Not Surveying Developers and Code About Languages

Leo A. Meyerovich
UC Berkeley
lmeyerov@eecs.berkeley.edu

Ariel Rabkin
Princeton University
asrabkin@cs.princeton.edu

Abstract

We present cross-sectional analyses of programming language use and reflect upon our experience in doing so. In particular, we directly survey 1500-13000 developers and indirectly do so by mining 200000 repositories. Our analysis reveals programming language adoption phenomena surrounding developer age, birth year, work place, opinions, gender, and choice of software repository.

We find that survey methods are increasingly accessible and relevant, but there are widespread problems in examining developers and code in particular. Prominently, we show that analyzing software repositories suffers from the same sample bias expected from directly polling developers. These problems question, for example, the scope of results of the increasingly common research practice of mining software. We conclude by describing concrete practices and opportunities for amplifying results of developer and code surveys.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: general

General Terms Languages, Human Factors

Keywords history, sociology, programming language adoption

1. Introduction

The programming language design community largely focuses on technical aspects of languages: how to efficiently implement them, how to automatically reason about programs written in them, and how to prove properties, such as type safety, about the language itself. However, programming is about more than technical aspects. Software development is a human process carried out in a social context, and psychological and sociological factors can make the difference between a successful language or an unsuccessful one. In a past publication, we called for the programming language community to devote more attention to analyzing the social processes that lead to language adoption [3].

There has been increasing interest in the social and psychological aspects of programming. Small-scale user studies are increasingly common and mining software repositories community even has its own long-running conference series. However, a missing technique relied upon in social research across many fields is large-scale cross-sectional surveys of people. These occur in parts of industry today, but they are quite rarely used in computer science as a research instrument. With the popularity of programming and the rise of the Internet, it is now relatively easy to do large-scale cross-sectional surveys of developers. Should we, and if so, how?

On and off again for two years, we have been launching and analyzing mass surveys of developers and repositories to understand the process of programming language adoption. This paper reflects on the methods we used, and particularly their strengths, weaknesses, and some of the pitfalls we encountered. Our audience is programming language and software engineering researchers who might wish to pursue similar questions or methods.

In particular, we examine three topics:

- **Analyzing language adoption through big yet sparse data sets.** We describe new results (including with respect to our concurrent work) relating to: programmers ranking languages according to various statements, programmer age, year of birth, and gender, and ZZZ. Many of our analyses are due to inherently sparse high dimensional data, so we describe algorithms and visualization techiques for understanding it.
- **Survey setup.** We discuss how the wording of questions can bias the results. **[[For example, ...]]**
- **Sources of sample bias in developers.** We show how developer demographics shape results of developer surveys. We also find demographic biases in software repository surveys. Put together, these highlight real methodological dangers in the increased use of mining programs for programming language and software engineering research.

Overall, we show surveys are an effective methodology – if used correctly. We end on a promising note by describing emerging opportunities for performing effective ones. For

Plateau 2012, October 19–26, 2012, Tucson, Arizona, USA.

Survey	Conducted by	Description	Scope
Hammer	David MacIver	Survey of developers about 50 languages and comparing 100 properties about them. No demographic information maintained.	13000 people 2 years
MOOC	David Patterson (course instructor), with advice from us	Entry survey for two offerings of a Massive Open Online Course in software engineering for software as a service.	10000 and 1100 people Half a year apart
Slashdot	us	Survey posted by us to understand audience from Hammer visualization	1600 people 2 weeks
SourceForge	SourceForge	Project descriptions from a massive open source software repository	10 years 200000 projects 260000 people

Table 1. Data sources mentioned in this paper

example, while traditional student surveys do not sufficiently sample developers, massive open online course surveys will.

