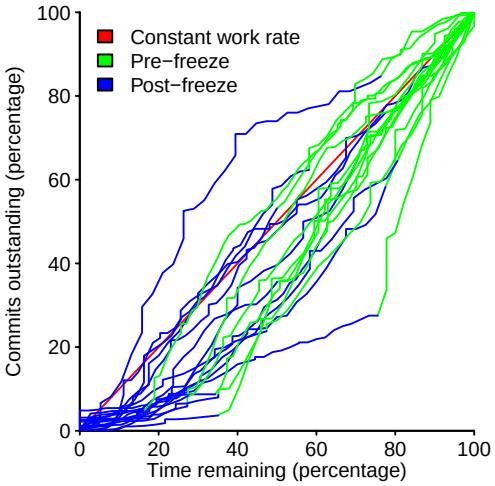


Figure 5.50: Percentage of commits outstanding against percentage the time remaining before deployment, for 18 releases; blue/green transition is the feature freeze date, red line shows a constant commit rate. Data kindly provided by Laukkanen.[1095] Github–Local

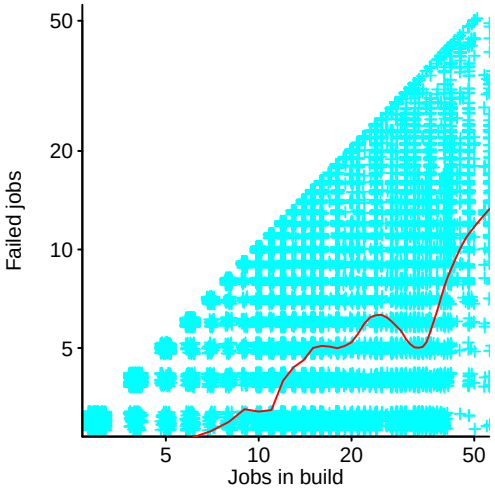


Figure 5.51: Number of failed jobs in Travis CI builds involving a given number of jobs (points have been jittered); line is a loess fit. Data from Gallaba et al.[644] Github–Local

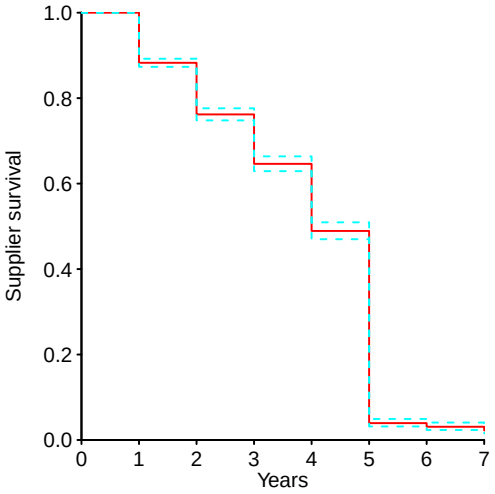


Figure 5.52: Survival curve of IT outsourcing suppliers continuing to work for 2,382 Credit Unions. Data kindly provided by Peukert.[1476] Github–Local

Traditionally releases have been either time-based (i.e., specified as occurring on a given date), or feature based, i.e., when a specific set of features has been implemented. Iterative development can enable faster, even continuous, releases (in a rapidly changing market the ability to respond quickly to customer feedback is a competitive advantage issues). Decreasing the friction experienced by a customer, in updating to a new release, increases the likelihood that the release is adopted.

One approach to working towards a release at a future date, is to pause work on new features sometime prior to the release, switching development effort to creating a system that is usable by customers; the term *code freeze* is sometimes used to describe this second phase of development.

A study by Laukkanen, Paasivaara, Itkonen, Lassenius and Arvonen[1095] investigated the number of commits prior to 19 releases of four components of a software system at Ericsson; the length of the code freeze phase varied between releases. Figure 5.50 shows the percentage of commits not yet completed against percentage of time remaining before deployment, for 18 releases; red line shows a constant commit rate, green lines are pre-freeze date, blue lines post-freeze. See Github–projects/laukkanen2017/laukkanen2017.R for component break-down, and regression models.

A basic prerequisite for making a new release is being able to build the software, e.g., being able to compile and link the source to create an executable. Modifications to the source may have introduced mistakes that prevent an executable being built. One study[1852] of 219,395 snapshots (after each commit) of 100 Java systems on Github found that 38% of snapshots could be compiled. Being able to build a snapshot is of interest to researchers investigating the evolution of a system, but may not be of interest to others.

Continuous integration is the name given to the development process where checks are made after every commit to ensure that the system is buildable (regression tests may also be run).

A study by Zhao, Serebrenik, Zhou, Filkov and Vasilescu[2016] investigated the impact of switching project development to use continuous integration, e.g., Travis CI. The regression models built showed that, after the switch, the measured quantity that changed was the rate of increase of monthly merged commits (these slowed considerably, but there was little change in the rate of non-merged commits); see Github–projects/ASE2017.R.

A study by Gallaba, Macho, Pinzger and McIntosh[644] investigated Travis CI logs from 123,149 builds, from 1,276 open source projects; 12% of passing builds contained an actively ignored failure. Figure 5.51 shows the number of failed jobs in each build involving a given number of jobs; line is a loess regression fit.

Building software on platforms other than the one on which it is currently being developed can require a lot of work. One approach, intended to do away with platform specific issues, is virtualization, e.g., Docker containers. One study,[364] from late 2016, was not able to build 34% of a sample of 560 Docker containers available on Github.

While some bespoke software is targeted at particular computing hardware, long-lasting software may be deployed to a variety of different platforms. The cost/benefit analysis of investing in reducing the cost of porting to a different platform (i.e., reducing the switching cost) requires an estimate of the likelihood that this event will occur.

For systems supporting many build-time configurations options, it may be more cost effective[766] to concentrate on the popular option combinations, and wait until problems with other configurations are reported (rather than invest resources checking many options that will never be used; various configuration sampling algorithms are available[1254]).

A study by Peukert[1476] investigated the switching costs of outsourced IT systems, as experienced by U.S. Credit Unions. Figure 5.52 shows the survival curve of IT outsourcing suppliers employed by 2,382 Credit Unions, over the period 2000 to 2010.

5.5 Development teams

A project needs people in possession of a minimum set of skills (i.e., they need to be capable of doing the technical work), and for individuals to be able to effectively work together as a team. Team members may have to manage a portfolio of project activities.

People working together need to maintain a shared mental model. Manned space exploration is one source for evidence-based studies of team cognition.[471]

Team members might be drawn from existing in-house staff, developers hired for the duration of the project (e.g., contractors), and casual contributors (common for open source projects[95]). Some geographical locations are home to a concentration of expertise in particular application domains, e.g., Cambridge, MA for drug discovery. The O-ring theory[1043] offers an analysis of the benefits, to employers and employees in the same business, of clustering together in a specific location.[625]

If a group of potential team members is available for selection, along with their respective scores in a performance test, it may be possible to select an optimal team based on selecting individuals, but selection of an optimal team is not always guaranteed.[1022] Personnel economics[1099] (i.e., how much people are paid) can also be an important factor in who might be available as a team member.

Developers can choose what company to work for, have some say in what part of a company they work in, and may have control over whether to work with people on the implementation of particular product features.

A study by Bao, Xing, Xia, Lo and Li[131] investigated software development staff turnover within two large companies. Figure 5.53 shows the mean number of hours worked per month, plus standard deviation, by staff on two projects (of 1,657 and 834 people). A regression model containing the monthly hours worked by individuals fits around 10% of the variance present in the data.

When a project uses multiple programming languages, a team either needs to include developers familiar with multiple languages, or include additional developers to handle specific languages. Figure 5.54 shows the number of different languages used in a sample of 100,000 GitHub projects (`make` was not counted as a language).

What is the distribution of team size for common tasks undertaken on a development project?

Figure 5.55 shows the number of tasks that involved a given number of developers, for tasks usually requiring roughly an hour or two of an individual's time. The SiP tasks are from one company's long-running commercial projects, while the other tasks are aggregated from three companies following the Team Software Process.

Microsoft's postmortem analysis[907] of the development of what was intended to be Windows office, but became a Windows word processor, illustrates the fits and starts of project development. Figure 5.56 shows the number of days remaining before the planned ship date, for each of the 63 months since the start of the project, against number of full time engineers. The first 12 months were consumed by discussion, with no engineers developing software. See figure 3.17 for a more consistent approach to project staffing.

What is a cost effective way of organizing and staffing a software project team?

There have been few experimental comparisons[1965] of the many project development techniques proposed over the years.

The early software developers were irregulars, in that through trial and error each found a development technique that worked for them. Once software development became a recurring activity within the corporate environment, corporate management techniques were created to try to control the process.[1042]

Drawing a parallel with the methods of production used in manufacturing, the factory concept has been adapted for software projects[425] by several large companies.[x] The claimed advantages of this approach are the same as those it provides to traditional manufacturers, e.g., control of the production process and reduction in the need for highly skilled employees.

The chief programmer team[122] approach to system development was originally intended for environments where many of the available programmers are inexperienced (a common situation in a rapidly growing field); an experienced developer is appointed as the chief programmer, who is responsible for doing detailed development, and allocating the tasks requiring less skill to others. This form of team organization dates from the late 1960s, when programming involved a lot of clerical activity, and in its original formulation emphasis is placed on delegating this clerical activity. Had it been successful, this approach could also be applied to reduce costs in environments where experienced programmers are available (by hiring cheaper, inexperienced programmers).

[x] It was particular popular in Japan.[426] Your author has not been able to locate any data on companies recently using the factory concept to produce software.

