Discussion paper: Metacommunities of practice in the code-sharing commons

Adrian Mackenzie (Lancaster),

(Goldsmiths),

Andrew Goffey (Nottingham),

Richard Mills (Cambridge),

Stuart Sharples (Lancaster)

December 2014, a.mackenzie@lancaster.ac.uk

Introduction

> A metacommunity is a set of interacting communities which are linked by the dispersal of multiple, potentially interacting species [@Leibold_2004].

> One of the areas that being most dramatically shaken up by $N = All$ is the social sciences. They have lost their monopoly on making sense of empirical social data, as big-data analysis replaces the highly skilled survey specialists of the past. ... More important, the need to sample disappears [@Mayer-Schonberger_2013, 30].

The 'Metacommunities of practice in the code-sharing commons' project was an $N = All$ style analysis of data relating to open source code repositories. By $N = All$, we mean that it sought to make use of all the publicly available data in the domain of research. The title of the project is a slightly loopy one, but relates to this $N = All$ approach in important ways. First of all, the domain of the research was software, code, and coding practices as seen in public software repositories. We focused almost solely on Github.com, the leading public code repository today. We could have done comparative work with other platforms such as GoogleCode or SourceForge, but working with all the data from Github was substantial enough. We did not sample the data on code and coding practices on Github. We sought to analyse all the code repositories precisely because we wanted to look for evidence of the *commons* in code. Our guiding theme of 'meta-community' was meant to open up a different perspective on the flow of code across platforms, devices, apps and applications.

This perspective contrasts with much of the existing social science that focuses right down on exemplary cases (often by doing fieldwork with particular communities of software developers such as Debian [@Coleman_2012]), and diverge also from the extensive work done by computer scientists and software engineers who do look at the aggregate phenomena but tend to focus on issues of productivity, reliability or code quality on particular projects (often by using software repositories as their data source [@Godfrey_2012]). In contrast with both the existing social science and the computer science, we were looking for transverse flows and connections between different parts of the seething pool of software development. More importantly, we were treating the repositories in aggregate as the form to be analysed, rather than seeing them as a site for sampling of independent, representative cases. The overarching assumption was something like what Viktor Mayer-Schőnberger and Kenneth Cukier suggest in *Big Data*: 'using all the data makes it possible to spot connections and details that are otherwise cloaked in the vastness of the information' [@Mayer-Schonberger_2013, 27].

Research objectives and their practicalities

This discussion paper is framed by the research objectives we formulated at the time of the grant proposal in late 2012. We are also taking the notion of the 'working paper' fairly literally in what follows, so we have not shied away from including code, data, and graphics to illustrate the discussion.

Our research objectives in order of priority were:

1. Using datasets generated from publicly accessible software code repositories and programmer question and answer (Q&A) sites, to explore ways of tracking programming practices as they move through digital economies/network media cultures.
2. To develop an empirically rich and reusable database of evidence describing the extent and diversity of code sharing practices across a comprehensive range of open-source software projects during the period 2008-2012.
3. To explore and document how data analytic techniques including visual data exploration, statistical machine learning and predictive models can be re-purposed to focus on questions of practice that are normally seen as the province of qualitative case-studies.
4. To investigate whether analytical and methodological problems in the social sciences often framed in terms of the 'empirical crisis of sociology' [@Savage_2009] or the dominance of social sciences by the 'general linear model of reality' [@Abbott_2000] can be addressed through a re-conceptualisations of what it means to track practices [@Latour_2012].
5. To demonstrate reproducible social scientific research in the form of tightly integrated 'virtual machine' distribution of data, analysis, code, text and visualizations.

In the course of the project, which ran for around 14 months overall, we did not meet all of these ambitious objectives. We did however address all of these objectives in various ways, ranging from using large public API (Application Programmer Interface) datastreams, using big data analytic tools such as Google BigQuery, and working with various analytic techniques coming from social network analysis, and statistical learning. As well, we worked extensively with the reproducible research tools and platforms in order to test how and to what extent reproducible research can be conducted using tools available in the wild rather than constructed as a bespoke or purpose-built platform for data analysis. As usual, we altered and reoriented some objectives, and curtailed others in order to keep the research in movement through its different phases.