The structure of this paper parallels the trajectory of our research. We started off analyzing an existing data set, with minimal demographic information; this is presented in the next section of this paper. That experience left us with a number of hypotheses. We arranged to have a number of questions about programming experience and beliefs added to the course survey for the Berkeley Massive Open Online Course in software engineering for software as a service. Our analysis of the Hammer data attracted significant social media attention, which we used to collect responses for our own mass survey. We refer to this survey as the Slashdot survey, labeling the source of many of the responses.

Beyond the general problems of survey research, we found a number of challenges specific to surveying programmers. Section 3 discusses some of the challenges in wording questions. Following that, Section 4 looks at demographic issues.

Our conclusion is three-fold. First, surveys are a powerful but rarely utilized research instrument for basic questions in programming languages and software engineering. Second, there are many emerging and underutilized opportunities for performing surveys. Finally, survey methodologies from other fields apply to our own. We especially stress analyzing and controlling for demographics before reporting conclusions about surveys. This warning applies irrespective of whether they are directly of developers or indirectly through software repositories.

2. Hammer: Sparse High-Dimensional Data

We describe our analysis of a survey of a sparse high dimensional data set. Our approach was to couple machine learning algorithms with interactive visualizations. Doing so exposed **xyz**, including exposing results **abc** that we were not explicitly looking for. This combination, though often with more lightweight variants, was common throughout our surveys.

Figure 1. The selection phase of Hammer Data gathering

The survey is of particular interest because of its scope: it provides a significant amount of data about how developers compare languages according properties such as correctness, speed, simplicity, job prospects, and enjoyment. Our interactive visualizations are available online[1]. We believe they can help other researchers frame hypotheses worth investigating carefully.

2.1 The Hammer Principle Survey

"The Hammer Principle" is a website by David MacIver that invites readers to compare programming languages based on a high-dimensional series of metrics [2]. He (graciously) provided us with anonymized survey results. Over two years, respondents came in bursts from popular online sites such as Slashdot, Hacker News, Reddit, and Lambda the Ultimate.

They went through the following process:

1. **Pick languages** Respondents picked a set of languages that they are comfortable out of a pool of 51 (Figure 2). The average set had about 7 languages.

[1] `www.eecs.berkeley.edu/~lmeyerov/projects/socioplt/viz/index.html`

Figure 2. The ranking phase of Hammer Data gathering

2. **Rank languages by statement** Respondents were shown a series of statements. For each statement, a respondent ordered the previously selected languages based on how well they matched the statement (Figure 2). The average respondent answered 10-11 questions.

Over 13,000 people filled out the survey. Challenging analysis, individual responses are sparse and often contradictory, so we use machine learning algorithms to compute strong relationships with high confidence. Furthermore, the data set is high-dimensional (110 statements about 50 languages), so we made several interactive visualizations to explore statistical analysis results.

2.2 Glicko 2 Ranking Algorithm

The Glicko 2 chess ranking algorithm drives our first visualization and subsequent analyses. The first interactive visualization compares languages according to how well they match different statements about them. This provides a human-understandable fingerprint for each language and reveals languages that rank similarly – or fail to – for certain properties.

Consider a respondent ranking a few languages for some statement X. The languages might be given order A $>$ B $>$ C. We treat this as the outcome of 3 different matches between players A, B, and C in sport X, where A $>$ B, A $>$ C, B $>$ C. While respondents only looked at a total of 140,000 statements, these sequences expand into 4 million pairwise comparisons. This is good coverage because the full space of pairwise comparisons is less then 300,000 points.

For a given statement X, globally ordering the languages is difficult. There is no logically consistent ordering – one respondent may rank A $>$ B and another B $>$ A. Worse, the data is sparse for unpopular languages.

The Glicko 2 ranking algorithm is designed for this scenario [1]. It is commonly used in sports: Glicko generalizes the original Elo rating system and underlies XBox's TrueSkill online player rankings.