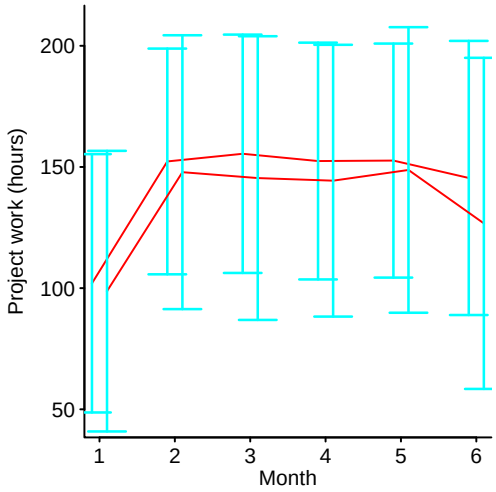


Figure 5.53: Average number of hours worked per month (by an individual), with standard deviation, for two projects staffed by 1,657 and 834 people; two red lines and corresponding error bars offset either side of month value. Data kindly provided by Bao.[131] Github–Local

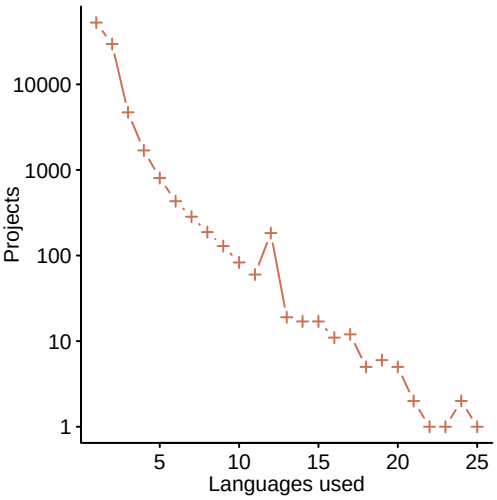


Figure 5.54: Number of projects making use of a given number of different languages in a sample of 100,000 GitHub project. Data kindly provided by Bissyande.[203] Github–Local

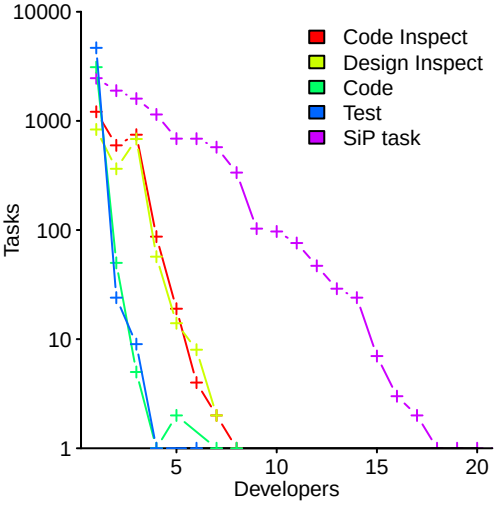


Figure 5.55: Number of tasks worked on by a given number of developers. Data from Nichols et al[1378] and Jones et al.[941] Github–Local

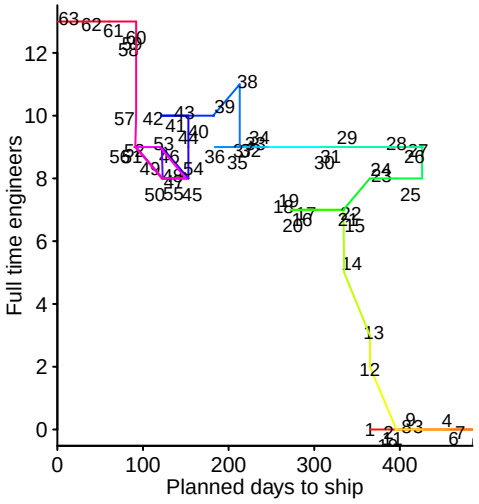


Figure 5.56: Number of days before planned product ship date, against number of full time engineers, for each of the 63 months since the project started (numbers show months since project started). Data from Jackson.[907] Github–Local

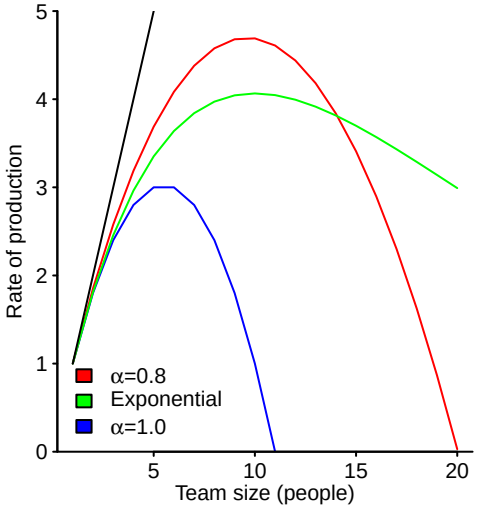


Figure 5.57: Effective rate of production of a team containing a given number of people, with communication overhead $t_0 = t_1 = 0.1$, and various distributions of percentage communication time; black line is zero communications overhead. Github–Local

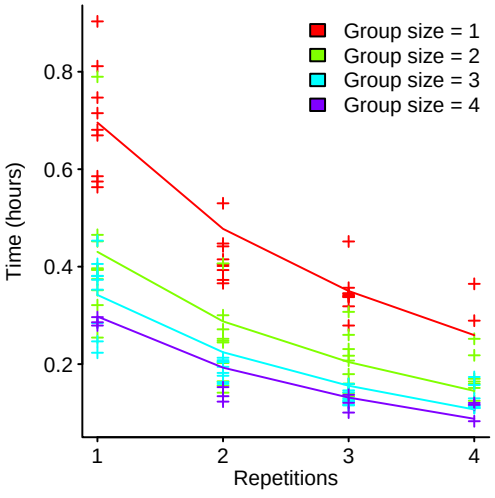


Figure 5.58: Time taken by groups of different sizes to manually assembly a product, over multiple trials; lines are fitted regression models of the form: $Time \propto \frac{0.5-0.2\log(Repetitions)}{Group\_size} - 0.1\log(Repetitions)$. Data kindly provided by Peltokorpi et al.[1464] Github–Local

Once a system has been delivered and continues to be maintained, developers are needed to fix reported problems, and to provide support.[1803] Once initial development is complete, management need to motivate some of those involved to continue to be available to work on the software they are familiar with; adding new features provides a hedonic incentive for existing developers to maintain a connection with the project, and potential new development work is an enticement for potential new team members.

A study by Buettner[275] investigated large software intensive systems, and included an analysis of various staffing related issues, such as: staffing level over time (fig 14.4) and staff work patterns (fig 11.61).

If team member activities are divided into communicating and non-communication (e.g., producing code), how large can a team become before communication activities cause total team output to decline when another person is added?

Assuming that communications overhead,[xi] for each team member, is: $t_0(D^\alpha - 1)$, where t_0 is the percentage of one person's time spent communicating in a two-person team, D the number of people in the team and α a constant greater than zero. The peak team size, before adding people starts reducing total output, is given by:[1815]

$$D_{peak} = \left[\frac{1+t_0}{(1+\alpha)t_0}\right]^{\frac{1}{\alpha}}$$

If $\alpha = 1$ (i.e., everybody on the project incurs the same communications overhead), then $D_{peak} = \frac{1+t_0}{2t_0}$, which for small t_0 is: $D_{peak} \approx \frac{1}{2t_0}$. For example, if team members spend 10% of their time communicating with every other team member: $D_{peak} = \frac{1+0.1}{2\times 0.1} \approx 5$.

In this team of five, 50% of each person's time is spent communicating.

If $\alpha = 0.8$, then: $D_{peak} = \left[\frac{1+0.1}{(1+0.8)\times 0.1}\right]^{\frac{1}{0.8}} \approx 10$.

Figure 5.57 shows the effective rate of production (i.e., sum of the non-communications work of all team members) of a team of a given size, whose culture has a particular form of communication overhead.

If people spend most of their time communicating with a few people and very little with the other team members, the distribution of communication time may have an exponential distribution; the (normalised) communications overhead is: $1 - e^{-(D-1)t_1}$, where t_1 is a constant found by fitting data from the two-person team (before any more people are added to the team).

Peak team size is now:

$$D_{peak} = \frac{1}{t_1}, \text{ and, if } t_1 = 0.1, \text{ then: } D_{peak} = \frac{1}{0.1} = 10.$$

In this team of ten, 63% of each person's time is spent communicating (team size can be bigger, but each member will spend more time communicating compared to the linear overhead case).

A study by Peltokorpi and Niemi[1464] investigated the impact of learning and team size on the manual construction of a customised product. Figure 5.58 shows how the time taken to manually assemble the product, for groups containing various numbers of members, decreases with practice. Group learning is discussed in section 3.4.5.

5.5.1 New staff

New people may join a project as part of planned growth, the need to handle new work, existing people leaving, management wanting to reduce the dependency on a few critical people, and many other reasons.

Training people (e.g., developers, documentation writers) who are new to a project reduces the amount of effort available for building the system in the short term. Training is an investment in people, whose benefit is the post-training productivity these people bring to a project.

Brooks' Law[266] says: "Adding manpower to a late software project makes it later", but does not say anything about the impact of not adding manpower to a late project. Under what conditions does adding a person to a project cause it to be delayed?

[xi] This analysis is not backed-up by any data.

If we assume a new person diverts, from the project they join, a total effort, T_e, in training and that after D_t units of time the trained person contributes E_n effort per unit time until the project deadline; unless the following inequality holds, training a new person results in the project being delayed (in practice a new person's effort contribution ramps up from zero, starting during the training period):

$E_{a1}D_r < (E_{a1}D_t - T_e) + (E_{a2} + E_n)(D_r - D_t)$, where E_{a1} is the total daily effort produced by the team before the addition of a new person, E_{a2} the total daily effort produced by the original team after the addition, and D_r is the number of units of time between the start of training, and the delivery date/time.

Adding a person to a team can both reduce the productivity of the original team (e.g., by increasing the inter-person communication overhead) and increase their productivity (e.g., by providing a skill that enables the whole to be greater than the sum of its parts). Assuming that $E_{a2} = cE_{a1}$, the equation simplifies to: $T_e < (D_r - D_t)(E_n - (1-c)E_{a1})$. If a potential new project member requires an initial investment greater than this value, having them join the team will cause the project deadline to slip.

