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 because fixes that involve them commonly include other less frequently-fixed files.

ACM Reference Format:


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnn.nnnnnn

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.
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 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 ($\log(y) = \log(x^{-\alpha}) = a\log(x)$), then on a log-scale this amounts to $Y = aX$ - 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 the impacts of a change is known as Change Impact Analysis [17]. The problems posed by the interdependencies 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
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.

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 (files 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 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 selected above, 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. The commit message was 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. 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 targeted, 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 ($\alpha$ and $\beta$) of a hypothetical power-law distribution that should fit the given fault data. 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-Smirnov 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

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

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 defective files.

For this reason 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

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Files</th>
<th>Language</th>
<th>Commits</th>
</tr>
</thead>
<tbody>
<tr>
<td>active_merchant</td>
<td>56,951</td>
<td>701</td>
<td>Ruby</td>
<td>11,414</td>
</tr>
<tr>
<td>activeadmin</td>
<td>53,040</td>
<td>589</td>
<td>Ruby</td>
<td>19,890</td>
</tr>
<tr>
<td>apex-malhar</td>
<td>35,502</td>
<td>2,362</td>
<td>Java</td>
<td>105,289</td>
</tr>
<tr>
<td>azure-powershell</td>
<td>21,810</td>
<td>16,221</td>
<td>C#</td>
<td>1,631,205</td>
</tr>
<tr>
<td>beaker</td>
<td>31,660</td>
<td>320</td>
<td>Ruby</td>
<td>17,261</td>
</tr>
<tr>
<td>bash</td>
<td>517,859</td>
<td>3,682</td>
<td>Ruby</td>
<td>140,199</td>
</tr>
<tr>
<td>boto3</td>
<td>47,935</td>
<td>190</td>
<td>Python</td>
<td>3,416</td>
</tr>
<tr>
<td>bourbon</td>
<td>4,787</td>
<td>154</td>
<td>Ruby</td>
<td>3,691</td>
</tr>
<tr>
<td>buck</td>
<td>1,156,559</td>
<td>11,009</td>
<td>Java</td>
<td>144,670</td>
</tr>
<tr>
<td>bundot</td>
<td>289,038</td>
<td>1,413</td>
<td>Python</td>
<td>58,852</td>
</tr>
<tr>
<td>bundler</td>
<td>76,029</td>
<td>812</td>
<td>Ruby</td>
<td>28,679</td>
</tr>
<tr>
<td>cakephp</td>
<td>314,920</td>
<td>1,284</td>
<td>PHP</td>
<td>224,090</td>
</tr>
<tr>
<td>capybara</td>
<td>24,666</td>
<td>215</td>
<td>Ruby</td>
<td>8,756</td>
</tr>
<tr>
<td>catapult</td>
<td>2,841,262</td>
<td>8,389</td>
<td>Ruby</td>
<td>104,062</td>
</tr>
<tr>
<td>cepf</td>
<td>1,540,865</td>
<td>5,845</td>
<td>C#</td>
<td>131,546</td>
</tr>
<tr>
<td>chapel</td>
<td>5,134,878</td>
<td>31,447</td>
<td>Ruby</td>
<td>882,499</td>
</tr>
<tr>
<td>chef-logstash</td>
<td>2,700</td>
<td>65</td>
<td>Ruby</td>
<td>2,533</td>
</tr>
<tr>
<td>cloud_controller_ng</td>
<td>357,752</td>
<td>2,597</td>
<td>Ruby</td>
<td>51,863</td>
</tr>
<tr>
<td>cocoad-x</td>
<td>1,583,122</td>
<td>4,552</td>
<td>C++</td>
<td>424,599</td>
</tr>
<tr>
<td>Codeception</td>
<td>105,038</td>
<td>892</td>
<td>PHP</td>
<td>38,392</td>