Glicko proceeds as a simulation. A weak language beating a strong language gives the weak one a big boost in score and the strong one a drop, while there is little change in ranks from a strong language beating a weak one. Time and contention is factored in by tracking the deviation across matches: an occasional upset is disregarded, but if the upsets become consistent (e.g., a language was upgraded), the rank will converge on the new value. Likewise, high disagreement about a statement is reflected by a high deviation.

2.3 Interactive Visualization of Language Rankings

We created an interactive visualization to explore the matrix of language vs. statement (Figure **??**). It is effectively a heatmap. Each row shows how a language ranks relative to others according to a particular statement: a big green circle shows that most people agree that the language matches the statement better than other languages. In contrast, a small red circle shows most agree it matches the statement less than most other languages. A language's row, in practice, is a fingerprint for quickly comparing languages. Circles different from their neighbors are generally of interest, as are particularly small or big ones.

Clickin on the list of languages sorts them, and toggles control which languages are statements are shown. Clicking on a circle will simply sort the languages by how well they match the statement corresponding to the circle. While simple, these three interactions enable the pattern of focusing on a statement or language, finding an unusual case of it, and then seeing how other languages and staetments compare.

For example, we lookup up Coq's fingerprint and found that the strongest statement programmers make (the biggest circle in the row) is that they do not feel smart enough to write in it. Curious about how Coq relates to other languages in this problem area, we clicked on the circle. Figure **??** shows that the two languages programmers feel least smart enough to write in are Coq and Haskell and then, with nearly equal reservations, Prolog and Factor. Visual inspection quickly revealed programmers believe these languages are inflexible, have an acceptable syntax, and they infrequently use them. Except for Haskell, the languages did not have particularly desirable features.

The visualization also led tweaking our ranking algorithm. In particular, we found REBOL to rank high in categories we did not expect. The basic reason is that it had a high deviation; it was placed highly with little confidence. This can be due to a combination of conflicting responses and overly sparse data. Our solution was to discount the ranking by $3 * \sigma$, similar to XBox player rankings, and visualize such contentious statements using transluscent circles. Hovering over a circle clarifies both the raw score and deviation.