Using public datasets

The metacommunities concept, with its focus on dispersion, on multiple 'species' (this could be understood in various ways in our domain), and on potential interactions, suggested a need to work with the data in aggregate. While we ended up focusing almost all of our efforts on the data that relates to the Github software repositories, we also made some of public data dumps of the popular Q&A site StackOverflow.com to help identify important processes of interaction between developers.

We faced some fairly fundamental practical choices in working with the Github data. We could harvest data from Github's extensive APIs (see https://api.github.com/):

```
  {
"current_user_url"
"current_user_authorizations_html_url"
"authorizations_url"
"code_search_url"
"emails_url"
"emojis_url"
"events_url"
"feeds_url"
"following_url"
"gists_url"
"hub_url"
"issue_search_url"
"issues_url"
"keys_url"
"notifications_url"
"organization_repositories_url"
"organization_url"
"public_gists_url"
"rate_limit_url"
"repository_url"
"repository_search_url"
"current_user_repositories_url"
"starred_url"
"starred_gists_url"
"team_url"
```

```
  "user_url"
  "user_organizations_url"
  "user_repositories_url"
  "user_search_url"
}
```

Each item in this list designates a specific facet of the Github platform data. The `events_url` is probably the most general one since it offers almost live access to everything that people do on Github. But as this list shows, we were facing several dozen different access points to what was happening on Github. The data available through the APIs varies in volume, and rate-limits apply to many of the API points of access, but in general we were faced with an abundance of data, of varying duration. The problem was more one of being spoiled for choice.

Even given this abundance of data, several other major viable alternatives presented themselves. This may not be typical of all social media data, but in the Github case, Github API data from the `events` API has been archived in a public archive http://www.githubarchive.org by a Google employee named Ilya Grigorik for the last year or two [@Grigorik_2012]. This archive was published after we submitted our grant application, but as soon as we discovered it in early 2013, we seriously considered using it because it does not suffer from the rate-limits that Github's own APIs impose. Initially, GithubArchive only reached back to early 2012, but now it extends back to February 2011, so offers a timeline datastream of every public event on Github for the last three years. This dataset can almost be wrangled on a single machine. It totals around 179 Gb compressed, and consists of a file for every hour since early 2011. As is often the case in software cultures, the code that runs GithubArchive can itself found on Github as a `git` repository: https://github.com/igrigorik/githubarchive.org.

A quick glance at that repository shows that GithubArchive.org feeds directly into another major Github data resource, the GoogleBigQuery data analytics platform, which hosts the full GithubArchive dataset at GithubArchive.timeline. On GoogleBigQuery the formatting of the event data is quite different. The data is stored in a single huge table, with almost 200 columns and, at the time of writing, around 280 million rows. This massive table can be queried using an SQL (Structured Query Language) dialect.

While we did use the Github APIs, and wrote scripts that allows us to move back past the February 2011 limits of the GithubArchive data, having

the full `events` timeline in both the compressed forms of hourly files from GithubArchive and in the GoogleBigQuery table made any efforts to construct our own dataset from scratch seem somewhat futile. But working with either the GithubArchive or the GoogleBigQuery data brings its own problems. GithubArchive is in around 30,000 files that need to be read or loaded into memory in order to work with them. Even using GoogleCompute platform instances as we did, processing those files takes takes many hours. We did process all the files several times, in particular in pursuit of a full description of the terrain covered by the 12 million or so code repositories on Github, but the result of the processing were new database summaries of the Github data (for instance, we build a Redis database to give us quick access to the main attributes of all the repositories, and all the user attributes whenever we needed it). The problem with these full dataset traversals is that each time they are run, they produce a new summary or synoptic dataset, but a slight change or variation in the information needed for a particular analysis means running all the scripts again. This is not a good workflow.

Most of our daily analyses were constructed then using GoogleBigQuery. As mentioned above, the advantage of this platform is that it allows SQL-style queries to be run against a large, albeit not quite $N = All$ Github dataset. On the one hand, these queries can be done interactively through a website interface (especially useful for incrementally constructing queries), but on the other hand, the nature of the GithubArchive data stored on GoogleBigQuery means that queries are quite difficult to construct. They are often highly intricate, nested queries.

