutorial on learning with bayesian networks. microsoft research. 1995.
15. Kurt Hornik, Friedrich Leisch, and Achim Zeileis. Jags: A program for analysis of bayesian graphical models using gibbs sampling. In *Proceedings of DSC*, volume 2, pages 1–1, 2003.
16. Oskar Jarczyk, Błażej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. Github projects. quality analysis of open-source software. In *International Conference on Social Informatics*, pages 80–94. Springer, 2014.
17. Michael Irwin Jordan. *Learning in graphical models*, volume 89. Springer Science & Business Media, 1998.
18. Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
19. Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.
20. Bernard J La Londe and James M Masters. Emerging logistics strategies: blueprints for the next century. *International journal of physical distribution & logistics management*, 24(7):35–47, 1994.
21. Douglas M Lambert, James R Stock, and Lisa M Ellram. *Fundamentals of logistics management*. McGraw-Hill/Irwin, 1998.
22. Quoc Le and Tomas Mikolov. Distributed representation of sentences and documents. In *Proceedings of the 31 st International Conference on Machine Learning*, volume 32, Beijing,China, 2014. JMLR.
23. Roderick JA Little and Donald B Rubin. *Statistical analysis with missing data*, volume 333. John Wiley & Sons, 2014.
24. Yuxing Ma, Tapajit Dey, Jarred M Smith, Nathan Wilder, and Audris Mockus. Crowdsourcing the discovery of software repositories in an educational environment. *PeerJ Preprints*, 4:e2551v1.
25. Roderick P McDonald. *Factor analysis and related methods*. Psychology Press, 2014.
26. Audris Mockus. Large-scale code reuse in open source software. In *ICSE'07 Intl. Workshop on Emerging Trends in FLOSS Research and Development*, Minneapolis, Minnesota, May 21 2007.
27. Audris Mockus. Amassing and indexing a large sample of version control systems: towards the census of public source code history. In *6th IEEE Working Conference on Mining Software Repositories*, May 16–17 2009.
28. Audris Mockus. Succession: Measuring transfer of code and developer productivity. In *2009 International Conference on Software Engineering*, Vancouver, CA, May 12–22 2009. ACM Press.
29. Audris Mockus. Organizational volatility and its effects on software defects. In *ACM SIGSOFT / FSE*, pages 117–126, Santa Fe, New Mexico, November 7–11 2010.
30. Audris Mockus. Engineering big data solutions. In *ICSE'14 FOSE*, pages 85–99, 2014.
31. Audris Mockus and James Herbsleb. Expertise browser: A quantitative approach to identifying expertise. In *2002 International Conference on Software Engineering*, pages 503–512, Orlando, Florida, May 19-25 2002. ACM Press.
32. Nachiappan Nagappan, Brendan Murphy, and Victor R. Basili. The influence of organizational structure on software quality: an empirical case study. In *ICSE 2008*, pages 521–530, 2008.
33. Judea Pearl. Bayesian networks. *Department of Statistics, UCLA*, 2011.
34. Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.
35. Peter Rigby, Yue Cai Zhu, Samuel M. Donadelli, and Audris Mockus. Quantifying and mitigating turnover-induced knowledge loss: Case studies of chrome and a project at avaya. In *ICSE'16*, pages 1006–1016, Austin, Texas, May 2016. ACM.
36. Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128. ACM, 2009.
37. Sunita Sarawagi and Anuradha Bhamidipaty. 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, 2002. ACM.
38. Murat Sariyar and Andreas Borg. The recordlinkage package: Detecting errors in data. *The R Journal*, 2(1):61–67, 2010.
39. Marco Scutari. Learning bayesian networks in r, an example in systems biology, 2013. http://www.bnlearn.com/about/slides/slides-useRconf13.pdf.

40. Giuseppe Silvestri, Jie Yang, Alessandro Bozzon, and Andrea Tagarelli. Linking accounts across social networks: the case of stackoverflow, github and twitter. In *International Workshop on Knowledge Discovery on the WEB*, pages 41–52, 2015.
41. Samuel L. Ventura, Rebecca Nugent, and Erica R.H. Fuchs. Seeing the non-starts: (some) sources of bias in past disambiguation approaches and a new public tool leveraging labeled records. *Elsevier*, Feb 2015.
42. Radu Vlas, William Robinson, and Cristina Vlas. Evolutionary software requirements factors and their effect on open source project attractiveness. 2017.
43. William E Winkler. Overview of record linkage and current research directions. Technical report, BUREAU OF THE CENSUS, 2006.
44. Qimu Zheng, Audris Mockus, and Minghui Zhou. A method to identify and correct problematic software activity data: Exploiting capacity constraints and data redundancies. In *ESEC/FSE'15*, pages 637–648, Bergamo, Italy, 2015. ACM.
45. Minghui Zhou and Audris Mockus. Developer fluency: Achieving true mastery in software projects. In *ACM SIGSOFT / FSE*, pages 137–146, Santa Fe, New Mexico, November 7–11 2010.
46. Jiaxin Zhu, Minghui Zhou, and Audris Mockus. The relationship between folder use and the number of forks: A case study on github repositories. In *ESEM*, pages 30:1–30:4, Torino, Italy, September 2014.