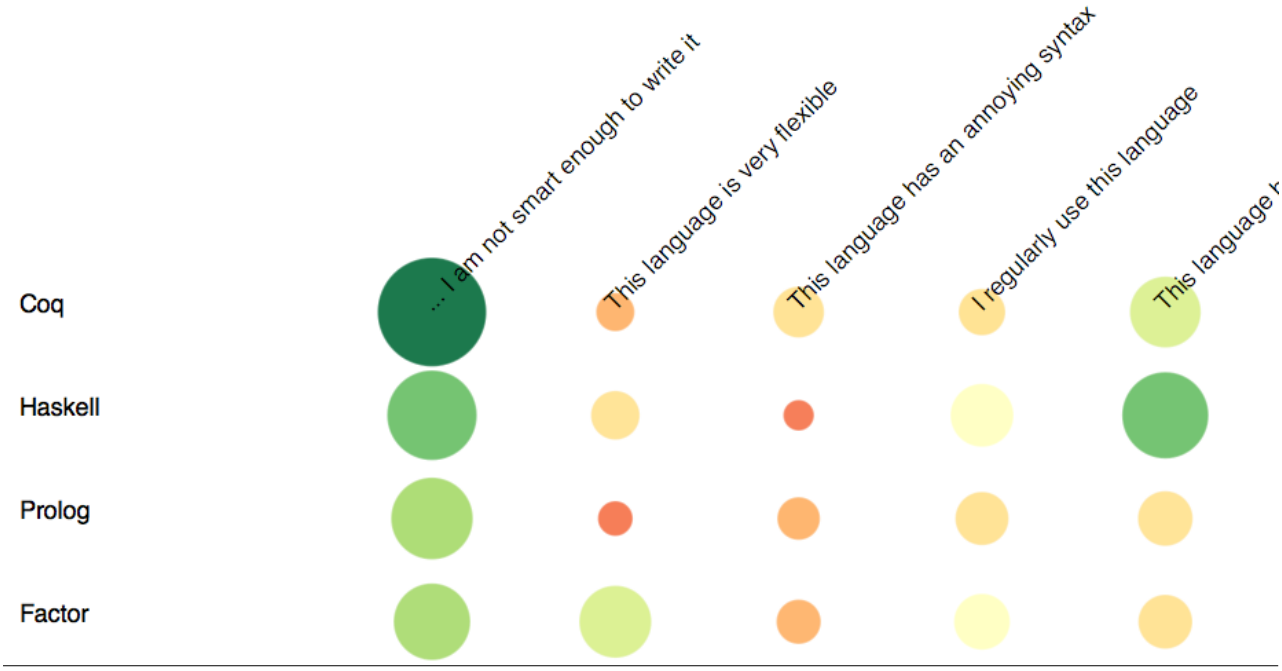

Figure 3. Interactive visualization of language rankings. Current state shows a filter for 4 particular languages across 5 particular statements sorted by match against the statement "I often feel like I am not smart enough to write this language." The clipped statement is "This language has unusual features that I often miss when using other languages."

2.4 Correlations and Clustering

As our research interest was about general language phenomena, not particular languages, we then analyzed the correlations across statements in the computed rankings. We reused the interactive heatmap for navigating the data.

We found many surprises. Perhaps most prominently, "This language has a strong static type system" strongly anti-correlated with "This language encourages writing reusable code." This conflicts with basic and long-held beliefs about the nature of modularity by the functional programming community [**?**]. We only suspect usability, not modularity, would rank low. Another surprise was that terse languages anti-correlate with annoying syntax: given long-standing critiques of "write-once" languages, we expected the reverse. This case is interesting in that, as language researchers, we were more interested in semantics. For both the anti-modularity of typed code and legibility of terse code, the visualization highlighted properties that dispute conventional wisdom and that we would have otherwise overlooked.

Finally, we computed the k-means clusterings of statements and languages. We use this both to expose relationships and improve the earlier visualizations.

Consider the first cluster of statements shown in Figure **??**. The numbers indicate distance from the center of the cluster: its average distance of 3.3 is good. The first and last statements about helpful conventions and dogma pairing together are unsurprising because social conventions often arise to solve problems. The second statement about debugging is fairly specific and goes against the common belief that patterns and conventions are often viewed as symptoms of linguistic defects, such as argued by Peter Norvig. [2] However, it seems that fixing these presumed defects may not be important. Language research for patterns may be a case of the streetlight effect, focusing on what is known and ignoring what is not.

A subtlety of the clustering is that, for the language rankings used to cluster statements, the languages did not have to rank highly for the statements to go in the cluster. They could be similar at any value, as long as it is consistent. The visualization therefore shows which languages support the cluster, and their rank for the center-most statement. Not shown, for the above clustering, languages within 5% of the center have average rank 43 with standard deviation 10. The rank is not high relative to other statements – languages are generally not considered overly dogmatic – but on the scale of dogma that languages exercise (according to the ranking visualization), it is.

We also used the clusterings to improve the original ranking visualization. Showing the matrix of all 50 languages and 111 statements was overwhelming and slow. Instead, by default, we only show the center-most items the statement and language clusters. For example, of the statements in Figure **??**, we would only show the first about conventions and fourth about well-organized libraries. If a user wants to ex-