The effort, T_e, that has to be invested in training a new project member will depend on their existing level of expertise with the application domain, tools being used, coding skills, etc (pretty much everything was new, back in the day, for the project analysed by Brooks, so T_e was probably very high). There is also the important ability, or lack of, to pick things up quickly, i.e., their learning rate.

How often do new staff encounter tasks they have not previously performed?

A study by Jones and Borgatti[940] analysed the tags (having the form @word) used to classify each task in the Renzo Pomodoro dataset; all @words were selected by one person, for the tasks they planned to do each day. Figure 5.59 shows the time-line of the use of @words, with the y-axis ordered by date of first usage. The white lines are three fitted regression models, each over a range of days; the first and last lines have the form: $at\_num = a + b(1 - e^{c \times days})$, and the middle line has the form: $at\_num = a \times days$; with a, b, and c constants fitted by the regression modeling process.

The decreasing rate of new @words, over two periods (of several years), shows how a worker within a company experienced a declining rate of new tasks, the longer the time spent within a particular role.

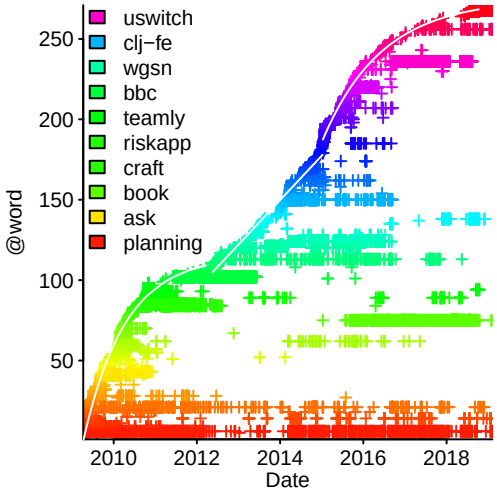


Figure 5.59: Time-line of first @word usage, ordered on y-axis by date of first appearance; legend shows @words with more than 500 occurrences. Data from Jones et al.[940] Github–Local

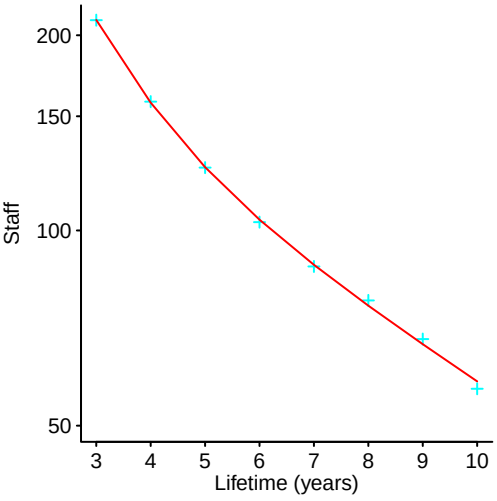


Figure 5.60: Average number of staff required to support renewal of code having a given average lifetime (green); blue/red lines show fitted biexponential regression model. Data extracted from Elliott.[540] Github–Local

5.5.2 Ongoing staffing

Software systems that continue to be used may continue to be supported by software developers, e.g., reported faults addressed and/or new features added.

A study by Elliott[540] investigated the staffing levels for large commercial systems (2 MLOC) over the lifetime of the code (defined as the time between a system containing the code first going live and the same system being replaced; *renewal* was the terminology used by the author); the study did not look at staffing for the writing of new systems or maintenance of existing systems. Figure 5.60 shows the average number of staff needed for renewal of code having a given average lifetime, along with a biexponential regression fit.

A study by Dekleva[481] investigated the average monthly maintenance effort (in hours) spent on products developed using traditional and modern methods (from a 1992 perspective). Figure 5.61 shows the age of systems, and the corresponding time spent on monthly maintenance.

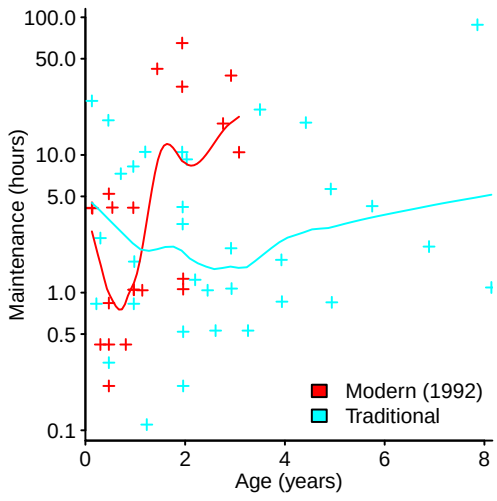


Figure 5.61: Age of systems, developed using one of two methodologies, and corresponding monthly maintenance effort, lines are loess regression fits. Data extracted from Dekleva.[481] Github–Local

5.6 Post-delivery updates

Once operational, software systems are subject to the economics of do nothing, update or replace. The, so-called, maintenance of a software system is a (potentially long term) project in its own right.

Software is maintained in response to customer demand, and changes are motivated by this demand, e.g., very large systems in a relatively stable market[1302] are likely to have a different change profile than smaller systems sold into a rapidly changing market. Reasons motivating vendors to continue to invest in developing commercial products are discussed in chapter 4; also see Github–ecosystems/maint-dev-ratio.R.

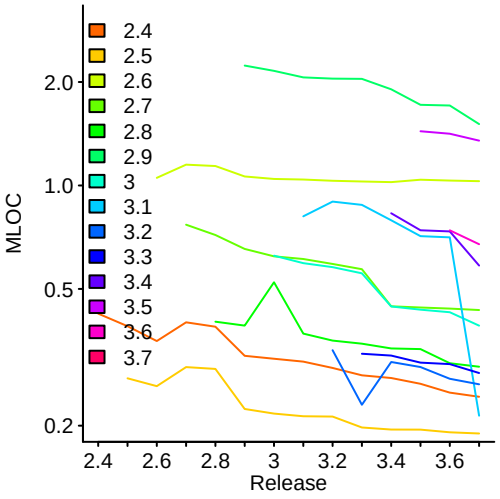


Figure 5.62: Number of lines of code in a release (x-axis) originally added in a given release (colored lines). Data kindly provided by Ozment.[1434] Github–Local

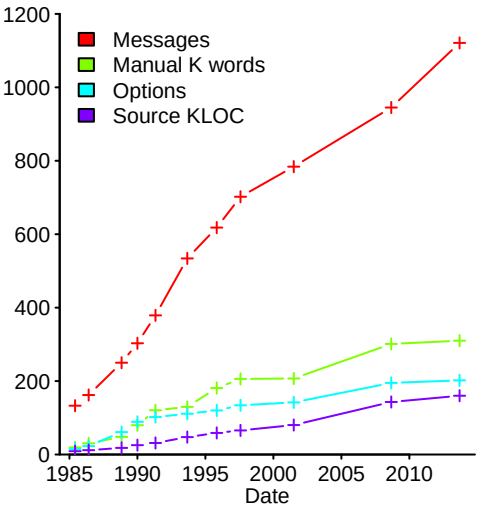


Figure 5.63: Growth of PC-Lint, over 11 major releases in 28 years, of messages supported, command line options, kilo-words in product manual, and thousands of lines of code in the product. Data kindly provided by Gimpel.[682] Github–Local

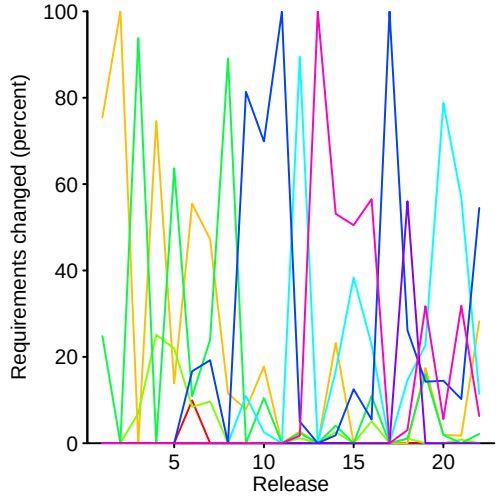


Figure 5.64: Percentage of requirements added/deleted/-modified for eight features (colored lines) of a product over 22 releases. Data extracted from Felici.[591] Github–Local

Work on software written for academic research projects often stops once project funding drys up. A study[867] of 214 packages associated with papers published between 2001-2015, in the journal Molecular Ecology Resources, found that 73% had not been updated since publication.

Updating software used in safety-critical systems is a non-trivial process;[451] a change impact analysis needs to be made during maintenance, and formal update processes followed.

Buildings are sometimes held up as exemplars of creative items that experience long periods of productive use with little change. In practice buildings that are used, like software, often undergo many changes,[244] and the rearchitecting can be as ugly as that of some software systems. Brand[244] introduced the concept of *shearing layers* to refer to the way buildings contain multiple layers of change (Lim[1143] categorised the RALIC requirements into five layers).

A new release contains code not included in previous releases, and some of the code contained in earlier releases may not be included.

A study by Ozment and Schechter[1434] investigated security vulnerabilities in 15 successive versions of OpenBSD, starting in May 1998. The source added and removed in each release was traced. Figure 5.62 shows the number of lines in each release (x-axis) that were originally added in a given release (colored lines).

Existing customers are the target market for product updates, and vendors try to keep them happy (to increase the likelihood they will pay for upgrades). Removing, or significantly altering, the behavior of a widely used product feature has the potential to upset many customers (who might choose to stay with the current version that has the behavior they desire). The difficulty of obtaining accurate information on customer interaction with a product incentivizes vendors to play safe, e.g., existing features are rarely removed or significantly changed. If features are added, but rarely removed, a product will grow over time.

Figure 5.63 shows the growth of lines of code, command line options, words in the manual and messages supported by PC-Lint (a C/C++ static analysis tool), in the 11 major releases over 28 years.

Companies in the business of providing software systems may be able to charge a monthly fee for support (e.g., fixing problems), or be willing to be paid on an ad-hoc basis.[1784]

An organization using in-house bespoke software may want the software to be adapted, as the world in which it is used changes. The company that has been maintaining software for a customer is in the best position to estimate actual costs, and the price the client is likely to be willing to continue paying for maintenance; see fig 3.23. Without detailed maintenance cost information, another company bidding to take over maintenance of an existing system[198] has to factor in an unknown risk; in some cases the client may be willing to underwrite their risk.