It should be noted too that these data analytic tools are changing rapidly. When we first started working with GoogleBigQuery, we were writing quite a lot of `Python` code to set up a query, execute and then reshape the data it returned into forms that we could work with. A typical set up query in June 2013 began something like this:

```
def query_table(query, max_rows=1000000, timeout=1.0):

    """ Returns the results of query on the githubarchive
    args:
    ----------------
    query: a BigQuery SQL query
    max_rows: maximum number of rows to return
    timeout: how long to wait for results before
    checking for actual data in the rows
```

```
    returns:
    --------------------------
    results_df: a pandas.DataFrame of the results
    """

    try:
        bigquery_service = setup_bigquery()
        job_collection = bigquery_service.jobs()

        # put limit on query if it doesn't have one
        if query.lower().find('limit') == -1:
            query = query.replace(';', ' ') + '    LIMIT %d' % max_rows + ';'

        print 'executing query:'
        print query
        query_data = {'query':query}

        query_reply = job_collection.query(projectId=PROJECT_NUMBER,
                                       body=query_data).execute()

        job_reference = query_reply['jobReference']

        while (not query_reply['jobComplete']):
            print 'Job not yet complete...'
            query_reply = job_collection.getQueryResults(
                              projectId=job_reference['projectId'],
                              jobId=job_reference['jobId'],
                              timeoutMs=timeout).execute()

        results_df = pn.DataFrame()
```

By January 2014, the query, including all the setup (not shown in the code vignette above) could be written in one line of `R` something like this:

```
query_exec(project="metacommunities",  query=sql)
```

This contrast is probably commonly experienced by researchers working with data analytics tools and commercial data sources. The churn of tools and

infrastructures can make even quite recent work, and indeed investments in infrastructure somewhat redundant. Much of the code we wrote to wrangle GithubArchive and GoogleBigQuery in the first half of the project is already somewhat beside the point in terms of ongoing use. At the time, it was the only way to work with the data, but because so many people are working on similar problems, the tools changes quickly.

	data types
boolean	7
integer	41
string	151

Table 1: Columns in GithubArchive on GoogleBigQuery

As Table 1 shows, 75% of the fields in the GoogleBigQuery dataset are string, around 20% are numerical count data, and the remainder are boolean or True/False values. But for any given event, many of these fields are likely to be empty because the GoogleBigQuery has been constructed to accommodate every conceivable event type without consideration of the distribution of events. Having all the attributes of every different event type stored in a single table, and trying to avoid moving things in and out of GoogleBigQuery too much leads to quite complicated SQL queries. The tremendous advantage of the GoogleBigQuery is freedom from the need to think very much about platform computing capacity. But adapting ourselves to the constraint to build densely nested queries that extract everything we want from a single table of GoogleBigQuery was quite hard.[1]

To both give an idea of the distribution of event types, and the relative distribution of events using GoogleBigQuery is relatively straightforward. A query to count show the frequency of events on Github is:

```
SELECT type, count(type) as event_count FROM [githubarchive:github.timeline]
group by type order by event_count desc
```

We should note in passing this is not a typical query. Most of the queries we

[1] As mentioned above, GoogleBigQuery can be accessed through a web interface, which is good for testing queries, and programmatically using scripts written in `R` or `Python`. We used scripts to run full queries. On one occasion, a `Python` script ran up a query charge of $US2400 overnight. We were shocked and dismayed at the cost. After verious discussions with the GoogleBigQuery engineering team, we received a credit for 75% of the charge. We also had quite an interesting and length discussion with Google Marketing team in California about some problems in using GoogleBigQuery. Soon after this event (October 2013), Google announced a massive reduction in the price of using GoogleCompute.

