Proceedings Paper:
Walkinshaw, N. and Minku, L. (2018) Are 20% of files responsible for 80% of defects? In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 11-12 Oct 2018, Oulu, Finland. ACM. ISBN: 978-1-4503-5823-1.

https://doi.org/10.1145/3239235.3239244

Are 20% of Files Responsible for 80% of Defects?

Neil Walkinshaw
The University of Leicester
Leicester, UK
ndwalkinshaw@gmail.com

Leandro Minku
The University of Birmingham
Birmingham, UK
L.L.Minku@cs.bham.ac.uk

ABSTRACT

Background: Over the past two decades a mixture of anecdote from the industry and empirical studies from academia have suggested that the 80:20 rule (otherwise known as the Pareto Principle) applies to the relationship between source code files and the number of defects in the system: a small minority of files (roughly 20%) are responsible for a majority of defects (roughly 80%).

Aims: This paper aims to establish how widespread the phenomenon is by analysing 100 systems (previous studies have focussed on between one and three systems), with the goal of whether and under what circumstances this relationship does hold, and whether the key files can be readily identified from basic metrics.

Method: We devised a search criterion to identify defect fixes from commit messages and used this to analyse 100 active Github repositories, spanning a variety of languages and domains. We then studied the relationship between files, basic metrics (churn and LOC), and defect fixes.

Results: We found that the Pareto principle does hold, but only if defects that incur fixes to multiple files count as multiple defects. When we investigated multi-file fixes, we found that key files (belonging to the top 20%) are commonly fixed alongside other much less frequently-fixed files. We found LOC to be poorly correlated with defect proneness, Code Churn was a more reliable indicator, but only for extremely high values of Churn.

Conclusions: It is difficult to reliably identify the "most fixed" 20% of files from basic metrics. However, even if they could be reliably predicted, focussing on them would probably be misguided. Although fixes will naturally involve files that are often involved in other fixes too, they also tend to include other less frequently-fixed files.

KEYWORDS

Defect distribution, Pareto principle, Survey

1 INTRODUCTION

The distribution of faults within software systems has been the subject of a considerable amount of research. Previous empirical studies indicate that software defects obey the Pareto Principle – that a minority of modules or files (the top 20%) are responsible for a majority of defects (around 80%) [3, 8, 11, 16, 23]. If such a 'golden' ratio exists, it raises the prospect of the more focussed application of verification and validation techniques that might not scale to a system-level, and could support the extraction of improved training sets for defect prediction models.

There are however some limitations to the aforementioned studies that place a question-mark over this ratio. They are based upon small numbers (between one and three) of industrial closed source systems, all of which revolve around the telecoms domain. They are also based on the premise that every defect is fixed by editing a single file; fixes that span multiple files (as is typically the case) are in fact counted as multiple separate defects. This gives rise to the question of the extent to which multi-file fixes are in fact concentrated on the most defect-prone files, or whether they are more diffuse. Finally, if the Pareto relationship does exist, it is not clear how to identify the critical 20% of defect-prone files.

This paper describes an empirical study that seeks to address these weaknesses. To scale the experiment up to larger numbers of systems we use an automated approach to estimate defect-fixing changes by identifying the presence (and ensuring the absence) of certain key-words in commit messages. We use this to automatically analyse 100 GitHub repositories, selected to focus on popular, active software projects with the help of a database curated by Munaiah *et al.*[19]. The goal is to answer the following high-level research questions:

- RQ1 If we replicate previous studies (assuming one fix per defect), does the Pareto Principle apply?
- RQ2 If so, can the most defect-prone files be easily identified by established metrics?
- RQ3 If we accept that a single defect can require fixes to multiple files, are all of these fixes concentrated on the most defect-prone files?

The rest of this paper is structured as follows. Section 2 motivates this work, and introduces related work in defect prediction and the analysis of Power Laws in software engineering. Section 3 presents the methodology used for this study. This is followed by the results in Section 4, further discussion and analysis in Section 5, and threats to validity in Section 6. Finally, in Section 7 we offer some conclusions, along with our plans for future work.

2 BACKGROUND AND RELATED WORK

We start with a brief introduction to power laws and the Pareto principle. This is followed by an overview of where these phenomena have been observed within Software Engineering, and what their implications are.

2.1 Power Laws and the Pareto Principle

In complex systems it is frequently observed that certain small minorities of elements within a system are orders-of-magnitude bigger or more influential than other elements. This relationship between the number of elements and their size or influence can often be neatly characterised mathematically as a Pareto distribution, Zipf's law, or a power law [9, 22].

In mathematical terms, a quantity obeys a power law if it is drawn from the distribution $y = x^{-\alpha}$ [9]. Smaller values of x have a very high value of y, which rapidly decreases as x increases. In intuitive terms, a power law can be explained with the help of a

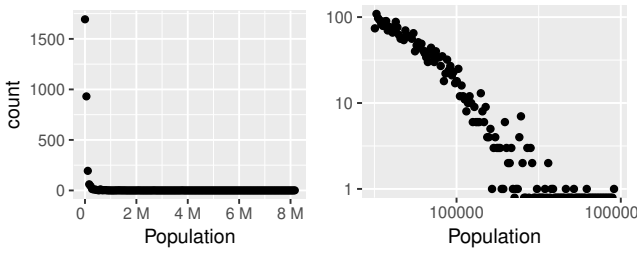

Figure 1: Histograms of U.S. city populations (for populations $> 10,000$) from 2010 census (n=2998) - data obtained from Spatial History Project [26].

popular example: the populations of cities in the US [12]. If plotted in order of magnitude, the sizes follow a curve, as shown in the left-hand plot in Figure 1; the vast majority of cities have relatively small populations (and so any "average" of city size would be unrepresentative).

One indicator that data is sampled from a power-law is to plot the data-points on a log-log scale. If one takes the logs of both sides of the power-law ($log(y) = log(x^{-\alpha}) = \alpha log(x)$), then on a log-scale this amounts to $Y = \alpha X$ - a straight line. This is what happens in the right hand plot in Figure 1 with the log-log plot of city populations.