A study by Felici[591] analysed the evolution of requirements, over 22 releases, for eight features contained in the software of a safety-critical avionics system. Figure 5.64 shows the requirements for some features completely changing between releases, while the requirements for other features were unchanged over many releases.

A study by Hatton[786] investigated 1,294 distinct maintenance tasks. For each task, developers estimated the percentage time expected to be spent on adaptive, corrective and perfective activities, this was recorded, along with the actual percentage. Figure 5.65 shows a ternary plot of accumulated (indicated by circle size) estimated and actual percentage time breakdown for all tasks.

With open source projects, changes may be submitted by non-core developers, who do not have access rights to change the source tree.

A study by Baysal, Kononenko, Holmes and Godfrey[154] tracked the progress of 34,535 patches submitted through the WebKit and Mozilla Firefox code review process, between April 2011 and December 2012. Figure 5.66 shows the percentage of patches (as a percentage of submitted patches) being moved between various code review states in WebKit.

Source code is changed via *patches* to existing code. In the case of the Linux kernel submitted patches first have to pass a review process; patches that pass review then have

to be accepted by the maintainer of the appropriate subsystem, these maintainers submit patches they consider worth including in the official kernel to Linus Torvalds (who maintains the official version).

A study by Jiang, Adams and German[918] investigated attributes of the patch submission process, such as the time between submission and acceptance (around 30% of the patches that make it through review are accepted into the kernel); the data includes the 81,000+ patches to the kernel source, between 2005 and 2012. Figure 5.67 shows a kernel density plot of the interval between a patch passing review and being accepted by the appropriate subsystem maintainer, and the interval between a maintainer pushing a patch and it being accepted by Torvalds. Maintainers immediately accept half of patches that pass review (Torvalds 6%). The kernel is on roughly an 80 day (sd 12 days) release cycle; the rate at which Torvalds accepts patches steadily increases, before plummeting at the end of the cycle.

Open source projects may be forked, that is a group of developers may decide to take a copy of the code, to work on it as an independent project; see section 4.2.1. Features added to a fork or fixed coding mistakes may be of use to the original project. A study by Zhou, Vasilescu and Kästner[2024] investigated factors that influence whether a pull request submitted to the parent project, by a forked project, are accepted. The 10 variables in the fitted model explained around 25% of the variance[xii]; see Github–economics/fse19-ForkEfficency.R.

A variety of packages implementing commonly occurring application functionality are freely available e.g., database and testing[2006] frameworks.

The need to interoperate with other applications can cause a project to switch the database framework used by an application, or to support multiple database frameworks. The likelihood of an increase/decrease in the number of database frameworks used, and the time spent using different frameworks, is analysed in table 11.6.

5.6.1 Database evolution

Many applications make use of database functionality, with access requests often having the form of SQL queries embedded in strings within the source code. The structure of a database, its schema, may evolve, e.g., columns are added/removed from tables, and tables are added/removed; changes may be needed to support new functionality, or involve a reorganization, e.g., to save space or improve performance.

The kinds of changes made to a schema will be influenced by the need to support existing users (who may not want to upgrade immediately), and the cost of modifying existing code.

A study by Skoulis[1720] investigated changes to the database schema of several projects over time, including: Mediawiki (the software behind Wikipedia and other wikis), and Ensembl (a scientific project). Figure 5.68 shows one database schema in a linear growth phase (like the source code growth seen in some systems, e.g., fig 11.2), while the other has currently stopped growing, e.g., source code example fig 11.52. Systems change in response to customer requirements, and there is no reason to believe that the growth patterns of these two databases won't change.

Figure 5.69 shows the table survival curve for the Mediawiki and Ensembl database schema. Why is the table survival rate for Wikimedia much higher than Ensembl? Perhaps there are more applications making use of the contents of the Wikimedia schema, and the maintainers of the schema don't want to generate discontent in their user-base, or the maintainers are just being overly conservative. Alternatively, uncertainty over what data might be of interest in the Ensembl scientific project may result in the creation of tables that eventually turn out to be unnecessary, and with only two institutions involved, table removal may be an easier decision. The only way of finding out what customer demands are driving the changes is to talk to those involved.

The presence of tables and columns in a schema does not mean the data they denote is used by applications; application developers and the database administrator may be unaware they are unused, or they may have been inserted for use by yet to be written code.

A database may contain multiple tables, with columns in different tables linked using foreign keys. One study[1886] of database evolution found wide variation in the use of

[xii] That is, lots of variables performing poorly.

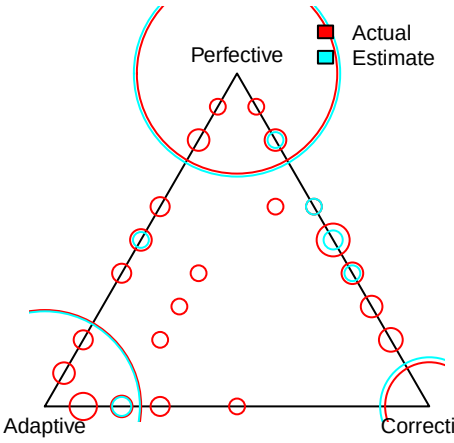


Figure 5.65: Ternary plot showing developers' estimated and actual percentage time breakdown performing adaptive, corrective and perfective work accumulated over 1,294 maintenance tasks; size of accumulation denoted by circle size. Data from Hatton.[786] Github–Local

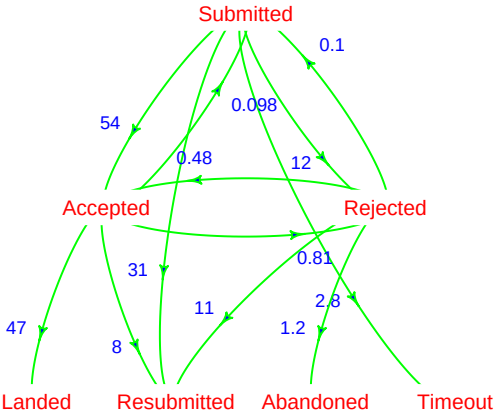


Figure 5.66: Percentage of patches submitted to WebKit (34,535 in total) transitioning between various stages of code review. Data from Baysal et al.[154] Github–Local

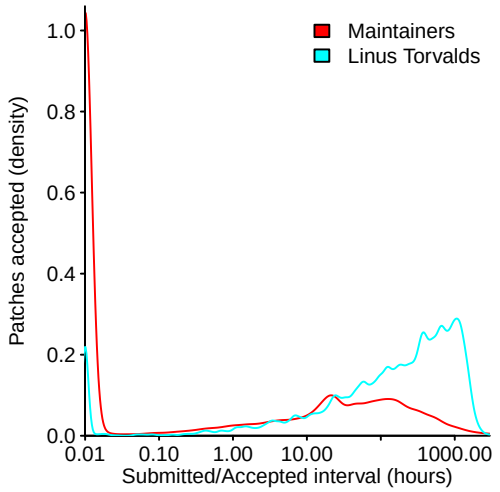


Figure 5.67: Density plot of interval between a patch passing review and being accepted by a maintainer, and interval between a maintainer pushing the patch to Linus Torvalds, and it being accepted into the blessed mainline (only patches accepted by Torvalds included). Data from Jiang et al.[918] Github–Local

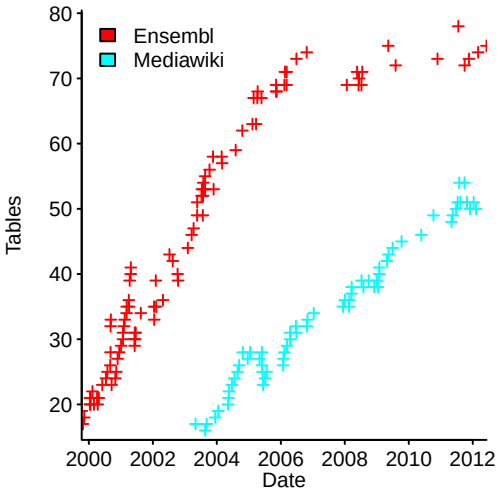


Figure 5.68: Evolution of the number of tables in the Mediawiki and Ensembl project database schema. Data from Skoulis.[1720] Github–Local

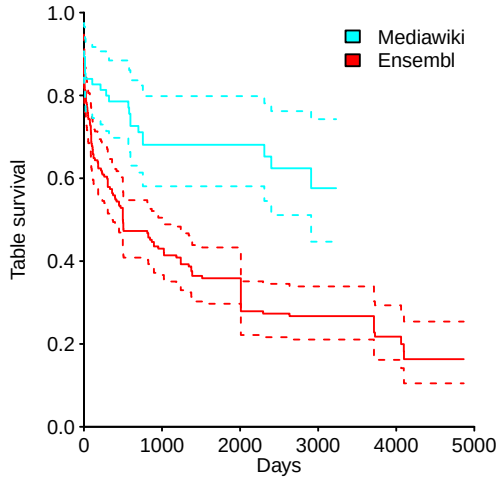


Figure 5.69: Survival curve for tables in Wikimedia and Ensembl database schema, with 95% confidence intervals. Data from Skoulis.[1720] Github–Local

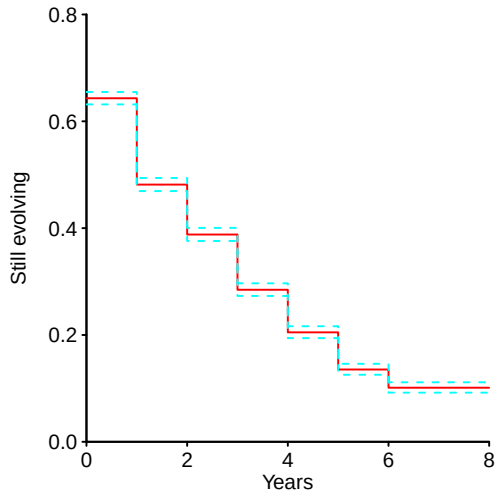


Figure 5.70: Survival curve for year of last modification of database programs, i.e., years before they stopped being changed, with 95% confidence intervals. Data from Blum.[213] Github–Local

foreign keys, e.g., foreign keys being an integral part of the database, or eventually being completely removed (sometimes driven to a change of database framework).

