The perils and pitfalls of mining SourceForge

James Howison and Kevin Crowston
Syracuse University School of Information Studies
4-206 SciTech, Syracuse, New York, USA
{jhowison, crowston}@syr.edu http://floss.syr.edu

Abstract

SourceForge provides abundant accessible data from Open Source Software development projects, making it an attractive data source for software engineering research. However it is not without theoretical peril and practical pitfalls. In this paper, we outline practical lessons gained from our spidering, parsing and analysis of SourceForge data.

SourceForge can be practically difficult: projects are defunct, data from earlier systems has been dumped in and crucial data is hosted outside SourceForge, dirtying the retrieved data. These practical issues play directly into analysis: decisions made in screening projects can reduce the range of variables, skewing data and biasing correlations.

SourceForge is theoretically perilous: because it provides easily accessible data items for each project, tempting researchers to fit their theories to these limited data. Worse, few are plausible dependent variables. Studies are thus likely to test the same hypotheses even if they start from different theoretical bases. To avoid these problems, analyses of SourceForge projects should go beyond project level variables and carefully consider which variables are used for screening projects and which for testing hypotheses.

1 Introduction

We are interested in identifying factors that predict the performance of Free, Libre and Open Source Software (FLOSS) teams. As part of this inquiry, we chose to analyze data from the SourceForge website, the largest repository of FLOSS project data and, as such, an excellent source of data on FLOSS team practices [9, 1].

While Data mining is the process both of data collection and data analysis this paper focuses only on the challenges faced in our first steps of data collection. Our data collection process involved spidering numerous Web pages, parsing the downloaded HTML files and producing summary data for analysis. Our research encountered a number of practical pitfalls that this paper outlines. Yet the difficulties are not merely practical—in our investigation of SourceForge data and in other papers dealing with SourceForge data we have encountered a number of theoretical caveats that we present here.

1.1 Research background

Our interest in FLOSS stems from a broader interest in distributed teams. The focus of our research is on team practices: coordination, development of collective mind and individual and organizational learning. Therefore we intend to examine a number of projects in detail with both qualitative content analysis and ethnographic methodology. However given the sheer number of projects and the volumes of data available, a prerequisite to our research is to identify appropriate projects for detailed study. We are seeking both successful and unsuccessful projects using the model of FLOSS project success we explored in [3].

We collected data from the project demographics, developer mailing lists and the SourceForge Tracker system—which is largely bug tracking data. With this data we conducted social network analysis to identify variance in communication structure (Results reported in [5]), process analysis of bug fixing (Reported in [10]) and an analysis of the speed with which projects fix bugs (Preliminary analysis reported in [4]).

2 Avoiding Pitfalls in Data collection

After receiving no response to our requests for direct access to the SourceForge database, we concluded that the best available method of data collection would be to spider and "screen-scrape" the data. We initially spidered the SourceForge project pages in April 2002 and used this data to identify 140 projects that had greater than seven listed developers and more than 100 bugs in the system. These criteria were theoretically chosen to match our interest in distributed teams and the bug-fixing process. We spidered the mailing lists and bug-tracking pages in April 2003, accessing data on over 62,110 bug reports.

There were three stages in our data collection: Spidering, Parsing and Summarizing [1]. Each presented its own practical difficulties and necessary choices. We outline these, and our solutions, below. We utilized Perl scripts for the data collection—some comments are specific to Perl, most are not. We conclude this section with testing strategies that we recommend for mining software repositories.

2.1 Avoiding Pitfalls in Spidering

Our spidering scripts utilized the `WWW::Mechanize` module available from `CPAN` [2]. The unfortunate necessity of spidering large datasets can place large strains on the servers. It is therefore important to be well behaved both during the development of your scripts and in their use.

- Code a `-n` (`--do-nothing` or 'dry-run') option to test your scripts.
- Consider running a local test server that mirrors the structure of your target site.
- Store the full HTML download rather than parsing 'live'. This ensures that any changes to the parser (expect many!) will not require a 're-spidering' of your target site.
- It is tempting to use forked processes to speed up spidering. But beware of forking too many processes and especially of 'lost children' who, due to a bug, might 'bang away' at the server for days—long after the parent process is killed.
- Code a 'wait loop' into the spidering code to reduce the density of your page requests. At the time we spidered we were never banned from the SourceForge servers, although we have recently heard that others have been. It is believed that this is a new defence introduced by SourceForge, the parameters of which are unknown. The spidering process can take a long time, extending over a number of days. It is therefore crucial to be sure to collect all relevant data at the time of collection.
- Whenever feasible prepare your analysis scripts and test them on data spidered from one or two projects, ensuring that the data being collected is sufficient. Repeating the spidering stage can be very time consuming.
- Be sure to store the time at which the page was downloaded. This is especially important for time-dependent analysis that has to account for censored data (such as Event History analysis) but equally it is required to anchor the effective date of your analysis for comparative and longitudinal analyses.
- It is useful to store the number of pages of each type downloaded, which gives a count of the expected number of Bugs that should be found after parsing. This count can be used as a test for the accuracy of your parsing scripts.