[2] http://norvig.com/design-patterns/

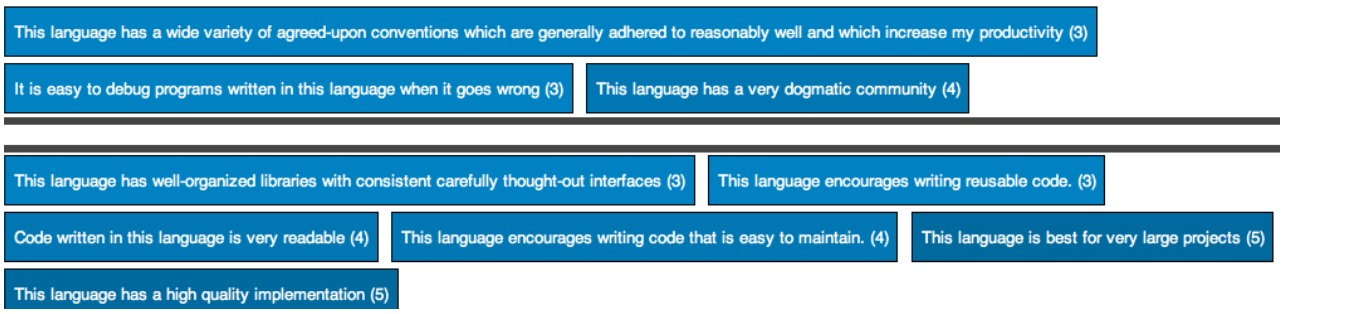

Figure 4. Interactive visualization of statement k-means clustering.

plore a particular family of languages or statements, they can be expanded.

3. Framing the Right Questions

This section talks about biases introduced by the questions

Developers don't necessarily know as much as they think they do. MOOC data shows that they overestimate how much they use things.

3.1 Names for language features

Software development is a technical field, and therefore has a technical vocabulary. However, researchers and practitioners do not always share the same terminology.

One question on the MOOC survey asked developers how often they created their own generic classes in Java. (A generic class is one that has a type parameter, such as `class Foo<T extends Collection>`.) In our sample, X% said they did this often or sometimes. This result is hard to believe, since a survey of existing code by XYZ found that only YYY% of developers are actually responsible for doing so. (Anedcotal evidence and our personal experience agrees with the study by Bird.)

There are two possible interpretations of this dichotomy. It may be that professional developers work very differently to the open source developers studied by XXX. We believe, however, that developers misunderstood the question. We had separately asked developers about *creating* generic classes and about *using* generic classes, such as the Java standard library `ArrayList<T>`. We think this distinction is not as clear to developers as it is to us.

For another example, consider the concept of determinism, that a piece of code should produce the same behavior each time it is invoked. We asked developers how important this was. We found that XYZ% of developers responded

A lesson to take away from both

3.2 Knowing a language

A basic question about programming language adoption is how many languages developers know. We asked this question several times, to several different audiences. The Slashdot survey (Figure 3) asked developers to list the number of languages they knew well, and separately, to estimate the total number of languages they know. As can be seen, there is virtually no age trend; developers of all ages list an average of six languages they know well, and claim to have ever learned about ten.

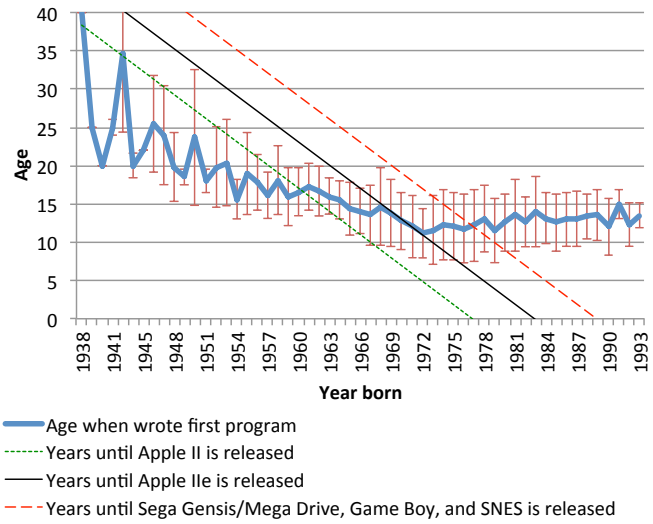

Figure 5. Age at time of first "Hello World" over time. Ages decrease until people born in 1972, after which ages climb until leveling in 1982.