One particularly popular characterisation of the power law is the Pareto principle – otherwise known as the 80:20 principle (the Pareto principle can be analytically derived from the power law [2, 18]). For example the biggest 20% of US cities house approximately 80% of the population (79.5% according to the 2010 census). This ratio was first suggested with respect to Italian land ownership by Vilfredo Pareto [24], who observed that 80% of the land was owned by 20% of the population.

2.2 Previous Results from Software Engineering

The power law (and Pareto principle) have predominantly cropped up in Software Engineering in two guises: in the dependencies that link software units together, and the relationship between files and defects. This subsection elaborates upon these two areas, and their respective implications for change impact analysis and defect prediction.

2.2.1 Dependencies between software units, and implications for change impact analysis. The power law, and in particular the 80:20 expression thereof, occurs frequently within Software Engineering. A raft of research [6, 18, 29] has shown that software systems tend to form "scale-free networks" [5]. If represented as a graph (where edges represent calls between functions or dependencies between classes or modules), the relationship between nodes and their in- or out-degree tends to obey a power law.

One notable property that is often associated with such scale-free networks is the fact that they obey 'small-world' characteristics [28]. In such graphs, the distance (number of edges on the shortest path) between any pair of nodes is remarkably small. This has been observed empirically for software dependencies [27].

This interconnectedness is intuitive. The various interdependencies that arise in software systems mean that the slightest change to source code can have wide-ranging ramifications. A seemingly innocuous change to a data-type or an interface can require adjustments to any files in the system that use or interact with it, and changes to these classes can propagate to other files in a similar fashion.

The task of predicting how a change might propagate through the code-base is known as Change Impact Analysis [17]. The problems posed by interdependencies (as mentioned above) is highlighted by size of the "change sets" computed by change impact analysis tools. An intuition of the problem can be found in the work by Acharya *et al.*, whose work on slice-based change impact analysis indicated that impact sets for an industrial system could routinely range from hundreds to hundreds of thousands of LOC [1].

2.2.2 Fault distributions and implications for defect prediction. The Pareto principle has also repeatedly been invoked by sources in industry and academia to characterise the effects and distribution of defects in software. In 2002 the Microsoft CEO at the time highlighted the fact that *"about 20 percent of the bugs cause 80 percent of the errors and– this is stunning to me – 1 percent of bugs caused half of all errors."* [4]. Boehm and Basili [8] suggested that *"about 90 percent of the downtime comes from, at most, 10 percent of the defects."*. They also suggested that *"about 80 percent of the defects come from 20 percent of the modules"*.

This latter suggestion that 20% of modules are responsible for 80% of defects has been corroborated by several studies. The distribution of defects was studied in 2000 by Fenton and Ohlsson [11], and was replicated in 2007 by Andersson and Runeson [3]. Both found that there appeared to be a power law relation between files and defects. This was further corrobrorated by Ostrand *et al.* [23], Kuo *et al.* [16] and Concas *et al.* [10].

There are however limitations to the aforementioned studies that could undermine this statistic. They have been restricted to a small number of systems (between one [11, 23] and three [3]), all of which are industrial systems that stem from the telecoms domain. One reason for this relatively restricted set of subjects is the effort required to manually collate the fault data from change reports and to map them to relevant code files.

There has been a large body of closely related work on the development of techniques to predict the "defect-proneness" of a file from extrinsic properties of that file [13]. These involve the development of models (often with the help of Machine Learners) that combine file properties (e.g. metrics such as LOC [23, 30] and code churn [20, 21]) to arrive at a prediction. One important factor, highlighted by Hall *et al.* in their systematic literature review [13], is training set balance. Machine Learning techniques used often fare poorly when the training set over-represents either faulty files or non-faulty files, and can produce a biased model as a result.

2.3 Motivation

The suggestion that 20% of modules are responsible for 80% of defects (or at least that there is a Pareto relationship of some sort between files and defects) has potentially significan ramifications.

If it is possible to reliably identify those specific files that are responsible for 80% of defects in a system, defect-detection efforts could be much more focussed. Expensive verification and validation techniques that might not typically be considered could become feasible if they merely had to consider a fraction of the files.

However, the empirical results that support this finding sit uneasily beside the results discussed in Section 2.2.1. These results suggest that software systems tend to be strongly interconnected, a finding that is underpinned by studies that have repeatedly illustrated how innocuous code changes can have wide-ranging, unintended consequences. The suggestion that bug fixes remain restricted to a very specific subset of the order of 20% of the files in a system appears to be incongruous.

3 EMPIRICAL STUDY

The overarching goal of this study is to shed some light on the apparent contradictions discussed in Section 2.3 – whether the vast majority of defect-fixes can be localised within a small fraction of files despite the fact that individual changes can so easily have far-reaching side-effects. We obtain our data via an automated repository-analysis of a large set of active open-source projects. We start by investigating whether we are able to replicate existing fault distribution results:

RQ1 If we replicate previous studies (assuming one fix per defect), does the Pareto Principle apply?

H1 *Given the consensus of previous studies [3, 11, 23] we hypothesise that the Pareto Principle does apply.*

RQ2 If so, can the most defect-prone files be easily identified by established metrics?

H2 Based on findings from the defect prediction literature we hypothesise that there exists a correlation between fix-frequency and LOC [23, 30] and between fix-frequency and code-churn [20, 21].

We continue by examining the make-up of a bug fix. Specifically, we seek to examine the spread of files to establish the extent to which bug fixes really are localised to a specific group of files:

RQ3 If we accept that a single defect can require fixes to multiple files, are all of these fixes concentrated on the most defect-prone files?

H3 Following on from H1 (that the buggiest files are responsible for most of the defects), we hypothesise that multi-file bug fixes tend to be concentrated on the most defect-prone files.

3.1 Subject Systems

Our goal is to base our analysis upon substantive, active projects that span a range of languages. Github does not have a reliable metric for this; Gitstars tend to include many non-software projects, or projects which happen to be popular but are not particularly substantive. As a result we start from Munaiah *et al.*'s database of GitHub projects [19], which attributes several metrics to each project, such as its maturity, the number of active developers, the use of continuous integration, as well as Gitstars etc.