ran against GoogleBigQuery were much more complicated, as shown in the example below:

```
SELECT
    ForkTable.repository_url,
    COUNT(DISTINCT ForkTable.url) AS f2p_number,
    AVG(PARSE_UTC_USEC(PullTable.created_at)-PARSE_UTC_USEC(ForkTable.created_at))/36
FROM
    (SELECT
    url,
    repository_url,
    MIN(created_at) AS created_at
    FROM
    [githubarchive:github.timeline]
    WHERE type='ForkEvent'
    AND PARSE_UTC_USEC(created_at) >= PARSE_UTC_USEC('2012-04-01 00:00:00')
    AND PARSE_UTC_USEC(created_at) < PARSE_UTC_USEC('2011-05-01 00:00:00')
    GROUP BY
    repository_url,
    url)
AS ForkTable
INNER JOIN
    (SELECT
    repository_url,
    payload_pull_request_head_repo_html_url,
    MIN(created_at) AS created_at
    FROM
    [githubarchive:github.timeline]
    WHERE type='PullRequestEvent'
    AND PARSE_UTC_USEC(created_at) >= PARSE_UTC_USEC('2012-04-01 00:00:00')
    AND PARSE_UTC_USEC(created_at) < PARSE_UTC_USEC('2012-05-01 00:00:00')
    GROUP BY
    repository_url,
    payload_pull_request_head_repo_html_url)
AS PullTable
ON
    ForkTable.repository_url=PullTable.repository_url AND
    ForkTable.url=PullTable.payload_pull_request_head_repo_html_url
GROUP BY
ForkTable.repository_url
```

```
ORDER BY
f2p_number DESC
```

Running the simply query directly against GoogleBigQuery, the accessibility of this dataset appears in Table 2.

```
library(bigrquery)
event_query = 'SELECT type, count(type) as event_count FROM [githubarchive:github
                    group by type order by event_count desc'
event_counts = query_exec(project="metacommunities",  query=event_query)
```

```
## Auto-refreshing stale OAuth token.
```

```
event_table = xtable(event_counts[event_counts$event_count>0,], label='tab:event_
print(event_table)
```

	type	event_count
1	PushEvent	138455718
2	CreateEvent	34873055
3	WatchEvent	25654015
4	IssueCommentEvent	24669358
5	IssuesEvent	15867858
6	PullRequestEvent	11014450
7	ForkEvent	9881006
8	GistEvent	4816399
9	GollumEvent	4198827
10	DeleteEvent	3598309
11	FollowEvent	3435804
12	PullRequestReviewCommentEvent	3009258
13	CommitCommentEvent	2464118
14	MemberEvent	1478053
15	ReleaseEvent	404030
16	DownloadEvent	302247
17	PublicEvent	257691
18	TeamAddEvent	175909
19	ForkApplyEvent	5628

Table 2: Event counts for all GithubArchive data

As we will see, the complexity of practices on Github means that there are more than 20 different event types in the Github event data. On the one hand, this is really helpful and interesting in terms of our aim to analyse the diversity and commonalities of coding practice. On the other hand, it poses some real analytic problems that we struggled with throughout the project. Event types occur in a huge range of patterns. They are dominated by PushEvents, but the event total includes many other other events types, and this suggests that code repositories are much more than just repositories. They are sites of production, sociality and community, not just places where people store code.

Standing back a little from this detail, the important point here is we were easily able to meet our first objective of 'using datasets generated from publicly accessible code repositories' almost by the accident of GithubArchive and GoogleBigQuery choosing to treat Github data as something worth publishing in bulk.[2]

Database on diversity of practices

Our second objective to develop an empirically rich and reusable database of evidence describing the extent and diversity of code sharing practices across a comprehensive range of software projects. This was a slightly misconceived objective in some ways. Evidence of the diversity and commonality of code sharing practice was at the centre of our interest as we analysed what has been happening in the last few years on Github. We did struggle with the pre-2011 history of the platform in terms of data analysis. Like many social media platforms – Github is a social media platform in key respects – Github has constantly changed since its launch in late 2007. Unlike many platforms, in which working with everything that happens in even one month is difficult (for instance, all Tweets for a month), it is possible to work with whole years of Github activity at a time, especially with GoogleBigQuery. This, however, does not mean that we could easily make a database of diversity.

The problem of constructing such a database can be detected in Figure 1. Although the existence of 12-13 million repositories on Github suggests that

[2]We should point out too that another massive source of data on what is happening on Github comes from a substrate of activity accessible using the `git` version control system. The `git` data is available on every public repository, and it usually ranges back to the inception of the repository in question. In comparison to the GithubArchive data, this data is very molecular. But `git` data has to be individually harvested from each repository, and this is slow, complicated work.