Spidering is clearly an area in which cooperation between research groups could present great benefits. It is also vital to ensure that the SourceForge site is operating properly at the time you spider—this can be checked through the Site Status page [3].

2.2 Avoiding Pitfalls in Parsing

Large websites are generated from HTML templates and databases, giving them a fairly consistent structure suitable for scripted parsing to extract the required data for analysis. Yet the level of consistency is not high enough to ensure that unexpected problems will not be faced.

- Simplify your parsing process as much as possible by reducing excess or non-standard HTML on the pages. Test the results of utilizing the `HTML::Tidy` module or W3Cs 'tidy' application which does a good job of standardizing the HTML and removing 'cruft'. However be sure to check that this has not altered your target data in any way and that its effects are consistent across your downloaded dataset.
- While regular expressions are vital to this type of parsing, we found it far simpler to utilize them in combination with HTML parsing utilities such as `HTML::Parser::Simple` and `HTML::TableExtractor`.

Many of the inconsistencies encountered were contained only in a limited number of projects or even only within a few bugs or mailing lists, yet they can significantly undermine your confidence in the cleanliness of your data. We found these to be important points to be aware of in the SourceForge data:

- Line breaks in fields are especially tough to observe in regular debugging output. Consider converting linebreaks to visible characters to avoid confusion.
- Unexpected characters in fields, such as non-ascii characters or HTML entities. These often show-up as errors in external modules being utilized making the situation difficult to debug

[1] Our analysis scripts are available on request from the first author

[2] the Comprehensive Perl Archive Network—a 'class-library' for Perl.

[3] `http://sourceforge.net/docman/display_doc.php?docid=2352&group_id=1#1076697351`

- Another very frustrating bug was caused by usernames that look like html (Thanks, <DeXtEr> ! (gaim/482924)). Our Perl regex to parse the fields of username and Real Name was /(.*)\s*\((.*)\)/.

- The layout of the information on status changes in SourceForge was inconsistent in the table at the bottom of the bug. Three separate methods had to be used to find the correct close_date. It appears that SourceForge has now changed this layout.

Many projects are inconsistent in their use of the SourceForge system. Be especially aware of projects that have moved old data into the SourceForge system (e.g, tcl). The 'official' fields may contain misleading data (e.g., a Start Date reflecting the day of re-entry). While the free-text fields may contain the data from the old system in parsable form, researchers need to decide whether to write a special case parser for this data or to drop the project from the analysis. Also be aware that the SourceForge Tracker stores interaction information for each Item as 'follow-up messages' in free-text fields that are of arbitrary length and have inconsistent endings as well as containing unexpected characters. See dynapi patch 207106 for a tricky example. Our intention was to use XML for data storage between scripts. Beware though: XML::Simple cannot read all that it can write! We successfully used Storable (Perl data-structure serialization module) to store and pass the data between modules.

2.3 Avoiding Pitfalls in Summarizing

Summarization requirements will vary according to the intended analyses. We pursued Social Network Analysis (SNA), which required interaction matrices, and event history analysis, which required data on lifetimes, bug status and assignment.

One problem in summarizing is missing data. For example, SourceForge allows users to post anonymously, giving such posts the username of "nobody". Since we couldn't predict the effect that different treatments of the "nobody" data would have on our analysis, we created a summarizer that produced each of four treatments for "nobody data" 1. Baseline case: No treatment, "nobody" appears as an individual. 2. Deletion case: All nobody data deleted. 3. Each "nobody" as separate individual. 4. One "nobody" per Item or thread i.e. "nobody340078" as separate individual. We were then able to compare the effect that these different treatments had on the outcome of our analysis (we found surprisingly little difference between the last two strategies) [5].

An opposite problem is that a number of the fields of interest turn out to be multi-valued. For example, a project can be given a development status of planning, alpha and beta simultaneously. To retain these multiple values would make analysis very complex. Researcher ought to make principled decisions about how to handle such cases, rather than letting them be made for convenience in parsing or summarizing.

A final problem is that different analysis tools will require different data output formats. We tested over 5 different Social Network analysis packages before settling on the sna module from r-project for its high degree of scriptability, vital for large data sets. We also used the NetMiner application for its presentable graphics capabilities. Each program required output in subtly different formats. We found that our summarization methods were being used in a number of different scripts, making summarization an excellent candidate for modularization.

2.4 Testing Strategies

On reflection, our testing strategy should have been considerably more systematic. We would recommend these techniques to those pursuing large data-collection projects involving spidering and parsing:

- Random selection of test pages (at least three from each project) that should be checked by hand to create known good output.