We used this database to select our list of 100 projects, with the goal of focussing on those projects that were genuine, substantial, active software projects. To do this, we first of all restricted the database to those that satisfied all of the following criteria (see Munaiah *et al.* [19] for more details about the various metrics):

- Munaiah *et al.*'s Random Forest classifier (which predicts whether a project is or is not a genuine software project) should evaluate to '1' (it *is* predicted to be a software project.)
- The project should have at least one git star.
- The project should be classified as "TRUE" by Munaiah *et al.*'s "organisation" and "utility" classifiers (which respectively estimate whether the project is (1) similar to other projects developed within popular software engineering organisations and (2) of value to a wide range of developers).
- The software should have a license.
- The unit test coefficient (a value between 0 and 1 indicating the ratio of test lines of code to source lines of code) calculated in the database should be > 0.1.
- The "issues" and "community" metrics, indicating the level of project management in the system and the extent of the developer community should be > 10.

Having restricted our list to what ought to be genuine, substantive, active software projects, we then ranked the projects in order of (1) their git-star rating (as given in the database), (2) their community size and (3) their age, and selected the top 100.

Since the database was constructed in 2016, some of the projects in that list have since migrated or have become inactive (e.g. because they were usurped by more successful projects). Whenever we encountered a project that was migrated to a different (Git) repository, we used the new repository. If a project was abandoned, or was a 'metapackage' (a small project with instructions to aggregate external components) we skipped it (there were three such projects). The resulting set of projects is shown in Table 1.

3.2 Methodology

We split our presentation of methodology into data collection and data analysis. The data analysis split according to the three research questions.

3.2.1 Data Collection. The data that was used, including the list of Git URLs, commit and LOC data, has been made openly available[1].

Project properties. For each project we determined the primary programming language. This was determined by examining the most prevalent file suffixes and skimming over the source code. The languages were largely restricted to those considered by Munaiah *et al.* [19]: Java, Python, C, C++, C#, PHP, and Ruby. In the case where these languages had been used to implement a new language (Kotlin and Chapel), the developed programming language was counted as the dominant one. We also calculated the number of files and the total LOC for each project.

File selection. For each of the projects in Table 1 the GitHub repository was cloned (the hash for that version has been stored to support replication). For each project, all non-binary files were considered apart from those that would trivially change with each build (such as 'CHANGELOG' or 'NEWS'). This included source code, documentation, make-files and other build-script configurations (e.g. for Maven). This enabled us to accommodate fixes to

[1] https://doi.org/10.5281/zenodo.1253262

Table 1: Subject systems.

Name	LOC	Files	Language	Commits
active_merchant	200,951	701	Ruby	14,414
activeadmin	53,040	589	Ruby	19,890
apex-malhar	354,502	2,362	Java	105,289
azure-powershell	21,810,922	16,221	C#	1,631,205
beaker	31,666	320	Ruby	17,261
bosh	517,859	3,682	Ruby	140,199
boto3	47,915	190	Python	3,416
bourbon	4,787	154	Ruby	3,691
buck	1,156,554	11,009	Java	144,670
buildbot	275,229	1,413	Python	80,852
bundler	76,029	812	Ruby	28,679
cakephp	314,920	1,284	PHP	224,095
capybara	24,466	215	Ruby	8,756
catapult	2,841,262	8,389	Python	104,062
ceph	1,540,868	5,845	C++	311,566
chapel	5,134,878	31,447	Chapel	882,499
chef-logstash	2,700	65	Ruby	2,533
cloud_controller_ng	337,752	2,597	Ruby	51,863
cocos2d-x	1,583,122	4,552	C++	424,599
Codeception	105,038	892	PHP	38,392
coi-services	270,427	744	Python	51,184
corefx	4,090,629	18,499	C#	611,489
darktable	948,928	1,144	C	90,485
data-access	127,985	471	Java	10,879
django-allauth	54,294	518	Python	7,459
django-rest-framework	92,059	365	Python	28,281
django-tastypie	30,362	209	Python	3,834
doorkeeper	13,295	242	Ruby	6,718
draper	5,980	145	Ruby	3,422
dropwizard	88,777	1,154	Java	22,957
edx-platform	1,756,395	6,601	Python	504,692
errbit	18,142	340	Ruby	26,436
exercism.io	59,945	571	Ruby	27,414
floodlight	144,666	710	Java	22,471
fog	71,348	1,666	Ruby	178,237
fpm	28,305	126	Ruby	3,695
framework	177,552	1,263	PHP	9,242
FrameworkBenchmarks	222,580	3,748	Shell	387,197
gazebo_ros_pkgs	33,970	259	C++	4,093
geoserver	1,520,184	11,335	Java	128,502
geotools	3,666,311	15,243	Java	85,119
grape	27,242	240	Ruby	7,592
infinispan	916,583	8,087	Java	101,775
jboss-eap-quickstarts	454,127	2,353	Java	43,691
jcabi-github	67,441	479	Java	23,742
jclouds	712,032	7,357	Java	217,635
jedis	40,847	181	Java	7,283
joomla-cms	1,136,698	5,945	PHP	485,818
jruby	1,167,419	10,183	Ruby	462,961
kc	2,841,966	16,775	Java	730,182
kotlin	2,459,598	63,849	Kotlin	503,344
linguist	571,457	2,424	Ruby	43,616
logstash	131,068	1,605	Ruby	25,773
luigi	63,957	287	Python	10,141
mackup	5,442	426	Python	4,227
mail	88,534	329	Ruby	9,586
manageiq	456,865	2,546	Ruby	317,124
mantid	3,271,526	13,458	C++	2,814,986
matplotlib	871,923	2,101	Python	133,395
monolog	22,563	193	PHP	4,181
mule	556,014	4,534	Java	358,989
Mvc	424,021	2,786	C#	53,399
Nancy	123,067	1,304	C#	33,625
neo4j	1,456,046	10,019	Java	753,798
octokit.net	170,951	1,091	C#	44,605
omniauth	2,832	27	Ruby	4,913
openproject	537,705	4,340	Ruby	279,837
opsworks-cookbooks	21,672	637	Ruby	3,275
pandas	501,678	968	Python	66,033
paperclip	17,026	166	Ruby	5,069
parquet-mr	133,878	808	Java	17,456
pelican	36,406	348	Python	11,967
phinx	36,188	187	PHP	5,560
phpbb	415,902	2,383	PHP	149,367
PHPoAuthLib	29,632	228	PHP	3,523
PiplMesh	19,242	120	Python	6,297
platform	1,638,444	15,743	PHP	1,954,608
Propel2	147,603	724	PHP	25,885
puppetlabs-apache	26,555	365	Ruby	10,048
puppetlabs-firewall	15,112	106	Ruby	3,868
rails_admin	51,961	544	Ruby	26,293
react-rails	75,580	452	Ruby	5,889
repose	270,218	3,138	Groovy	189,762
resque	9,570	88	Ruby	2,893
RestSharp	23,572	168	C#	7,764
retire	18,191	143	Ruby	2,112
rubinius	1,675,723	6,633	Ruby	301,510
salt	1,570,758	4,754	Python	531,987
scribejava	15,886	250	Java	5,423
shogun	509,899	2,846	C++	118,117
sidekiq	19,278	203	Ruby	8,194
silverstripe-framework	224,678	1,670	PHP	143,831
simple_form	10,782	106	Ruby	5,218
Spout	151,722	1,186	Java	48,738
spree	128,004	2,002	Ruby	127,205
stringer	40,293	227	Ruby	3,287
swot	15,959	7,445	Ruby	46,824
Theano	328,964	673	Python	66,018
toxcore	58,045	176	C	12,020
ZfcUser	12,862	137	PHP	2,464