A similar question was on the the MOOC survey. That survey asked developers to list the languages they knew well, and to separately list the languages they knew slightly. The results are shown in Figure 5. As can be seen, the result is broadly similar to the above. *DISCUSS DIFFERENCES HERE*

This lack of change might suggest that developers learn their languages early, and never learn more. This does not seem to be the case, however. Figure 4 shows the mean age for developers who claim to know a variety of languages, along with the 25th and 75th percentiles.

Developers who claim to know a variety of different languages have virtually the same age profile. The only exception is Ada, which skews old. Languages like Python and Ruby are much more popular now than they were a decade ago, while C is less popular. Even so, the age statistics for

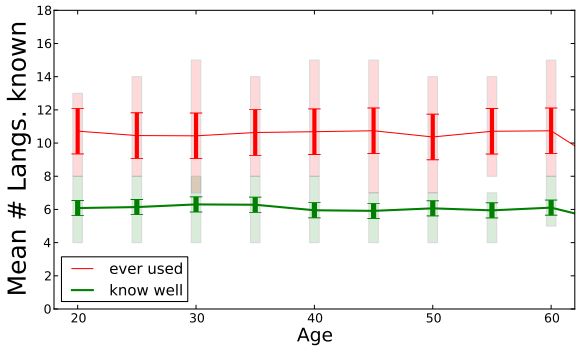

Figure 6. Developers of different ages seem to know a similar number of languages. Lightly shaded rectangles show 25th and 75th percentiles, darker solid bars show standard error of mean. (Slashdot)

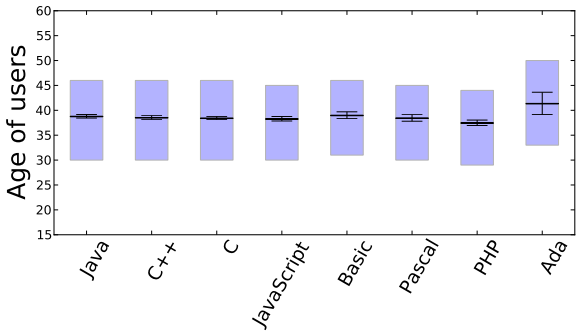

Figure 7. Developer age does not vary significantly across most popular languages. (Slashdot)

the developers are similar. This is evidence that developers are routinely willing to learn new languages.

This leaves us with an inconsistency: if developers are routinely learning languages, why does the number of languages they know not rise over their careers? Or at least, why does their estimate for "languages ever learned" not rise over time? We suspect that developers are forgetting languages, or at least, forgetting to mention them. This suggests that asking developers to estimate the number of languages is not a reliable technique.

A broader problem is that we are unconvinced that different developers interpret "knowing" a language the same way. There must be some minimum level of mastery or comfort before a developer will claim to "know assembly." Is this level the same as that required to claim to "know Python"? We suspect not, but have no good way to assess this.

4. Asking the Right People

This section talks about biases introduced by demographics

Not all developers are alike. The same word might describe hobbyist users of Visual Basic, semi-skilled PHP de-

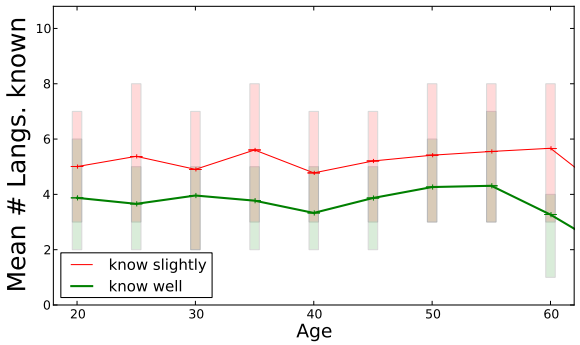

Figure 8. Developers of different ages seem to know a similar number of languages. Lightly shaded rectangles show 25th and 75th percentiles, darker solid bars show standard error of mean. (MOOC)

velopers, domain experts in scientific computing, and expert distributed systems programmers at a large Internet services company. Asking about "the average developer" or "the true population statistic" requires picking out some subset of the people who have ever programmed and defining them as the population of interest.