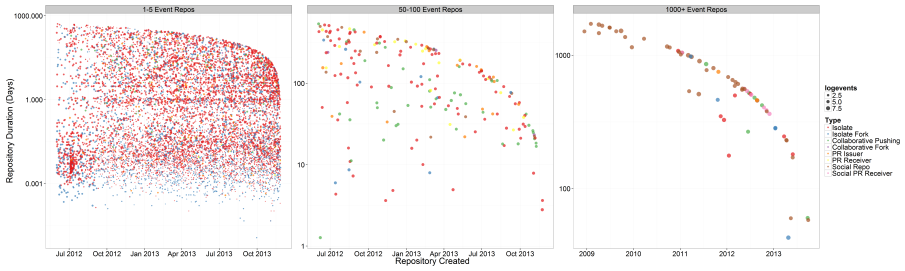


Figure 1: Github repository durations by event count

there are great variety of different things happening on Github, when you count those things, some familiar patterns begin to appear. The vast majority of repositories only survive for a few events (<5) before lapsing into stasis. At the other end, a small number of massive repositories attract hundreds of thousands of events. This distribution of event counts/repository is another instance of the infamous power law distributions of events endemic to many social processes. Faced with a power law distribution of events/repository, we could not design a database without first trying to disentangle some of the processes that gave rise to that distribution (and we discuss the problems in doing that in the next section).

```
##  [1] "early_repos_2000_only"         "all_repos_4_2"
##  [3] "repo_summary"                  "post_domains"
##  [5] "200k_active_parents_PRheads"   "all_forks_2"
##  [7] "100k_repos_most_pushes"        "100k_DeleteEvents"
##  [9] "all_repos_1_3"                 "activity"
## [11] "tags"                          "200k_active_forks_parentdata"
## [13] "all_repos_4_3"                 "actor_location"
## [15] "alphagov_forks"                "200k_active_parents_PRbase"
## [17] "actors_who_fork_a_lot"         "PR_AddedUsers_NoIntra"
## [19] "100k_pull_requests"            "all_forks_old"
```

More importantly, the point of developing an empirically rich and reusable database on code repositories has in some ways been obviated. GithubArchive and GoogleBigQuery render that objective slightly redundant. We did construct many tables derived from those datasets in the course of using GoogleBigQuery (a partial listing is shown above), but these tables are themselves intermediate representations that support high-level analysis. They can be constructed by running the queries again, and this bring more

recent events into the data. But these tables themselves do not constitute a schema that organises the $N = All$ data on Github. They are more like instruments that create views on events. That datastream itself carries much more value than any of the many intermediate tables we produced using queries.

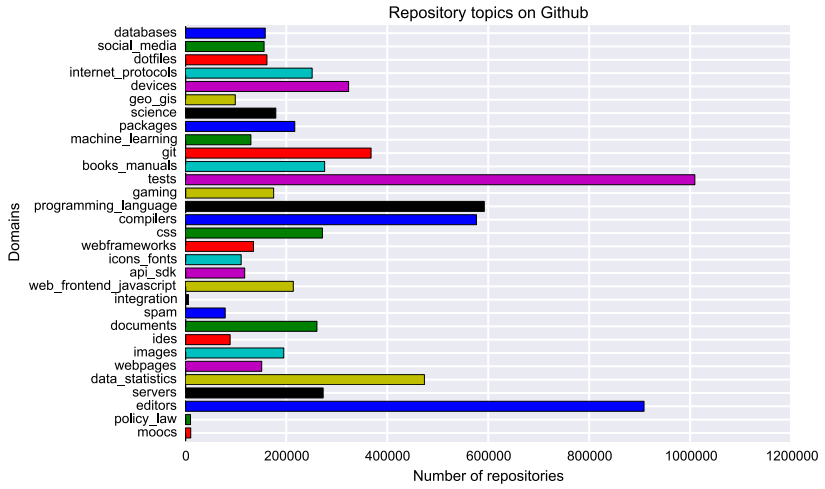
Learning and modelling diversity of practices

A key objective of our project was to explore the potential of $N = All$ data using classificatory and inferential techniques from statistical learning. The main obstacle in bring these methods to bear is the sheer diversity of practices. We tried various ways of cutting through the noise of many small repositories and a small number of very large repositories. Again, the range of activities on Github is nothing like the relative homogeneity of tweets on Twitter, or the network of relations in the Facebook Social Graph. Github is a magnet for very different software development communities, ranging from the highly commercial worlds of Javascript coders building websites or 'devops', through to very esoteric hobbyist groups working on drone software or 3D printer CAD designs. A stunning diversity of non-coding practices also cascade across Github – people using it to co-author legal documents (Whitehouse.gov) or recipes, manifestos and novels.