build configuration errors and documentation etc., which would be missed by restricting to source code alone. Accordingly, the LOC values in Table 1 can appear high (c.f. azure-powershell) because the repository can include very large text files that are used for testing purposes, etc.

File attributes. For each file the following attributes were computed:

- Lines of code
- Code churn, measured as the number of changes made to the file (specifically referred to as 'churn count' [20]).

Identifying "defect-fixing" commits. Commit messages were analysed to determine whether a commit was "defect-fixing" or not. For this we searched for messages containing the terms 'bug' and 'fix'. We excluded any commits that contained the terms 'merge', 'conflict' and 'license' or 'licence' (to avoid large numbers of commits that were fixing merge conflicts or changes to licence headers, which would routinely encompass large numbers of files).

To mitigate the risk that the expression would include commits that were not genuine bug fixes, we took a random sample of five commits for each of the projects (i.e. we manually inspected 500 of the commit messages identified by the approach). This indicated that all of the extracted commits appear to correspond to genuine bug fixes.

Nevertheless, relying on commit messages alone to identify defect fixes does come with some significant limitations that are important to bear in mind [14]. Developers can fail to explicitly mention fixes in commit messages, and single commits are not necessarily atomised (a single commit might not just fix a single bug, but might include sundry other changes). These will be discussed in more detail when we discuss threats to validity in Section 6.

3.2.2 Data Analysis. We restrict our description of the data analyses used specifically to answer the research questions. These have been subsequently explored with more targetted, exploratory analyses, which will be described in Section 4.

RQ1: Does the Pareto Principle apply to software defects? For this question we carried out two analyses. The goal of the first analysis was to determine whether or not the relationships between files and defects followed a power law. Even if the power law does not apply, it is still possible for the Pareto principle to apply - for a small proportion of files to account for a majority of the defect fixes. Thus, the second analysis aims to examine the distributions of defect fixes for each quintile of files.

For the test we adopt a procedure suggested by Clauset *et al.* [9]. We start by using a Monte Carlo simulation to estimate the parameters (x_min and α) of a hypothetical power-law distribution that should fit the given fault data[2]. For the test, the resulting distribution is then used to synthesise a large number of data points. These are then compared against the empirical data points using a Kolmogrov-Smirnoff test. We follow Clauset *et al.* in choosing a relatively conservative p-value threshold of of < 0.1 to indicate that the distribution does not follow a power-law. In other words, to identify the proportion of projects for which the distribution of fix-frequencies constitute a power-law, we count the number for which $p \geq 0.1$.

To explore the extent to which the Pareto principle applies, we calculated, for each project, the proportion of files that belong to the five quintiles (the top 20%, second 20%, etc.). These results were then summarised as a box plot with five boxes, where each box represents one of the quintiles. Each box represents the distribution of file-proportions for a given quintile. If the Pareto principle applies, we would expect the distribution of proportions in the top quintile (the top 20%) to be particularly high (around 80%), with other quintiles to be substantially lower.

[2]This was carried out using the PoweRlaw package in R: https://cran.r-project.org/web/packages/poweRlaw/index.html

RQ2: Can the most fixed files be easily identified by established metrics? The answer to this question has two parts. Firstly, we establish whether there is in principle a correlation between the number of fixes and LOC or churn. We then investigate whether the correlation is strong enough, by establishing to what extent the top 20% of files with the highest LOC or Churn overlap with the top 20% of the most fixed files.

For this question we examine, for each project, the relationship between the number of defect-fixing commits and the LOC (for RQ 2(a)), and the correlation between the number of defect-fixing commits and the code churn (for RQ 2(b)). To accommodate the skew in the distribution of defects we use the Spearman-Rank method to compute the correlation, and do so on a project-by-project basis. To summarise the correlations across all projects we apply Fisher's Z-transformation.

Having calculated the correlations on a file-by-file basis, we also establish how successful Churn and LOC are specifically for identifying the top 20% of most fixed files. For this we look at the proportion of files that belong to the top 20% of most fixed files that also belong to the top 20% of files in terms of LOC and Churn respectively. We compute this for each project and present the result as a box-plot.

Given the inconclusiveness of prior research linking LOC to number of fixes, we do not posit a hypothesis for the correlation with LOC. We do however expect a reasonably strong correlation (> 0.7) for code churn.