</tr>
<tr>
<td>coi-services</td>
<td>270,472</td>
<td>744</td>
<td>Python</td>
<td>51,184</td>
</tr>
<tr>
<td>corefx</td>
<td>4,090,629</td>
<td>18,499</td>
<td>C#</td>
<td>611,489</td>
</tr>
<tr>
<td>darktable</td>
<td>948,928</td>
<td>1,144</td>
<td>C</td>
<td>90,485</td>
</tr>
<tr>
<td>django-ai</td>
<td>127,965</td>
<td>471</td>
<td>Java</td>
<td>10,879</td>
</tr>
<tr>
<td>django-ast</td>
<td>30,362</td>
<td>209</td>
<td>Python</td>
<td>3,834</td>
</tr>
<tr>
<td>doorkeeper</td>
<td>13,295</td>
<td>242</td>
<td>Ruby</td>
<td>6,718</td>
</tr>
<tr>
<td>draper</td>
<td>5,980</td>
<td>145</td>
<td>Ruby</td>
<td>3,422</td>
</tr>
<tr>
<td>dropwizard</td>
<td>88,777</td>
<td>1,154</td>
<td>Java</td>
<td>22,957</td>
</tr>
<tr>
<td>edx-platform</td>
<td>1,756,395</td>
<td>6,601</td>
<td>Python</td>
<td>504,692</td>
</tr>
<tr>
<td>erbbit</td>
<td>18,142</td>
<td>340</td>
<td>Ruby</td>
<td>26,937</td>
</tr>
<tr>
<td>exercism.io</td>
<td>59,945</td>
<td>571</td>
<td>Ruby</td>
<td>27,414</td>
</tr>
<tr>
<td>floodlight</td>
<td>144,666</td>
<td>710</td>
<td>Java</td>
<td>22,471</td>
</tr>
<tr>
<td>fog</td>
<td>71,348</td>
<td>1,666</td>
<td>Ruby</td>
<td>178,237</td>
</tr>
<tr>
<td>fpm</td>
<td>28,305</td>
<td>126</td>
<td>Ruby</td>
<td>9,695</td>
</tr>
<tr>
<td>FrameworkBenchmarks</td>
<td>222,580</td>
<td>3,748</td>
<td>Shell</td>
<td>387,197</td>
</tr>
<tr>
<td>golang</td>
<td>33,970</td>
<td>258</td>
<td>C++</td>
<td>4,095</td>
</tr>
<tr>
<td>goserver</td>
<td>1,520,184</td>
<td>11,335</td>
<td>Java</td>
<td>128,502</td>
</tr>
<tr>
<td>gocarts</td>
<td>3,666,311</td>
<td>15,243</td>
<td>Java</td>
<td>85,119</td>
</tr>
<tr>
<td>grape</td>
<td>27,242</td>
<td>240</td>
<td>Ruby</td>
<td>7,592</td>
</tr>
<tr>
<td>finnishpan</td>
<td>916,583</td>
<td>8,087</td>
<td>Java</td>
<td>201,770</td>
</tr>
<tr>
<td>jboss-eap-quickstarts</td>
<td>454,127</td>
<td>2,353</td>
<td>Java</td>
<td>43,691</td>
</tr>
<tr>
<td>jclint</td>
<td>46,471</td>
<td>449</td>
<td>Java</td>
<td>23,742</td>
</tr>
<tr>
<td>jclouds</td>
<td>712,032</td>
<td>7,357</td>
<td>Java</td>
<td>217,635</td>
</tr>
<tr>
<td>jedit</td>
<td>40,847</td>
<td>181</td>
<td>Java</td>
<td>7,283</td>
</tr>
<tr>
<td>joomla-cms</td>
<td>1,136,698</td>
<td>9,545</td>
<td>PHP</td>
<td>485,818</td>
</tr>
<tr>
<td>jruby</td>
<td>1,167,419</td>
<td>10,183</td>
<td>Ruby</td>
<td>462,961</td>
</tr>
<tr>
<td>kc</td>
<td>2,841,966</td>
<td>16,775</td>
<td>Java</td>
<td>730,182</td>
</tr>
</tbody>
</table>

Table 1: subject systems.
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 top 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.

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%).
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 to 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 the results indicated 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 readily be due 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.
The question of how faults are spread throughout a system of capturing defective files that are not frequently fixed, and areas in particular: (1) The implications for selecting data changes to the surrounding system. We did also not collect 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 strongly 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 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 our method of identifying 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 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) 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