Much of our modelling, clustering and classificatory work focused on trying to tame this diversity, to code it in various ways, or to locate it in various domains. The idea here is that if we could manage to model, cluster or code significant regions of the sprawling terrain of Github, we could begin to cut through the power law distributions more effectively, and begin to show how they come into being. This proved challenging in key respects.

For instance, one might think it is simple matter to decide how many repositories contain code and how many repositories relate to something else based on the event timeline data. Of course, by looking directly at a code repository on Github using a web browser, you can quite quickly tell what a repository generally contains. But we couldn't look at 12 million repositories, only at 280 million events in the GithubArchive event data. Can one tell from the patterns of events concerning a particular repository whether it contains software or not? We couldn't find a way to do that.

We made various attempts to orient ourselves in the event datastream. Some of our early efforts focused on the 'topics' of repositories. We tried to construct topic models (or Latent Dirichlet Allocation document models [@Blei_2011])



for code repositories. The topic model algorithms, which are based on quite recent statistical learning research into probability distributions of words in documents, and attempt to identify mixtures of latent topics in large document collections, foundered heavily in the Github repositories. The problem is that descriptions of code repositories take many different forms. Some repositories offer very practical installation and configuration details, others report on very specific technical details (sometimes reaching down into hardware device specificities or infrastructural configurations). The texts differ greatly in length and in technical complexity. Topic models were not able to produce very useful results. We made use of some machine learning techniques such as random forest and Naive Bayes classifiers. For instance, given the prominence of *organisations* on Github (including BBC, Google, Facebook, Mozilla, Apache, etc.), we hypothesised that organisations might organise Github, or that organisation might shape what happens on the event timeline in important ways. We manually classified 1000 organisations using an organisational typology derived from recent social theory, and then used that coded dataset to classify organisations more generally on Github. But again, as in the topic models, our classifier struggled to find much signal amidst the extremely variability of organisational forms and scales on Github.

Similarly, with perhaps greater success, we worked extensively with the names of the repositories and supervised machine learning techniques to classify repositories in terms of the many separate domains in which coding work is done. Using just the names of repositories, we labelled several thousand

repositories in terms of approximately 20 different software categories (file formats, editors, database, package management, web frameworks, javascript library, compilers, geographic software, etc.). We then trained Naive Bayes classifiers on the labelled data, and used them to identify repository topics on a much larger number of repositories (see Figure for a preliminary result from this work). While this work remains incomplete, it does demonstrate the possibility of generating accounts of what is happening on Github in aggregate or at the metacommunity level. In particular, if these classifiers can be tuned so that they work with a bit more precision, we can begin to access the mixing of coding domains on Github. This was one of the primary intuitions in our 'metacommunity' concept.

Tracking practices in the field of devices

This objective was mainly focused on current debates in sociology and social science methodology. We discuss this objective at much greater length in some forthcoming publications [@Mackenzie_2015a].

Distributing reproducible data-based research

Our original plan was to bundle all of the data, scripts and data analysis into a virtual machine distribution. During the early phases of the project we were working towards this objective. As in the case of the analytical tools, the terrain shifted considerably during the course of the project. First of all, while virtual machine distributions were prominent in 2011-2012 as a way of publishing major scientific datasets and analyses, usually as a supplement to a single scientific paper. The main example is the international ENCODE consortium's 'Encyclopedia of DNA Elements' https://www.encodeproject.org/about/2012-integrative-analysis/. Virtual machine distributions might be the gold standard of reproducible scientific research, especially for large consortial projects in life or physical sciences. The problem with the distributions is that are relatively frozen in time. They bundle datasets, data processing and data analysis flows that relate to a single publication. They do not lend themselves to ongoing analysis, to augmentation by new data, or for that matter, open up entirely new lines of analysis. Practically, there are difficult to move around. The ENCODE virtual machine is 18GB. Since its initial release in early 2012, the ENCODE project has been packaged as an AWS Cloud Computing Instance.