A study by Blum[213] investigated the evolution of databases and associated programs in the Johns Hopkins Oncology Clinical Information System (OCIS), a system that had been in operational use since the mid-1970s. Many small programs (a total of 6,605 between 1980 and 1988, average length 15 lines) were used to obtain information from the database. Figure 5.70 shows the survival curve for the year of last modification of a program, i.e., the probability that they stopped evolving after a given number of years.

Chapter 6

Reliability

6.1 Introduction

People are willing to continue using software containing faults, which they sometimes experience,[i] provided it delivers a worthwhile benefit. The random walk of life can often be nudged to avoid unpleasantness, or the operational usage time can be limited to keep within acceptable safety limits.[260] Regions of acceptability may exist in programs containing many apparently major mistakes, but supporting useful functionality.[1587]

Software systems containing likely fault experiences are shipped because it is not economically worthwhile fixing all the mistakes made during their implementation; also, finding and fixing mistakes, prior to release, is often constrained by available resources and marketing deadlines.[334]

Software release decisions involve weighing whether the supported functionality provides enough benefit to be attractive to customers (i.e., they will spend money to use it), after factoring in likely costs arising from faults experienced by customers (e.g., from lost sales, dealing with customer complaints and possible fixing reported problems, and making available an updated version).

How many fault experiences will customers tolerate, before they are unwilling to use software, and are some kinds of fault experiences more likely to be tolerated than others (i.e., what is the customer utility function)? Willingness-to-pay is a commonly used measure of risk acceptability, and for safety-critical applications terms such as *As Low As Reasonably Practicable* (ALARP)[764] and *So Far As Is Reasonably Practicable* (SFAIRP) are used.

Some hardware devices have a relatively short lifetime, e.g., mobile phones and graphics cards. Comparing the survival rate of reported faults in Linux device drivers, and other faults in Linux,[1438] finds that for the first 18 months, or so (i.e., 500 days), the expected lifetime of reported fault experiences in device drivers is much shorter than fault experiences in other systems (see figure 6.1); thereafter, the two reported fault lifetimes are roughly the same.

People make mistakes;[1533, 1566] economic considerations dictate how much is invested in reducing the probability that mistakes leading to costly fault experiences remain (either contained in delivered software systems, or as a component of a larger system). The fact that programs often contained many mistakes was a surprise to the early computer developers,[1960] as it is for people new to programming.[ii]

Developers make the coding mistakes that create potential fault experiences, and the environment in which the code executes provides the input that results in faults occurring (which may be experienced by the user). This chapter discusses the kinds of mistakes made, where they occur in the development process, methods used to locate them and techniques for estimating how many fault experiences can potentially occur. Issues around the selection of algorithms is outside the scope of this chapter; algorithmic reliability issues include accuracy[484] and stability of numerical algorithms,[826] and solutions include minimising the error in a dot product by normalizing the values being multiplied.[541]

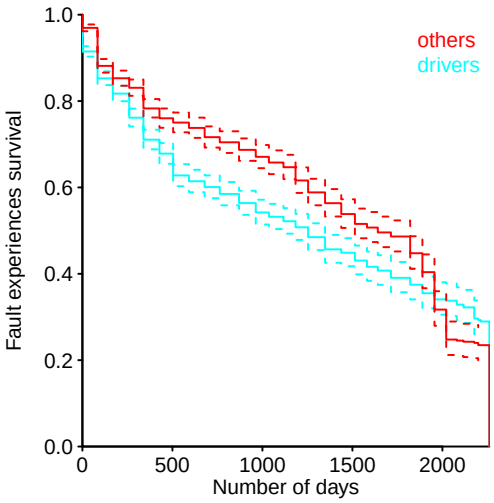


Figure 6.1: Survival rate of reported fault experiences in Linux device drivers and the other Linux subsystems. Data from Palix et al.[1438] Github–Local

[i] Experience is the operative word, a fault may occur and not be recognized as such.

[ii] The use of assertions for checking program behavior was proposed by Turing in 1949,[1313] and was later reinvented by others.

Operating as an engineering discipline does not in itself ensure that a constructed system has the desired level of reliability. There has been, roughly, a 30-year cycle for bridge failures;[1475] new design techniques and materials are introduced, and growing confidence in their use leads to overstepping the limits of what can safely be constructed.

What constitutes reliability, in a given context, is driven by customer requirements, e.g., in some situations it may be more desirable to produce an inaccurate answer than no answer at all, while in other situations no answer is more desirable than an inaccurate one.

Program inputs that cause excessive resource usage can also be a reliability issue. Examples of so-called *denial of service* attacks include regular expressions that are susceptible to nonlinear, or exponential, matching times for certain inputs.[442]

The early computers were very expensive to buy and operate, and much of the software written in the 1960s and 1970s was funded by large corporations or government bodies; the US Department of Defence took an active role in researching software reliability, and much of the early published research is based on the kinds of software development projects undertaken on behalf of the DOD and NASA projects during this period.

The approach to software system reliability promoted by these early research sponsors set the agenda for much of what has followed, i.e., a focus on large projects that are expected to be maintained over many years, or systems operating in situations where the cost of failure is extremely high and there is very limited time, if any, to fix issues, e.g., Space shuttle missions.[706]

This chapter discusses reliability from a cost/benefit perspective; the reason that much of the discussion involves large software systems is that a data driven discussion has to follow the data, and the prexisting research focus has resulted in more data being available for large systems. Mistakes have a cost, but these may be outweighed by the benefits of releasing the software containing them. As with the other chapters, the target audience is software developers and vendors, not users; it is possible for vendors to consider a software system to be reliable because it has the intended behavior, but for many users to consider it unreliable because it does not meet their needs.

The relative low cost of modifying existing software, compared to hardware, provides greater flexibility for trading-off upfront costs against the cost of making changes later (e.g., by reducing the amount of testing before release), knowing that it is often practical to provide updates later. For some vendors, the Internet provides an almost zero cost update distribution channel.

In some ecosystems it is impractical or impossible to update software once it has been released, e.g., executable code associated with the Ethereum cryptocurrency is stored on a blockchain (coding mistakes can have permanent crippling consequences[1827]).

Mistakes in software can have a practical benefit for some people, for instance, authors of computer malware have used mistakes in cpu emulators to detect that their activity may be monitored[1436] (and therefore the malware should remain inactive).

Mistakes are not unique to software systems; a study[1268] of citations in research papers found an average error rate of 20%.

Proposals[835] that programming should strive to be more like mathematics are based on the misconception that the process of creating proofs in mathematics is less error prone than creating software.[453]

The creation of mathematics shares many similarities with the creation of software, and many mistakes are made in mathematics;[1511] mathematical notation is a language with rules specifying syntax and permissible transformations. The size, complexity and technicality of modern mathematical proofs has raised questions about the ability of anybody to check whether they are correct, e.g., Mochizuki's proof of the *abc* conjecture,[306] and the Hales-Ferguson proof of the Kepler Conjecture.[1068] Many important theorems don't have proofs, only sketches of proofs and outline arguments that are believed to be correct;[1354] the sketches provide evidence used by other mathematicians to decide whether they believe a theorem is true (a theorem may be true, even though mistakes are made in the claimed proofs).

Mathematical proof differs from software in that the proof of a theorem may contain mistakes, and the theorem may still be true. For instance, in 1899 Hilbert found mistakes[1927] in Euclid's Elements (published around 300 BC); the theorems were true, and Hilbert was able to add the material needed to correct the proofs. Once a theorem is believed to be true, mathematicians have no reason to check its proof.

The social processes involved in the mathematics community coming to believe that a theorem is true, is evolving, to come to terms with believing machine-checked proofs.[1502] The nature and role of proof in mathematics continues to be debated.[819]

Mistakes are much less likely to be found in mathematical proofs than software, because a lot of specialist knowledge is needed to check new theorems in specialised areas, but a clueless button pusher can experience a fault in software simply by running it; also, there are few people checking proofs, while software is being checked every time it is executed. One study[313] found that 7% of small random modifications to existing proofs, written in Coq, did not cause the proof to be flagged as invalid.

Fixing a reported fault experience is one step in a chain of events that may result in users of the software receiving an update.

For instance, operating system vendors have different approaches to the control they exercise over the updates made available to customers. Apple maintains a tight grip over use of iOS, and directly supplies updates to customers cryptographically signed for a particular device, i.e., the software can only be installed on the device that downloaded it. Google supplies the latest version of Android to OEMs and has no control over what, if any, updates these OEMs supply to customers (who may chose to install versions from third-party suppliers). Microsoft sells Windows 10 through OEMs, but makes available security fixes and updates for direct download by customers.

Figure 6.2 shows some of the connections between participants in the Android ecosystem (number of each kind in brackets), and some edges are labeled with the number of known updates flowing between particular participants (from July 2011 to March 2016).

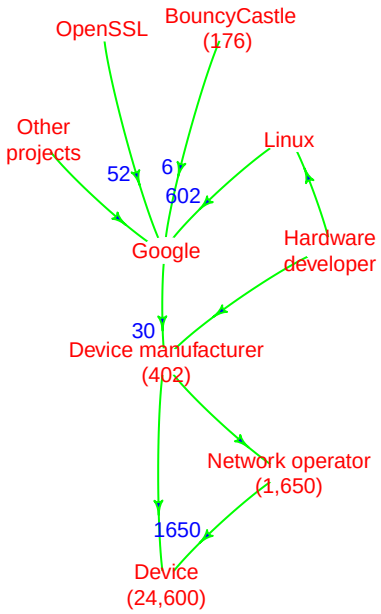


Figure 6.2: Flow of updates between participants in one Android ecosystem; number of each kind of member given in brackets, number of updates shipped on edges (in blue). Data from Thomas.[1829] Github–Local

Experiments designed to uncover unreliability issues may fail to find any. This does not mean that they are rare, the reason for failing to find a mistake may be lack of statistical power (i.e., the likelihood of finding an effect if one exists); this topic is discussed in section 10.2.3.