- When making changes to the parser or summarizer, preserve the output of earlier runs to check for unexpected regressions. One strategy would be to diff old and new results, noting the items whose values have changed and check that against the intended and anticipated changes.

- Remember to note in your comments the bug reports for which special cases of code are written (or alternations in regular expressions). It is surprising how quickly these are forgotten in the flow of bug-fixing.

- Further, it would be useful to create test cases for each quirk identified, both to ensure the correctness of a proposed solution and to prevent regressions.

- Test cases could be shared with others seeking to parse similar data from the repository of interest (e.g., we could have a location to share test cases for SourceForge, CVS, Bugzilla, Subversion, mailing lists etc.)

3 Interpretation and Analysis

There are several important issues to consider when undertaking analysis and interpretation of SourceForge data. Those seeking to utilize this data must carefully consider their choice of screening variables and keep these separate from their analysis variables.

3.1 Challenges in cleaning dirty data

Despite the template and database nature of the SourceForge website there is a significant amount of 'dirty' data and it is hard to be sure of the extent of these problems without time-consuming and costly manual checking.

As described above, there is a large amount of anonymous data in the SourceForge system that cannot be attributed to any individual participant. For some analyses this will not have an impact but it could be crucial for others. Also described above, there is data that has been 'dumped' into the system, yielding valid yet totally inaccurate data.

Furthermore SourceForge has become the 'repository of record' for the FLOSS community, yet for important projects it is not the 'repository of use'. For example `vim`, an important programmers editor, is listed at SourceForge but has only 3 developers and 0% activity and has not released any files—all clearly wrong. The page is simply a placeholder that points to the `vim` 'repository of use'. It is likely that there are many entries like this and identifying them is difficult, at the very least it requires the use of a data source outside of SourceForge.

3.2 Skewed and truncated data

Firstly the projects in SourceForge are of highly different shapes, sizes and structures, which leads to much of the data being highly skewed. For example our screening conditions (> 7 developers and > 100 bugs) reduced the projects of interest from the 52,000 hosted by SourceForge at the time of spidering to only 140 projects. This skew extends to project activity and is reported in the findings of [7]. It appears that there are a very large number of one-person projects entered into SourceForge that never progress beyond announcement [9].

The problems above, and their possible solutions, may yield truncated results which complicate variance analyses, such as regressions. When it is necessary to choose screening variables to reduce the dataset to the projects of theoretical interest, the analysis must acknowledge that the variance in those screening variables has been significantly reduced and attempt to compensate for this reduction (or better still avoid the further use of the use of that variable at all).

Even avoiding the use of screening variables may not be sufficient, because when a dataset is reduced by screening on one variable there may be significant truncation of correlated variables. Even worse, it is difficult to know this in advance without collecting all the data to look for these correlations.

These difficulties are further compounded by the difficulty in predicting the direction of the impact of the truncation. For example truncating a variable that is highly varied may result in a variable that appears to be less varied, increasing correlations. But truncating a variable that exhibits low variance can cause the variable to increase its relative variance, leading to reduced correlations (Figure 1). Equally a decision to truncate only sections of a variable (either the top and bottom, or the mid-range) can have quite unpredictable effects (in Figure 2 substantially increasing the correlation).

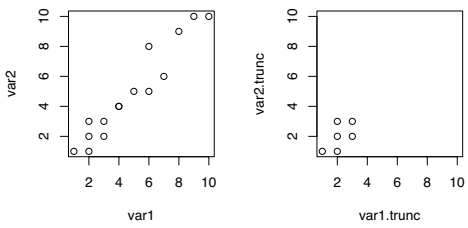

Figure 1. Truncating the upper range of var1 reduces the correlation from 0.91 to 0.52

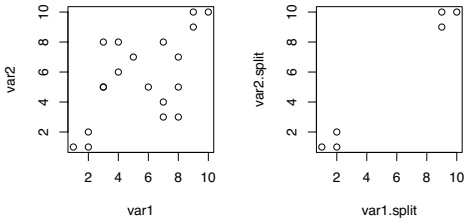

Figure 2. Splitting the distribution increases the correlation from 0.57 to 0.99

4 Peril in Research Design

SourceForge is a highly available dataset, but it provides only a limited number of easily available variables, that is, variables that are pre-calculated and available from each project's homepage or in full lists. Examples include: Number of developers, Project status, Activity, Downloads, Page Views and Numbers of Tracker items. The restricted amount of data poses problems in research design.