Rather than distributing shrink-wrapped virtual machines via download or as cloud computing instance, we have accessioned our data processing, data analysis, interactive notebooks and draft publications as a Github repository https://github.com/metacommunities/metacommunities. Like many other researchers, we used a Github repository as a way to organise our own work, to keep track of different versions and code and for collaborative work (see Metacommunities. Because of their deeply versionable and highly granular structure, `git` repositories combined with the Github's accessibility, offer a way to continue to keep working on the research even after a particular release. For instance, we released a preliminary version of our research in October 2014 with a DOI (Digital Object Identifier) http://dx.doi.org/10.5281/zenodo.12651[3]. Github repositories, because they are based on the `git` version control system, offer great flexibility in offering different versions of the same research project over time. Moreover, these repositories can have a multi-branch structure. Different branches of the same repository can very easily offer different perspectives on the same project. For instance, a publication relation to the role of organisation on Github can exist as a distinct branch of the metacommunities repository. These branches can be radically different to each other, yet still draw on common scripts and datasets. This has huge advantages over the virtual machine packaging of the research, since new branches can be added as needed (for instance, in working towards a new publication), and when work is completed on that branch, it can be released and assigned a DOI so that it begins to have an independent existence.

Like many researchers in life, earth and physical sciences, we also made extensive use of the newer 'executable' scientific notebook tools such as IPython http://www.ipython.org for Python work and `knitr` for work in the statistical programming language `R` [@Xie_2013]. We experimented widely with different ways of bringing together the data processing (querying, transforming, re-coding, etc), data analysis (exploratory data analysis, clustering,

[3]The Zenodo research sharing platform allows Github repositories to be re-packaged as a single zip file download https://zenodo.org/. Github itself allows this, and indeed `git`, the underlying tool on which everything in Github depends allows cloning of repositories (e.g. `git clone https://github.com/metacommmunities/metacommunities`). Zenodo adds something important, at least for academic purposes, to Github and Git: a permanent accession number in the form of a DOI. This means that a repository can be referenced in publications. More importantly, by granting a software/data repository a DOI, it begins to function as a primary research output, not just a supplementary material. In general, data analysis has appeared as 'supplementary materials' in scientific publications.

classification, visualisation). Given that the operational endpoint for social science researchers is not the model, the dashboard or the visualisation (as it might be in a commercial or industry setting) but the publishable paper, we spent quite a lot of effort in aligning the flows of data from GithubArchive or GoogleBigQuery in the direction of written papers. The goal of the executable paper somewhat eluded us, although it is not in principle impossible with good resources. A truly executable paper would pull data from primary sources (perhaps using SQL queries on GoogleBigQuery), transform the data, generate various statistics, clusterings and visualization in a series of steps that could be followed in the electronic version of the paper itself. The code and the social science analysis co-inhabit the same document (as for instance, the code above that pulls current events counts does). We managed to do this to varying extents, but not 100% for any particular publication during the course of the funded research. We will need further time to retrospectively re-engineer some of our written outputs in order to fully realise this possibility in all its different facets. Our goal here would be produce an executable paper that stands at the endpoint of a branch of our 'metacommunities' repository, and reaches right back along the branch through all the preparatory steps that begin with the GithubArchive data.

Conclusion

We have not described anything of the substantive key findings on contemporary coding practice. The 'Metacommunities of practice in the code-sharing commons' project has made substantial progress in developing sociological accounts of software cultures as they take shape in the post-open source world. In this paper, however, we have focused on describing the design, methods and practices of our work in relation to data analysis and its associated infrastructures.

The interest, we would suggest, of cases like Github concerns their messiness and complexity. Although they bear all the marks of a typical social media platform – followers, likes, sharing, participation, etc. – they differ greatly from many other contemporary platforms in terms of their complicated event structures, their wide variations in scale, and wildly disparate patters of practice. Atypically, the data relating to such practices (at least, in so far as they are public) is 100% available, and this again contrasts markedly with many other settings where social science researchers are compelled to work on $N \neq All$ samples of the data (thus putting them in a different position to the platforms themselves). Github is one of the few cases where the full

data needs to be available for the platform to work properly.

Another interest of the Github case is the fact there is so much interest in the Github dataset. We were not alone in analysing the GithubArchive data. Various data competitions and many independent mapping efforts have worked with the data. A sub-culture of data visualization pivots on this data, with many dashboards and network visualizations spinning around it. Many of these stem from software developers who use Github and take a strong interest in what happens. Their projects and visualizations provide an extremely useful backdrop or pinboard to use as points of comparison and sometimes springboard from.

References