The software used for some applications is required to meet minimum levels of reliability, and government regulators (e.g., Federal Aviation Administration) may be involved in some form of certification (these regulators may not always have the necessary expertise, and delegate the work to the vendor building the system[508]).

In the U.S., federal agencies are required to adhere to an Executive Order[iii] that specifies: "Regulatory action shall not be undertaken unless the potential benefits to society for the regulation outweigh the potential costs to society." In some cases the courts have required that environmental, social and moral factors be included in the cost equation.[311]

Quality control: Manufacturing hardware involves making a good enough copy of a reference product. Over time[383] manufacturers have developed quality control techniques that support the consistent repetition of a production process, to deliver a low defect finished product. Software manufacturing involves copying patterns of bits, and perfect copies are easily made. The quality assurance techniques designed for the manufacture hardware are solving a problem relevant to software production.

The adoption of a quality certification process by an organization, such as ISO 9000, may be primarily symbolic.[813]

6.1.1 It's not a fault, it's a feature

The classification of program behavior as a fault, or a feature, can depend on the person doing the classification, e.g., user or developer. For instance, software written to manage a parts inventory may not be able to add a new item once the number of items it contains equals 65,536; a feature/fault that users will not encounter until an attempt is made to add an item that would take the number of parts in the inventory past this value.

Studies[493,820] of fault reports have found that many of the issues are actually requests for enhancement.

The choice of representation for numeric values places maximum/minimum bounds on the values that can be represented, and in the case of floating-point a percentage granularity on representable values. Business users need calculations on decimal values to be exact, something that is not possible using binary floating-point representations (unless emulated in software), and business oriented computers sometimes include hardware support

[iii] President Reagan signed Executive Order 12291 in 1981, and subsequent presidents have issued Executive Orders essentially affirming this requirement.[1797]

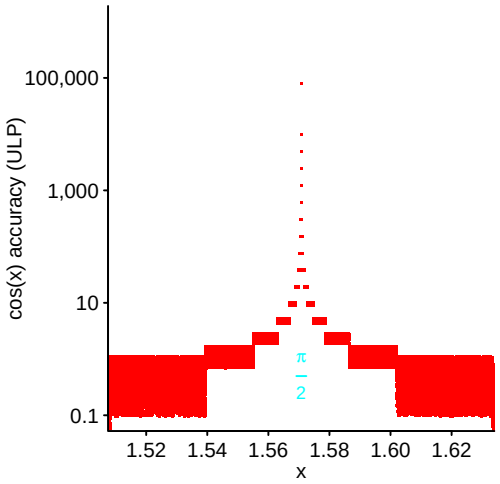


Figure 6.3: Accuracy of the value returned by the cos instruction on an Intel Core i7, for 52,521 argument values close to $\frac{\pi}{2}$. Data kindly provided by Duplichan.[519] Github–Local

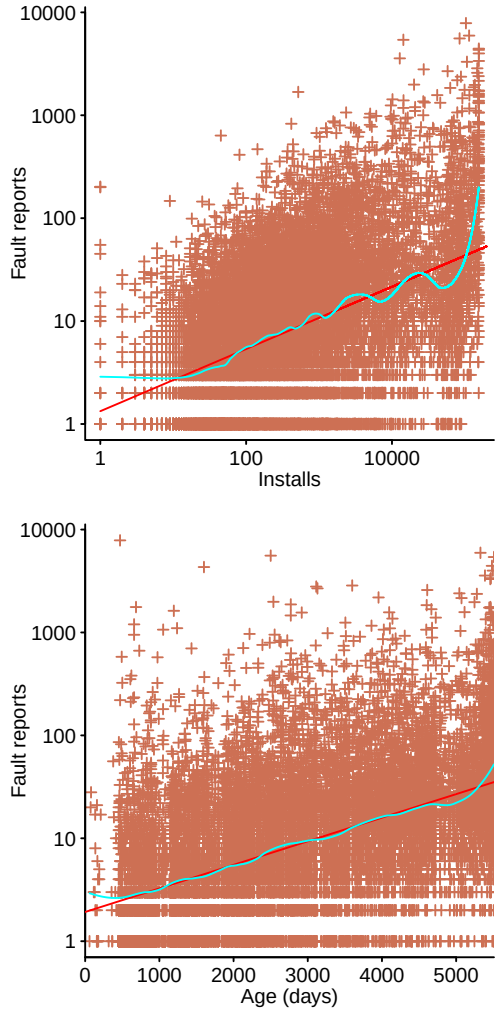


Figure 6.4: Reported faults against number of installations (upper) and age (lower). Data from the "wheezy" Debian release.[1862] Github–Local

for decimal floating-point operations; in other markets, implementation costs[414] resulted in binary becoming the dominant hardware representation for floating-point. While implementation costs eventually decreased to a point where it became commercially viable for processors to support both binary and decimal, much existing software has been written for processors that use a binary floating-point representation.

Intel's Pentium processor was introduced in 1993, as the latest member of the x86 family of processors. Internally the processor contains a 66-bit hardwired value for π.[iv] A double precision floating-point value is represented using a 53-bit mantissa, which means internal operations involving values close to π (e.g., using one of the instructions that calculate a trigonometric function) may have only 13-bits of accuracy, i.e., $66-53$. To ensure that the behavior of new x86 family processors are consistent with existing processors, subsequent processors have continued to represented π internally using 66-bits (rather than the 128-bits needed to achieve an accuracy of better than 1.5 ULP).

Figure 6.3 shows the error in the values returned by the cos instruction in Intel's Core i7-2600 processor, for 52,521 argument values close to $\frac{\pi}{2}$, expressed in units in the last place (ULP); from a study by Duplichan.[519]

The variation in the behavior of software between different releases, or running the same code on different hardware can be as large as the behavior affect the user is looking for, potentially a serious issue when medical diagnosis is involved.[746]

The accuracy of calculated results may be specified in the requirements, or the developer writing the code may be the only person to give any thought to the issue. Calculations involving floating-point values may not be exact, and the algorithms used may be sensitive to rounding errors.[1962] Developers may believe they are playing safe by using variables declared to have types capable of representing greater accuracy than is required[1619] (leading to higher than necessary resource usage,[726] e.g., memory, time and battery power). Small changes to numerical code can produce a large difference in the output produced,[1813] and simple calculations may require complex code to correctly implement, e.g., calculating $\sqrt{a^2+b^2}$.[222]

6.1.2 Why do fault experiences occur?

Two events are required for a running program to experience a software related fault:

- a mistake exists in the software,
- the program processes input values that cause it to execute the code containing the mistake in a way that results in a fault being experienced[411] (software that is never used has no reported faults).

Some coding mistakes are more likely to be encountered than others, because the input values needed for them to trigger a fault experience are more likely to occur during the use of the software. Any analysis of software reliability has to consider the interplay between the probabilistic nature of the input distribution, and coding mistakes present in the source code (or configuration information).

An increase in the number of people using a program is likely to lead to an increase in fault reports, because of both an increase in possible reporters and an increase in the diversity of input values.

The Ultimate Debian Database project[1862] collects information about packages included in the Debian Linux distribution, from users who have opted-in to the Debian Popularity Contest. Figure 6.4 shows the numbers of installs (for the "wheezy" release) of each packaged application against faults reported in that package, and also age of the package against faults reported (data from the Debian Bug Tracking System, which is not the primary fault reporting system for some packages); also, see fig 11.23. A fitted regression model is:

$$reported\_bugs = e^{-0.15+0.17\log(insts)+(30+2.3\log(insts))\times age\times 10^{-5}}$$

For an *age* between 1,000–6,000 and installs between 10–20,000 ($\log(insts)$ is between 2–10), the number of installations (a proxy for number of users) appears to play a larger role in the number of reported faults, compared to *age* (i.e., the amount of time the package has been included in the Debian distribution). The huge amount of variance in the data points to other factors having a major impact on number of reported faults.

[iv] The 66-bit value is: `C90FDAA2 2168C234 C`, while the value to 192-bits is: `C90FDAA2 2168C234 C4C66 28B 80DC1CD1 29024E08 8A67CC74`.

A study[1787] of TCP checksum performance found that far fewer corrupt network packets were detected in practice, than expected (by a factor of between 10 and 100). The difference between practice and expectation was found to be caused by the non-uniform distribution of input values (the proof that checksum values are uniformly distributed assumes a uniform distribution of input values).

A hardware fault may cause the behavior of otherwise correctly behaving software to appear to be wrong; hardware is more likely to fail as the workload increases.[903]

6.1.3 Fault report data

While fault reports have been studied almost since the start of software development, until open source bug repositories became available there was little publicly available fault report data. The possibility of adverse publicity, and the fear of legal consequences, from publishing information on mistakes found in software products was not lost on commercial organizations, with nearly all of them treating such information as commercially confidential. While some companies maintained software fault report databases,[724] these were rarely publicly available.