RQ3: Are multi-file fixes concentrated on the most defect-prone files? For each project we identified every bug-fixing commit (identified as described above). To determine the 'spread' of a commit we identified two measures: (1) the sizes (in terms of the numbers of files) of defect-fixing commits, and (2) the extent to which multi-file commits that involved files in the most defect-prone quintile also involved files in other quintiles (i.e. less defect-prone files).

The fix-sizes were summarised as a box plot with one box for each quintile. For each quintile, this will show the distribution of the number of other files that comprise fixes, which include files in that quintile.

The 'spread' of fixes involving files in the top quintile was also summarised as a box-plot, where each box represented the extent to which files within a given quintile were co-edited with files in the top quintile.

Given that fixes might involve multiple files that do not belong to the top quintile, we compute the 'true' spread of defect fixes that involve files in the top quintile. For each project, and every bug fix involving at least one file in the top quintile, we recorded all of the files involved in those fixes. This would then give us the true proportion of files that were involved in the 80% of defect fixes.

If the answer to RQ1 is yes (approximately 20% of the files are responsible for 80% of the bugs), then we would expect the fixes to be distributed in a similar manner - for bug fixes to be concentrated overwhelmingly on the to 20%. We would expect the 'true' number of files involved to be close to 20%.

4 RESULTS

In this section we present the results. These will be discussed more fully in Section 5.

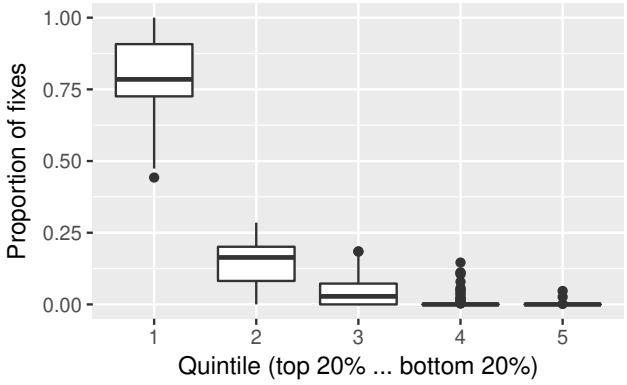

Figure 2: Proportion of fixes involving files in each quintile (where quintile 1 represents the top 20% of the files etc.).

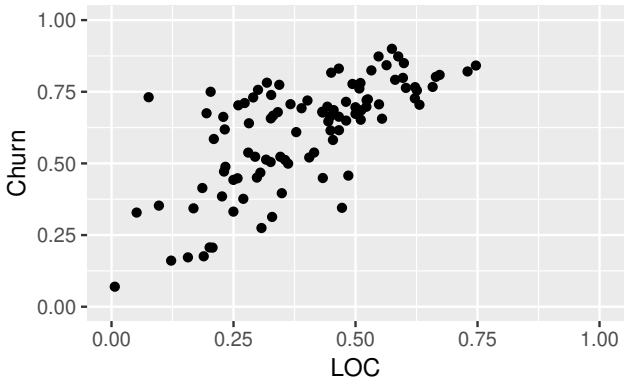

Figure 3: Churn and LOC correlations for each project.

RQ1: Does the Pareto Principle apply to software defects?

Clauset*et al.*'s power law tests produce a result of $p \geq 0.1$ for 66% of the projects. For the law to have not applied in over 34% of projects suggests that the law is far from universal.

To investigate the Pareto-principle, the box-plots for each quintile of files are shown in Figure 2. These indicate that the Pareto Principle does apply, in the sense that the top 20% of files tend to be involved in approximately 80% of fixes of bugs (mean is 80.53 %, median is 78.49%).

RQ1: The relationship between files and bug-fixes does tend to obey a Power Law (although this is far from universal). The Pareto Principle does however apply strongly; on average the top 20% of the most defect-prone files involved in 80% of defect fixes.

RQ2: Can the most frequently-fixed files be easily identified by established metrics?

Figure 3 plots the Churn and LOC correlations across all of the projects. The aggregate correlations computed through the Fisher's Z-transformation indicate that there is a weak correlation between defect-proneness and LOC (0.4), and a moderate correlation with code churn (0.6). As can be seen from the plot, however, there is a significant variance between projects.

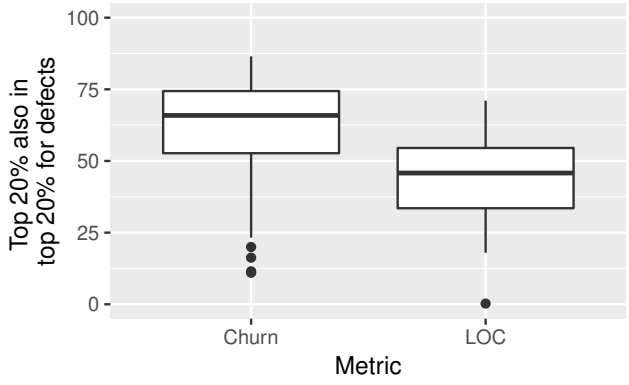

Figure 4: Proportion of fixes belonging to top quintiles that also belong to top quintile of defect-prone files.

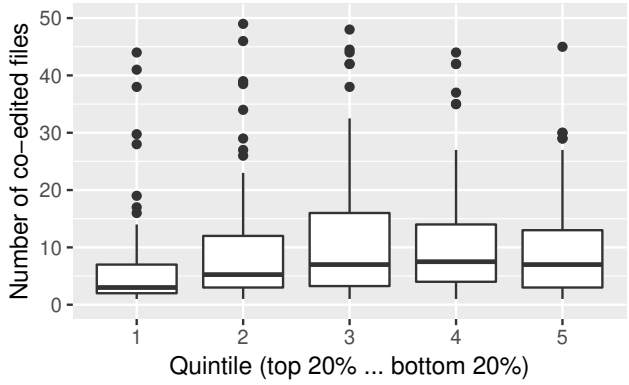

Figure 5: Fix-sizes for each quintile. A size is counted for a quintile if one of the files involved belongs to that quintile.

Figure 4 shows the proportion of the top 20% of Churn and LOC files that also belong to the top 20% of files for defects. These indicate that between 50% and 75% of the top 20% of files for Churn and less than 50% of files for LOC tend to also belong to the top 20% of defect-prone files.