One problem is that there are only a limited number of ways of mapping theoretical constructs on to these available variables. As a result, studies using SourceForge data may end up with similar regression equations while purporting to study quite divergent concepts. For example, Crowston and Scozzi present an analysis of OSS communities as virtual organizations by applying Katzy and Crowston's competency rallying theory to the case of OSS development

projects [8, 6]. The theory explains project success in terms of the availability of competencies, the ability of developers to recognize opportunities, the ability of project to marshall resources and the ability to manage short-term cooperation, but end up regressing downloads, development status and activity against development status, number of administrators, popularity of the programming language, number of developers, lifespan, activity and factors for audience and topic (omitting as necessary the dependent variables as independent variables). Chengalur-Smith and Sidorova start with a population ecology perspective on projects, but propose to regress project survival against number of releases, number of development statuses, audiences and topics, age and number of developers [2].

A second problem is that these available variables may have low validity as measures in any particular theory. For example, many studies have chosen to use downloads as their dependent variable, arguing that it is a plausible proxy for "use". This is problematic for two reasons. Firstly, in the Information Systems literature "use" is already largely a proxy for "impact" [11], so downloads becomes a proxy for a proxy. Secondly, downloads is not even a good proxy for use, being both inaccurate and systematically biased. Much fundamental FLOSS software is distributed through distributions (e.g., RedHat CDs or Debian's `apt-get` system and FreeBSD's `ports`) and therefore not often downloaded from SourceForge (how often has a user downloaded the `vim` program or the `Xfree86` distribution directly from SourceForge?). Conversely userland packages facing frequent changes in their environment (e.g., `Gaim`, an instant messenger package), or new packages not yet included in a distribution, would have higher downloads. We deal with similar difficulties with alternative dependent variable measures and develop, hopefully, a more useful approach to FLOSS project success in [4, 3].

5 Conclusion

SourceForge remains an excellent source of data for those interested in studying the processes of FLOSS teams and distributed teams in general—one of many such repositories. Screen-scraping remains an unfortunate necessity faced by researchers seeking to mine online repositories. We have presented our experiences in mining SourceForge, and made available our code. We have also sought to highlight the general lessons for mining software repositories.

Regardless of data collection method those wishing to use sourceforge data face significant challenges in cleaning, screening and interpreting the data, we have outlined those we have identified and the solutions we employ: researchers should be tuned to the impact of their screening and attempt to minimize the impact of that screening on their analyses.

Finally, as a discipline, we must be conscious of the limitations of the 'ready-made' data-points available through repositories such as SourceForge. Researchers must take care in operationalizing their theoretical constructs and should be prepared to go well beyond the "low hanging fruit".

Once these challenges have been faced in gathering data, SourceForge can produce useful research results. In [5] social network analysis showed that FLOSS projects vary widely in their communications centralization and our correlations demonstrated that larger projects decentralize into a 'shallot' shaped structure. In [4] our event history analysis of bug fixing produced an intriguing measure of team performance which is sharply differentiated from other measures of project success. In [10] we mapped clear coordination practices in the bug fixing process. We are continuing to explore this intriguing dataset.

References

[1] A. Capiluppi, P. Lago, and M. Morisio. Evidences in the evolution of os projects through changelog analyses. In *Proceedings of the 3nd Workshop on Open Source Software Engineering, Int. Conf. Software Engineering*, 2003.

[2] S. Chengalur-Smith and A. Sidorova. Survival of open-source projects: A population ecology perspective. In *Proc. of 24th International Conference on Information Systems (ICIS '03*, Seattle, WA., 2003.

[3] K. Crowston, H. Annabi, and J. Howison. Defining open source software project success. In *Proc. of International Conference on Information Systems (ICIS)*, 2003.

[4] K. Crowston, H. Annabi, J. Howison, and C. Masano. Towards a portfolio of FLOSS project sucess measures. In *ICSE Open Source Workshop*, 2004.

[5] K. Crowston and J. Howison. The social structure of open source software development teams. In *OASIS 2003 Workshop (IFIP 8.2 WG)*, 2003.

[6] K. Crowston and B. Scozzi. Open source software projects as virtual organizations: Competency rallying for software development. *IEE Proceedings on Software*, 149(1):3–17, 2002.

[7] R. A. Ghosh, G. Robles, and R. Glott. Free/libre and open source software: Survey and study floss. Technical report, International Institute of Infonomics,, University of Maastricht: Netherlands, 2002.

[8] B. R. Katzy and K. Crowston. A process theory of competency rallying in engineering projects. In *Proc. of CeTIM*, Munich: Germany, 2000.

[9] S. Krishnamurthy. Cave or community?: An empirical examination of 100 mature open source projects. *First Monday*, 7(6), June 2002.

[10] B. Scozzi and K. Crowston. Coordination practices for bug fixing within FLOSS development teams. In *First International Workshop on Computer Supported Activity Coordination (CSAC 2004)*, Porto (Portugal), 2004.

[11] P. B. Seddon. A respecification and extension of the delone and mclean model of is success. *Information Systems Research*, 8(3):240–253, 1997.