During the 1970s, the Rome Air Defence Center published many detailed studies of software development,[424] and some included data on faults experienced by military software projects.[1967] However, these reports were not widely known about, or easy to obtain, until they became available via the Internet; a few studies were published as books.[1826]

The few pre-Open source datasets analysed in research papers contained relatively small numbers of fault reports, and if submitted for publication today would probably be rejected as not worthy of consideration. These studies[1471,1473] usually investigated particular systems, listing percentages of faults found by development phase, and the kinds of faults found; one study[1918] listed fault information relating to one application domain: medical device software. An early comprehensive list of all known mistakes in a widely used program was for LaTeX.[1030]

The economic impact of loss of data, due to poor computer security, has resulted in some coding mistakes in some programs (e.g., those occurring in widely used applications that have the potential to allow third parties to gain control of a computer) being recorded in security threat databases. Several databases of security related issues are actively maintained, including: the NVD (National Vulnerability Database[890]), the VERIS Community Database (VCDB);[1887] an Open source vulnerability database, the Exploit database (EDB)[529] lists proof of concept vulnerabilities; mistakes in code that may be exploited to gain unauthorised access to a computer (vulnerabilities discovered by security researchers who have a motivation to show off their skills), and the Wooyun program.[2014]

While many coding mistakes exist in widely used applications, only a tiny fraction are ever exploited to effectively mount a malicious attack on a system.[1470]

In some application domains the data needed to check the accuracy of a program's output may not be available, or collected for many years, e.g., long range weather forecasts. The developers of the Community Earth System Model compare the results from previous climate calculations to determine whether modified code produces statistically different results.[121]

A study by Sadat, Bener and Miranskyy[1625] investigated issues involving connecting duplicate fault reports, i.e., reports involving the same mistake. Figure 6.5 shows the connection graph for Eclipse report 6325 (report 4671 was the earliest report covering this issue).

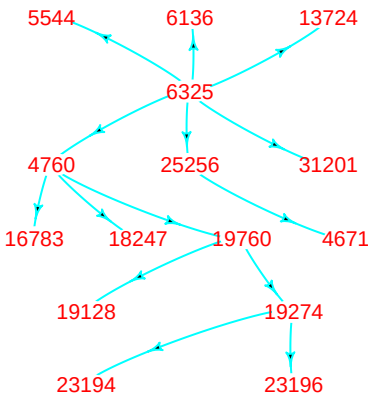


Figure 6.5: Duplicates of Eclipse fault report 4671 (report 6325 was finally chosen as the master report); arrows point to report marked as duplicate of an earlier report. Data from Sadat et al.[1625] Github–Local

Fixing a mistake may introduce a new mistake, or may only be a partial fix, i.e., fixing commits may be a continuation of a fix for an earlier fault report.

A study by Xiao, Zheng, Jiang and Sui[1985] investigated *regression* faults in Linux (the name given to fault experiences involving features that worked correctly, up until a code change). Figure 6.6 shows a graph of six fault reports for Linux (in red), the commits believed to fix the coding mistake (in blue), and subsequent commits needed to fix mistake(s) introduced by an earlier commit (in blue, follow arrows).

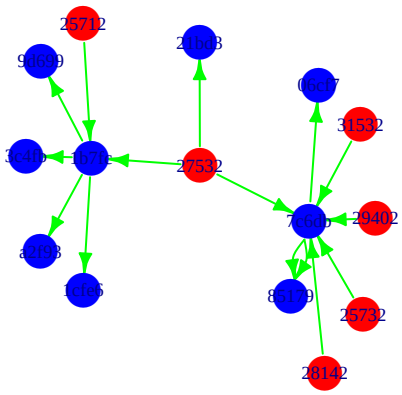


Figure 6.6: Six fault reports (red), their associated bug fixing commits (blue), and subsequent commits to fix mistakes introduced by the earlier commit (blue). Data from Xiao et al.[1985] Github–Local

How similar are the characteristics of Open source project fault report data, compared to commercial fault report data?

Various problems have been found with Open source fault report data, which is not to say that fault report data on closed source projects is not without its own issues; known problems include:

- reported faults do not always appear in the fault report databases; e.g., serious bugs tend to be under-reported in commit logs.[200, 1373] One study[105] of fault reports for Apache, over a 6-week period, found that only 48% of bug fixes were recorded as faults in the Bugzilla database; the working practice of the core developers was to discuss serious problems on the mailing list, and many fault experiences were never formally logged with Bugzilla,
- fault reports are misclassified. One study of fault reports[820] (also see section 14.1.1) found that 42.6% of fault reports had been misclassified, with 39% of files marked as defective not actually containing any reported fault (e.g., were requests for enhancement),
- reporting bias: fault experiences discovered through actively searching for them,[574] rather than normal program usage (e.g., Linux is a popular target for researchers using fuzzing tools, and `csmith` generated source code designed to test compilers). The reporting of vulnerabilities contained, or not, in the NVD has been found to be driven by a wide variety social, technical and economic pressures.[359, 1374] Data on fault reports discovered through an active search process may have different characteristics compared to faults experienced through normal program usage,
- the coding mistake is not contained within the source of the program cited, but is contained within a third-party library. One study[1178] surveyed developers about this issue,
- fault experience could not be reproduced or was intermittent: a study[1628] of six servers found that on average 81% of the 266 fault reports analysed could be reproduced deterministically, 8% non-deterministically, 9% were timing dependent, plus various other cases,

Fault report data does not always contain enough information to answer the questions being asked of it, e.g., using incidence data to distinguish between different exponential order fault growth models[1283] (information is required on the number of times the same fault experience has occurred; see section 4.3.2).

6.1.4 Cultural outlook

Cultures vary in their members' attitude to the risk of personal injury and death; different languages associate different concepts with the word *reliability* (e.g., Japanese[1289]), and the English use of the term *at risk* has changed over time.[2030] A study by Viscusi and Aldy[1904] investigated the value of a statistical life in 11 countries, and found a range of estimates from $0.7 million to $20 million (adjusted to the dollar rate in 2000). Individuals in turn have their own perception of risk, and sensitivity to the value of life.[1728] Some risks may be sufficiently outside a persons' experience that they are unable to accurately estimate their relative likelihood,[1134] or be willing to discount an occurrence affecting them, e.g., destruction of a city, country, or all human life, by a meteor impact.[1737]

Public perception of events influences, and is influenced by, media coverage (e.g., in a volcano vs. drought disaster, the drought needs to kill 40,000 times as many people as the volcano to achieve the same probability of media coverage[536]). A study[536] of disasters and media coverage found that when a disaster occurs at a time when other stories are deemed more newsworthy, aid from U.S. disaster relief is less likely to occur.

Table 6.1, from a 2011 analysis by the UK Department for Transport,[1842] lists the average value that would have been saved, per casualty, had an accident not occurred. A report[354] from the UK's Department for Environment, Food and Rural Affairs provides an example of the kind of detailed analysis involved in calculating a monetary valuation for reducing risk.

Injury severity	Lost output	Human costs	Medical and ambulance	Total
Fatal	£545,040	£1,039,530	£940	£1,585,510
Serious	£21,000	£144,450	£12,720	£178,160
Slight	£2,220	£10,570	£940	£13,740
Average	£9,740	£35,740	£2,250	£47,470

Table 6.1: Average value of prevention per casualty, by severity and element of cost (human cost based on willingness-to-pay values); last line is average over all casualties. Data from UK Department for Transport.[1842]

A study by Costa and Kahn[408] investigated changes in the value of life in the USA, between 1940 and 1980. The range of estimates, adjusted to 1990 dollars, was $713,000 to $996,000 in 1940, and $4.144 million to $5.347 million in 1980.

Some government related organizations, and industrial consortia, have published guidelines covering the use of software in various applications, e.g., in medical devices,[581] cybersecurity,[1558] and automotive.[92, 1291, 1292] Estimates of the number of deaths associated with computer related accidents contain a wide margin of error.[1186]

Governments are aware of the dangers of society becoming overly risk-averse, and some have published risk management policies.[1916]

People often express uncertainty using particular phrases, rather than numeric values. Studies[1317] have found that when asked to quantify a probabilistic expression, the range of values given can be quite wide.

A study by Budescu, Por, Broomell and Smithson[274] investigated how people in 24 countries, speaking 17 languages, interpreted uncertainty statements containing four probability terms, i.e., very unlikely, unlikely, likely and very likely, translated to the subjects' language. Figure 6.7 shows the mean percentage likelihood estimated by people in each country to statements containing each term.

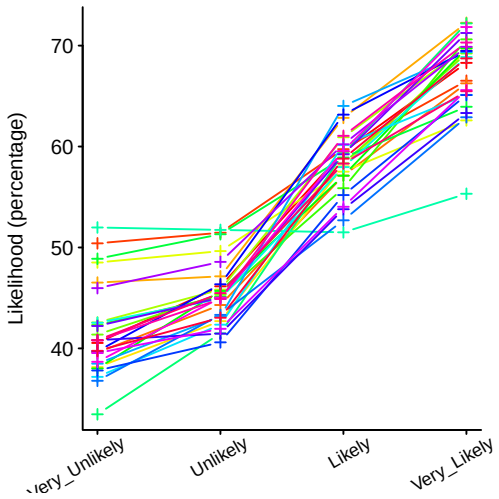


Figure 6.7: Mean percentage likelihood of (translated) statements containing a probabilistic term; one colored line per country. Data from Budescu et al.[274] Github–Local

The U.S. Department of Defense Standard MIL-STD-882E[474] defines numeric ranges for some words that can be used to express uncertainty, when applied to an individual item; these words include:

- *Probable*: "Will occur several times in the life of an item"; probability of occurrence less than 10^{-1} but greater than 10^{-2}.
- *Remote*: "Unlikely, but possible to occur in the life of an item"; probability of occurrence less than 10^{-3} but greater than 10^{-6}.
- *Improbable*: "So unlikely, it can be assumed occurrence may not be experienced in the life of an item"; probability of occurrence less than 10^{-6}.

Some phrases are used to express relative position on a scale, e.g., hot/warm/cold water describe position on a temperature scale. A study by Sharp, Paul, Nagesh, Bell and Surdeanu[1679] investigated, so-called *gradable adjectives*, e.g., huge, small. Subjects saw statements such as: "Most groups contain 1470 to 2770 mards. A particular group has 2120 mards. There is a moderate increase in this group."; subjects were then asked: "How many mards are there?".

Figure 6.8 shows violin plots for the responses given to statements/questions involving various gradable quantity adjectives (see y-axis); the x-axis is in units of standard deviation from the mean response, i.e., a normalised scale.