RQ2: The reliability of LOC or Churn to suggest defect-proneness can vary substantially from one project to the next. In general, Churn tends to be a more reliable indicator than LOC.

RQ3: Are multi-file fixes concentrated on the most defect-prone files?

The sizes of the fixes for each quintile are shown in Figure 5. Typically, fixes tend to be relatively small (fewer than 10 files). Fixes that involve files in the top quintile tend to be significant smaller than other fixes (with a median of 3.25).

Figure 6 shows the spread of files over the quintiles for all multi-file fixes where at least one file belongs to the top (most fixed) 20%. This shows that, typically, under half of the files involved belong

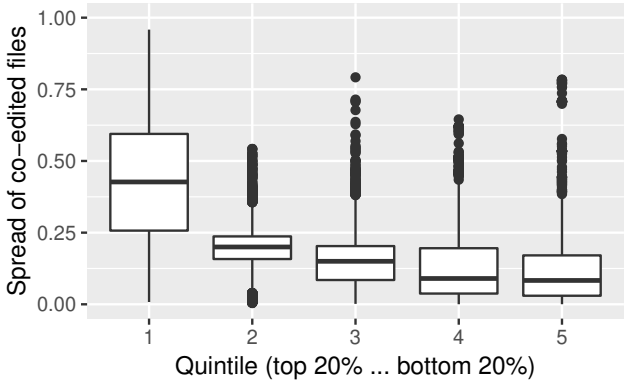

Figure 6: Spread of files co-edited with files in the top quintile of defect-prone files.

to the top 20%, and the others tend to belong to other quintiles. In other words, files that are frequently fixed tend not to be edited alongside other files that are as frequently fixed.

The complete set of files involved in fixes that involve files in the top quintile varies widely from one project to the next. Across all projects, the median proportion of files involved in 80% of bug fixes was 32%, with a lower quartile of 20% and an upper quartile of 47%.

RQ3: Most defect-fixing commits involve multiple files. For fixes involving files in the top 20%, fewer than 50% of the other files tend be in the top 20%.

5 DISCUSSION

In this section we discuss our findings and, where relevant, present additional analyses. For all of the additional analyses carried out in this section it is important to bear in mind that they are merely for the sake of exploration and corroboration, and will in most cases require more data to be conclusive.

5.1 The Role of Language and Paradigm

Our selection of projects includes software that has been written in a multitude of languages. There were 11 different principal languages in total. Given that different languages tend to imply different design paradigms and conventions, it is plausible that this could lead to different types of defects and correspondingly different types of fixes. To investigate this, we re-ran our analyses to separate out results on a per-language basis.

Although there was some variance between languages, the results did not suggest that language was a significant factor with respect to our findings for RQ1 and RQ2. The Pareto principle applied to files and defects regardless or language. Churn and LOC performed in a similar vein to all languages (it is perhaps worth noting that both metrics performed particularly well on the two systems written in C).

For RQ3 the results merit some discussion. Figure 7 shows how multi-file defect fixes are spread across the quintiles; it is equivalent to Figure 6, split up by language. Two languages to focus on are C (left-most) and C++ (third from the left). For the C projects, the co-edited files tended to be much more contained within the top 20%. For C++, there was a remarkable spread, where approximately the same proportion of files would be spread throughout all five quintiles (there was virtually no increase in the proportion of files contained in the top quintile).

Given that only two of the 100 projects were in C, and five were in C++, these differences could come down to project-specific conventions rather than language or paradigm-specific reasons. Nevertheless, this is something that we will investigate more thoroughly in future work (see Section 7).

5.2 A Link to Connectedness?

Previous studies on power laws in software systems have focussed on the interconnectivity of software elements (e.g. the dependencies that arise between software modules [18]). These have repeatedly demonstrated that a power law does appear to exist – that a small minority of the most connected modules are responsible for a majority of dependencies. In other words, there tend to be a small proportion of heavily connected "hub" modules, and a large proportion of relatively disconnected "satellite" modules.

When carrying out a code-change (such as in our case a bug fix), it is often necessary to adapt the surrounding software artefacts to accommodate the change. For example, if a change is made to the structure of a class in an Object-Oriented system, then any other classes in the system that use that class may need to be adapted to accommodate the new structure. The task of assessing this impact (as carried out by *Change Impact Analysis* techniques [17]) often involves tracing dependencies between files.

Change Impact Analysis suggests an intuitive relationship between the extent to which a file is 'linked to' by the rest of the system and the extent to which it needs to be updated when elements within the system are changed. This is corroborated by research that suggests that the 'fan-in' metric, which measures the centrality of files within a system, is governed by a power-law [18].

In our work we do not distinguish between file-changes that produce the essential fix to a defect and these adaptive changes to the surrounding system. We did also not collect metrics for files that could indicate their incoming dependencies from the rest of the system (in part because we did not have access to the necessary analysis infrastructure for all eleven programming languages used here). Nevertheless, we do posit a conjecture (which we shall explore in our future work): *The majority of files in the top 20% "most fixed" files are not especially defective, but are highly connected with the rest of the system and need to be updated frequently to accommodate fixes to genuine defects made elsewhere.*

5.3 Potential Implications for Defect Prediction

The question of how faults are spread throughout a system is related to defect prediction. We identify three areas in particular: (1) The implications for selecting data through which to train and evaluate models, (2) the question of capturing defective files that are not frequently fixed, and (3) the role of LOC and Churn.