On our slashdot survey, we asked developers how quickly they learned the language they used for their last project. We found that developers learned faster for hobby projects than for work projects. For work, only half the users learned within three months whereas it was over 60% for hobby projects.

One explanation might be that developers work harder on their own projects. Another explanation is to observe that not all developers will program for fun. Developers who program as hobbyists are likely to be biased towards the developers who enjoy programming more, and who may consequently be better at it.

This shows that professionals and hobbyists are not interchangeable. The hobbyists may be drawn from those developers who are the most fluent and capable. A consequence of this is that open-source development, with a heavy hobbyist contingent, may not be a reliable proxy for closed-source development, which is usually conducted by professionals. This is a threat to the generality of research that tries to extrapolate from open source to all kinds of programming.

4.1 Sample Bias in Open Source Repositories

We found that the population dynamics change over time for the SourceForge dataset. Figure **??** illustrates several trends by showing, for each language, which months it was most used in. The rise (and fall) of each language varies. For example, D spiked in late 2006, and AspectJ plummeted in early 2009. We also found that the best single predictor for the language of a project is the language used on the previous one, which is true 30% of the time. The changes in popularity, and the tendendency to reuse a language, mean

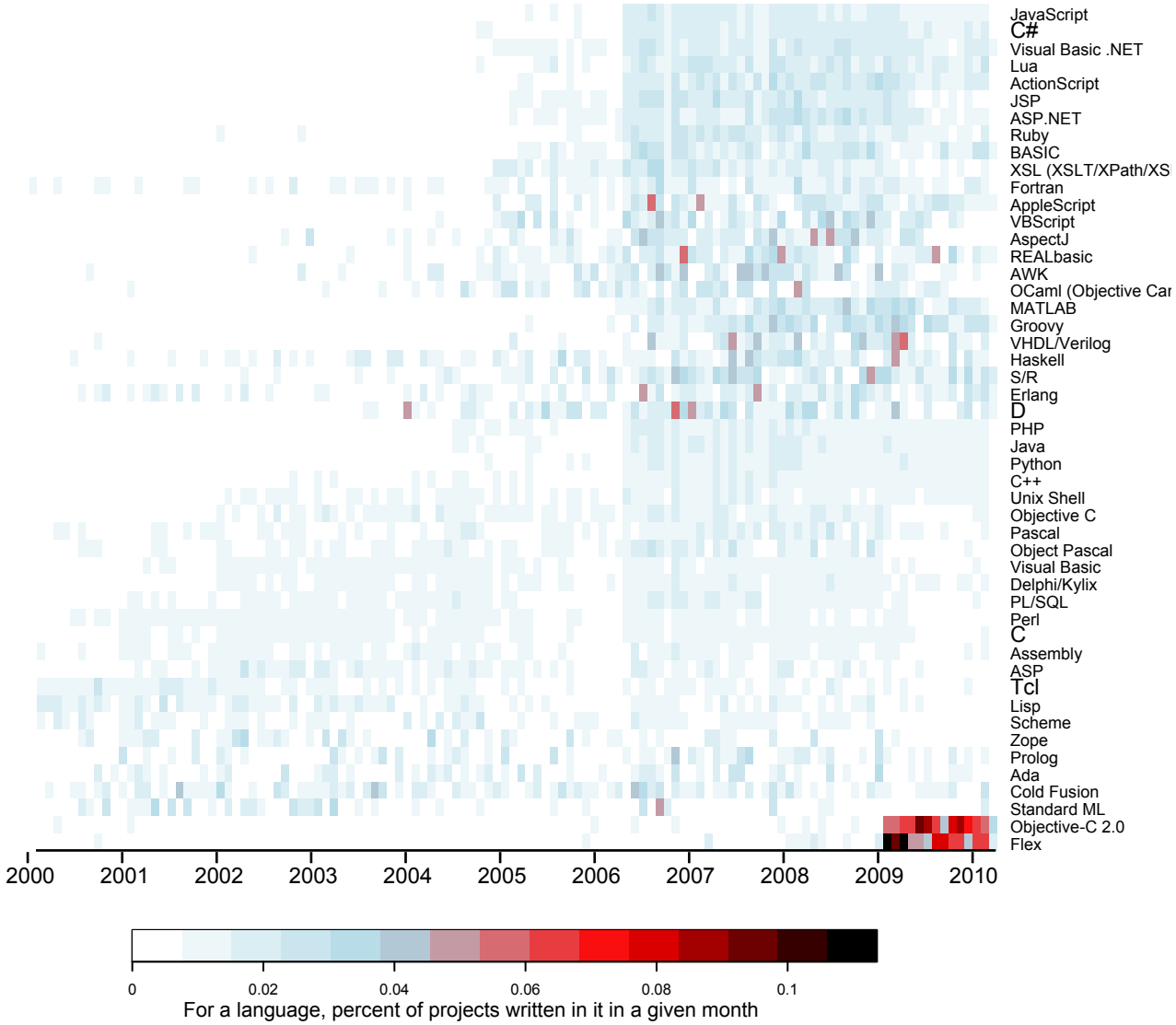

Figure 9. Relative language use over time. (SourceForge data set). Each row is independently normalized to itself and languages with fewer than 100 projects are elided.

that the year in which a project is developed in a language reveals artifacts about the developer. Sampling from popular years vs. unpopular ones may be subject to experienced and unexperienced language years. *The year a project is written in a particular language matters in understanding the type of developer.*