5.3.1 Training and evaluation. Defect prediction models [13] are often trained to predict whether a file is "defect prone" by examining its dependencies (either syntactic [25] or dependencies that

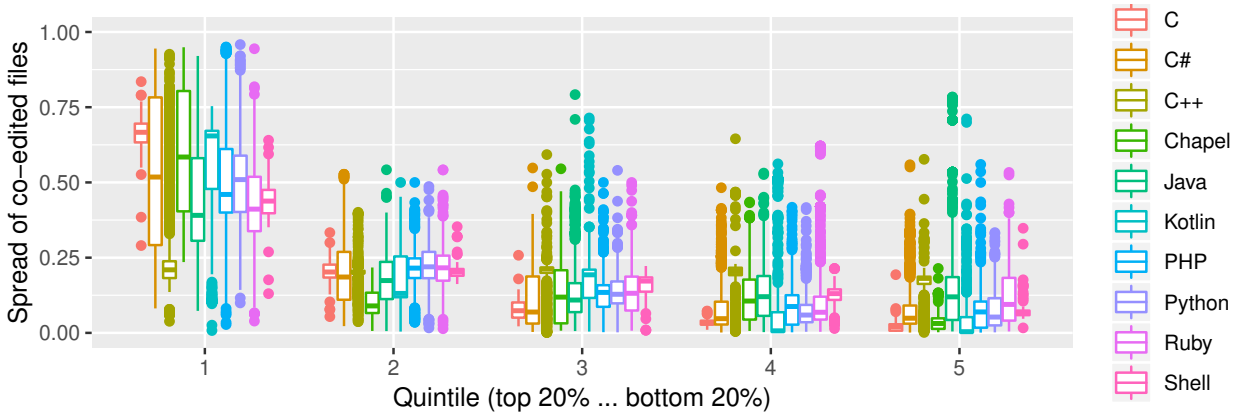

Figure 7: Spread of fixes across quintiles. The box order from left-to-right is the same as the top-to-bottom order in the legend.

involve relations between developers and code units [7]), and linking these with historical fault data. Our research (specifically the answer to RQ3) suggests, however, that defect-fixes are not particularly focussed on those files that are frequently subject to defect fixes. Instead, it suggests that many of the files involved in a fix are actually only rarely involved in fixes. We have conjectured above that those files that *are* frequently involved in fixes are involved because of their connectivity with the rest of the system, not because they are especially buggy.

If our conjecture from Section 5.2 is true (and this is what our future work will be aiming to establish), it would imply that there is a strong need to refine the data used to train and evaluate defect predictors, otherwise the accuracy of the predictor (and the reported accuracy of the technique) could be badly skewed. This is however in itself problematic because it requires extensive human intervention [15].

5.3.2 Capturing the defect. The question of whether the Pareto principle holds also has some more general potential implications for the usefulness of defect-prediction models. If only a small minority of files are genuinely responsible for a large majority of defects, then an effective defect prediction model could be a vital tool. However, our results indicate that this is not necessarily the case. Fixes tend to incorporate multiple files and tend *not* to be restricted to the top 20% of frequently fixed files (as shown in the results for RQ3).

This mixture of files within a fix (some are frequently fixed, others are not), is potentially problematic. We can expect defect-prediction models to be reasonably good at predicting the files in this 20%; for files that are frequently fixed there should be an abundance of training data. However, if our conjecture holds – that fixes to such files tend to be adaptations and that the genuine bugs happen elsewhere – then the ability to highlight faults in files that are infrequently fixed becomes particularly critical.

5.3.3 The role of Churn and LOC. In RQ2 we examined the relationship between LOC, Churn, and the fix-frequency of a file. Several studies have suggested that defect prediction models based upon LOC alone tend to fare relatively well [23, 30]. Conventionally, prediction models tend to be produced by some form of regression including other metrics such as Churn [13].

Our findings indicate that Churn is a better feature than LOC when it comes to predicting which files belong to the top 20% of defective files. Our task (of identifying the top 20% of most defective files) is different from the file-by-file defect prediction task. Nevertheless, our finding that Churn is more useful than LOC would appear to contradict findings from defect-prediction studies (as summarised by Hall *et al.* [13]), where models based on LOC have tended to outperform Churn (or 'Process-based metrics').

6 THREATS TO VALIDITY

This section describes the internal, external, and construct threats of the study.

6.1 Internal Threats

As far as *instrumentation* is concerned, there is a risk that the identification of defect fixes from commit messages is inaccurate. Genuine bug fixes may be missed out if their commit message does not satisfy our pattern, and non-fixing commits might be erroneously included if their message happens to satisfy our pattern.

We sought to attenuate the second risk (of including irrelevant commits) by checking a random sample of five messages per project (500 in total) to ensure that there were no obviously incorrect fixes included. This came after several iterations of scrutinising the returned fix commit messages to refine our search criteria to skip non-fixing commits.

It is much harder to guard against the risk of missing out relevant fixes. A degree of underreporting of fixes can be tolerated as long as the fixes that are missed follow a similar distribution to the fixes that were found. We have not observed any fixes that were missed, and thus have not observed any indicators that this should be the case.

6.2 External Threats

There is a risk that our process of selecting relevant projects from Munaiah *et al.*'s database of GitHub projects [19] biased us towards

Table 2: Proportion of text, XML, and JSON files involved in fixes, per quintile.

Quintile	1	2	3	4	5
Proportion .txt,.xml,.json	11%	12%	4%	10%	6%

a particular family of projects. Using git-stars as a primary ranking factor favours highly popular projects, which appear to favour web-development frameworks (probably because these have especially large communities of developers who rely upon them). As a result frameworks written in Ruby and and PHP are particularly prevalent. Nevertheless, the sample is sufficiently large to include a broad range of other projects, and our language-specific analysis in Section 5.1 did not indicate that language was a significant factor.

The selected projects are also all open source. It is possible that closed-source projects developed within an industrial setting could have different properties. However, there is no obvious indicator that this is the case, given that our results do not contradict the results produced by previous studies [3, 11, 23] which focussed on closed-source industrial C projects. Furthermore, several of the systems we include in our sample are developed by industry (c.f. azure powershell by Microsoft and buck by Facebook).

6.3 Construct Threats

In this study we took the decision not to focus our attention exclusively on source code files. The goal was to encompass defects that might include non-code defects as well such as configuration errors (requiring fixes to build scripts), documentation errors, or defective test data, etc. Accordingly we included every non-binary file in our analysis.

Doing so does introduce the risk that, in projects with large numbers of static non-source files, our analysis might be skewed. Table 2 shows the proportion of file extensions that were .txt, .xml and .json (the most prevalent non-source file extensions), they are even slightly more prevalent in the quintiles 1 and 2, indicating that they feature prominently in defect fixes.

There is also the possibility that, by including non source-code files, we are obscuring potentially significant relationships that might arise if we focussed entirely upon the (executable) source code. Relationships that are only weak in our analysis (e.g. between defects and LOC) could be much stronger in a more restricted scenario. This is a possibility that we intend to investigate in our future work.

Finally, there is also the risk that, by using the version history as a basis for identifying defects and their fixes, we only include those defects that have been detected (and fixed). There is a probability that there are many undetected and unfixed defects within the files. This would only skew our results if the undetected faults were distributed differently (amongst the quintiles) from the detected ones. We have tried to attenuate this risk by selecting projects that are well-established projects that are (it is hoped) less prone to extensive, potentially defect-inducing restructurings.

7 CONCLUSIONS AND FUTURE WORK

The question of whether the Pareto Principle applies to software defects ultimately depends on the definition of a "defect". If we count a fix that spans multiple files as multiple separate defects, then the principle holds; 20% of files are responsible for (almost exactly) 80% of defects.

However, our paper also shows that this definition is too simplistic. Focussing on 20% of the files only makes practical sense if all of the files required for a given fix reside within that set of files. In this paper we have shown that, for every multi-file fix that involves a file that is frequently fixed, it invariably also involves a multitude of files that are only fixed very infrequently (and are thus not part of this supposedly critical 20%).

There is an apparent contradiction between the findings from change impact analysis (that a small change can have wide-ranging impacts across the system), and fault distribution analysis which suggests that the majority of bug fixes are restricted to a small cohort of files. We conjecture that these can be reconciled by the fact that a relatively small cohort of files does in fact need to be changed frequently as part of bug fixes. However, this is not because they are especially buggy, but because they are especially well connected within the system, and need to be updated to accommodate changes to, for example, data structures or interface adaptations that are routinely carried out as part of bug fixes.

Our most pressing goal in our future work is to establish experimentally whether this conjecture is indeed true. This will require a more focussed selection of subject systems, along with a hand-curated database of defects (such as the Defects4J bug database [15]) that separate out the 'core' fixes from the adaptations within the system to accomodate these fixes. Once we have this data, we would investigate the following specific hypotheses: (1) files that belong to a fix but do *not* contain the 'core' are more likely to belong to the top quintile of fixed files, and (2) are more likely to be highly connected than files that contain the genuine defects.

There is also the question of how important the choice of language of design paradigm and the choice of file types is. Our subsequent analysis has shown that there are potentially significant differences between languages, and we have not investigated the relationships that arise if we focus entirely on source code. In our future work, we will replicate this experiment, but will focus on a larger selection of C and C++ projects (since these are particularly distinctive according to Figure 7), with the additional aim of exploring the change in relationship if we choose to focus on source code files alone.

REFERENCES

[1] Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 746–755.

[2] Lada A Adamic and Bernardo A Huberman. 2002. Zipf's law and the Internet. *Glottometrics* 3, 1 (2002), 143–150.

[3] Carina Andersson and Per Runeson. 2007. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering* 33, 5 (2007), 273–286.

[4] Steve Ballmer. 2002. Connecting with Customers. (2002).

[5] Albert-László Barabási and Eric Bonabeau. 2003. Scale-free networks. *Scientific american* 288, 5 (2003), 60–69.

[6] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. 2006. Understanding the shape of Java software. In *ACM Sigplan Notices*, Vol. 41. ACM, 397–412.

[7] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. 2009. Putting it all together: Using socio-technical networks to predict failures. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*. IEEE, 109–119.

[8] Barry Boehm and Victor R Basili. 2005. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili* 426, 37 (2005).

[9] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. 2009. Power-law distributions in empirical data. *SIAM review* 51, 4 (2009), 661–703.

[10] Giulio Concas, Michele Marchesi, Alessandro Murgia, Roberto Tonelli, and Ivana Turnu. 2011. On the distribution of bugs in the eclipse system. *IEEE Transactions on Software Engineering* 37, 6 (2011), 872–877.

[11] Norman E. Fenton and Niclas Ohlsson. 2000. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software engineering* 26, 8 (2000), 797–814.

[12] Xavier Gabaix. 1999. Zipf's law for cities: an explanation. *The Quarterly journal of economics* 114, 3 (1999), 739–767.

[13] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304.

[14] Tracy Hall, David Bowes, Gernot Liebchen, and Paul Wernick. 2010. Evaluating three approaches to extracting fault data from software change repositories. In *International Conference on Product Focused Software Process Improvement*. Springer, 107–115.

[15] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

[16] Chih-Song Kuo and Chin-Yu Huang. 2010. A study of applying the bounded Generalized Pareto distribution to the analysis of software fault distribution. In *Industrial Engineering and Engineering Management (IEEM), 2010 IEEE International Conference on*. IEEE, 611–615.

[17] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. 2013. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability* 23, 8 (2013), 613–646.

[18] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. 2008. Power laws in software. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18, 1 (2008), 2.

[19] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[20] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*. ACM, 284–292.

[21] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. 2010. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 309–318.

[22] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[23] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering* 31, 4 (2005), 340–355.

[24] V Pareto. 1896. Cours d\'economie politique. (1896).

[25] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. 2006. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 18–27.

[26] Erik Steiner and U.S. Census Bureau. 2010. Spatial History Project. Stanford University. https://github.com/cestastanford/historical-us--populations

[27] Sergi Valverde, R Ferrer Cancho, and Richard V Sole. 2002. Scale-free networks from optimal design. *EPL (Europhysics Letters)* 60, 4 (2002), 512.

[28] Xiao Fan Wang and Guanrong Chen. 2003. Complex networks: small-world, scale-free and beyond. *IEEE circuits and systems magazine* 3, 1 (2003), 6–20.

[29] Richard Wheeldon and Steve Counsell. 2003. Power law distributions in class relationships. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*. IEEE, 45–54.

[30] Yuming Zhou, Baowen Xu, and Hareton Leung. 2010. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software* 83, 4 (2010), 660–674.