Signficantly, overall use of SourceForge waxed and then waned. Mid 2006 began a long-running boom that began to collapse in 2009. Not shown, major reason is the rise of alternative repositories. GitHub launched in April of 2008, reached 40,000 repositories by early 2009 (25% of the shown SourceForge projects), and 1 million total repositories by July of 2010. In April of 2011, it hit 2 million repositories.[3] Bitbucket also launched in 2008, and Google Project Hosting has been available at least by 2007. These repositories vary by language and license. *The repository selected for a project matters, and by year.* For example, projects at inflection points are made by early adopters. Likewise, a project not using a language specific repository may be due to an estranged language user, and therefore one who deviates from linguistic norms.

We see that the language, repository, year, and developer history are all considerations when analyzing an open source project.

[3] https://github.com/blog/455-100-000-users, https://github.com/blog/936-one-million, https://github.com/blog/841-those-are-some-big-numbers

4.2 Sample Bias in Work Environments

5. Conclusions: Surveys are a Delicate Opportunity

Analysis is important Important results XYZ popped out. (short paragraph, not contentious).

phrasing exists in forms XYZ, can be surmounted by ZZZ

bias exists in forms XYZ, can be surmounted by ZZZ

Emerging data collection opportunities Businesses routintely perform them. MOOCS (as opposed to normal schools). Language developers are pretty interested in this (e.g., Scala does analytics, the clojure annual survey). As a subtle point, practicing language developers survey code, such as to understand legacy impact. As data-driven methodologies become more ingrained and acceptable, we expect more of this.

Acknowledgment

We thank David MacIver for graciously offering us the data from the Hammer Principle website.

References

[1] M. E. Glickman. Parameter estimation in large dynamic paired comparison experiments. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 48(3):377–394, 1999. ISSN 1467-9876. doi: 10.1111/1467-9876.00159.

[2] D. R. MacIver. The hammer principle. `http://hammerprinciple.com/therighttool`, 2010.

[3] L. A. Meyerovich and A. Rabkin. Socio-plt: Principles for programming language adoption. In *ONWARD*, 2012.