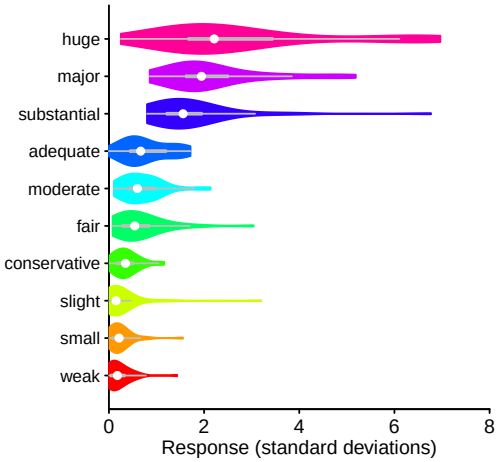


Figure 6.8: Subjects' perceived change in the magnitude of a quantity, when the given gradable size adjective is present. Data from Sharp et al.[1679] Github–Local

The interpretation of a quantifier (i.e., a word indicating quantity) may be context dependent. For instance, "few of the juniors were accepted" may be interpreted as: a small number were accepted; or, as: less than the expected number were accepted.[1866]

6.2 Maximizing ROI

A vendor's approach to product reliability is driven by the desire to maximize return on investment. The reliability tradeoff involves deciding how much to invest in finding and fixing implementation mistakes prior to release, against fixing fault experiences reported after release. Factors that may be part of the tradeoff calculation include:

- some mistakes will not generate fault experiences, and it is a waste of resources finding and fixing them. The lack of fault experiences, for a particular coding mistake, may be a consequence of software having a finite lifetime (see fig 3.7), or the source code containing the mistake being rewritten before it causes a fault to be experienced.

  A study by Di Penta, Cerulo and Aversano[492] investigated the issues reported by three static analysis tools (Rats, Splint and Pixy), when run on each release of several large software systems, e.g., Samba, Squid and Horde. The reported issues were processed, to find the first/last release where each issue was reported for a particular line of code (in some cases the issue was reported in the latest release).

  Figure 6.9 shows the survival curve for the two most common warnings reported by Splint (memory problem and type mismatch, make up over 85% of all generated warnings), where the warnings ceased because of code modifications that were not the result of a reported fault being fixed; also see fig 11.80.

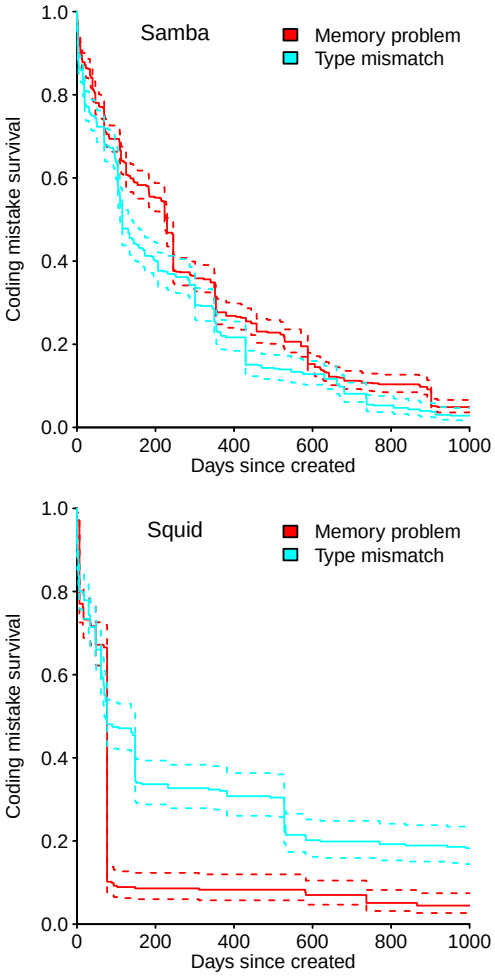


Figure 6.9: Survival curves of the two most common warnings reported by Splint in Samba and Squid, where survival was driven by code changes and not fixing a reported fault; with 95% confidence intervals. Data from De Penta et al.[492] Github–Local

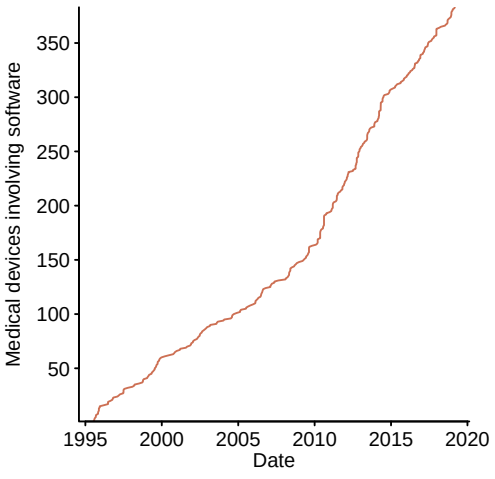


Figure 6.10: Cumulative number of class III (high-risk) medical devices, containing software in their product summary, achieving premarket approval from the FDA. Data from FDA.[582] Github–Local

The average lifetime of coding mistakes varies between programs and kind of mistake.[1437]

- there may be a competitive advantage to being first to market with a new or updated product; the largest potential costs may be lost sales, rather than the cost of later correction of reported faults, i.e., it may be worth taking the risk that customers are not deterred by the greater number of fault experiences; so-called *frontier risk* thinking,
- too much investment in finding and fixing mistakes can be counterproductive, e.g., customers who do not encounter many fault experiences may be less motivated to renew a maintenance agreement,
- whether the vendor cost of a fault experience includes the user costs associated with the user fault experience, e.g., when software is developed for in-house use. In extreme cases the cost of a fault experience can be hundreds of millions of dollars.[1332]

  With COTS the cost of fault experiences is asymmetric, i.e., the customer bears the cost of the fault experience itself, while the vendor can choose whether to fix the mistake and ship an update. Existing consumer laws provide some legal redress[973] (at least until the existing legal landscape changes[1662]).

  While coding mistakes are exploited by computer viruses, causing business disruption, the greatest percentage of computer related loses come from financial fraud by insiders, and traditional sources of loss such as theft of laptops and mobiles.[1581]

All mistakes have the potential to have costly consequences, but in practice most appear to be an annoyance. One study[39] found that only 2.6% of the vulnerabilities listed in the NVD have been used, or rather their use has been detected, in viruses and network threat attacks on computers.

Figure 6.10 shows the growth in the number of high-risk medical devices, containing the word software in their product summary, achieving premarket approval from the Federal Food and Drug Administration.

The *willingness-to-pay* (WTP) approach to reliability aims to determine the maximum amount that those at risk would individually be willing to pay for improvements to their, or other people's safety. Each individual may only be willing to pay a small amount, but as a group the amounts accumulate to produce an estimated value for the group "worth" of a safety improvement. For instance, assuming that in a group of 1.5 million people, each person is willing to pay £1 for safety improvements that achieve a 1 in 1 million reduction in the probability of death; the summed WTP *Value of Preventing a Statistical Fatality* (VPF) is £1.5 million (the same approach can be used to calculate *Value of Preventing non-fatal Injuries*).

A market has developed for coding mistakes that can be exploited to enable third parties to gain control of other peoples' computers[38] (e.g., spying agencies and spammers), and some vendors have responded by creating vulnerability reward programs. The list of published rewards are both a measure of the value vendors place on the seriousness of particular kinds exploitable mistakes, and the minimum amount the vendor considers sufficient to dissuade discoverers spending time finding someone willing to pay more.

Bountysource is a website where people can pledge a monetary bounty, payable when a specified task is performed. A study by Zhou, Wang, Bezemer, Zou and Hassan[2021] investigated bounties offered to fix specified open issues, of some Github project (the 2,816 tasks had a total pledge value of $365,059). Figure 6.11 shows the total dollar amount pledged for each task; data broken down by issue reporting status of person specifying the task.

The viability of making a living from bug bounty programs is discussed in connection with figure 4.43.

If a customer reports a fault experience, in software they have purchased, what incentive does the vendor have to correct the problem and provide an update (they have the customers' money; assuming the software is not so fault ridden that it is returned for a refund)? Possible reasons include:

- a customer support agreement requires certain kinds of reported faults to be fixed,
- public perception and wanting to maintain customer good will, in the hope of making further sales. One study[78] found that the time taken to fix publicly disclosed vulnerabilities was shorter than for vulnerabilities privately disclosed to the vendor; see section 11.11.3.1,

- a fill-in activity for developers, when no other work is available,
- it would be more expensive not to fix the coding mistake, e.g., the change in behavior produced by the fault experience could cause an accident, or negative publicity, that has an economic impact greater than the cost of fixing the mistake. Not wanting to lose money because a mistake has consequences that could result in a costly legal action (the publicity around a decision driven by a vendors' cost/benefit analysis may be seen as callous, e.g., the Ford Motor Company had to defend itself in court[1110] over its decision not to fix a known passenger safety issue, because they calculated the cost of loss of human life and injury did not exceed the benefit of fixing the issue).

Which implementation mistakes are corrected? While there is no benefit in correcting mistakes that customers are unlikely to experience, it may not be possible to reliably predict whether a mistake will produce a customer fault experience (leading to every mistake having to treated as causing a fault experience). Once a product has been released and known to be acceptable to many customers, there may not be any incentive to actively search for potential fault experiences, i.e., the only mistakes corrected may be those associated with a customer fault reports.

In some cases applications are dependent on the libraries supplied by the vendor of the host platform. One study[1149] of Apps running under Android found that those Apps using libraries that contained more reported faults had a slightly smaller average user rating in the Google Play Store.

What motivates developers to fix faults reported in Open source projects? Possible reasons include:

- they work for a company that provides software support services for a fee. Having a reputation as the go-to company for a certain bundle of packages is a marketing technique for attracting the attention of organizations looking to outsource support services, or pay for custom modifications to a package, or training.

  Correcting reported faults is a costly signal that provides evidence a company employs people who know what they are doing, i.e., status advertising,
- developers dislike the thought of being wrong or making a mistake; a reported fault may be fixed to make them feel better (or to stop it preying on their mind), also not responding to known problems in code is not considered socially acceptable behavior in some software development circles. Feelings about what constitutes appropriate behavior may cause developers to want to redirect their time to fixing mistakes in code they have written or feel responsible for, provided they have the time; problems may be fixed by developers when management thinks they are working on something else.

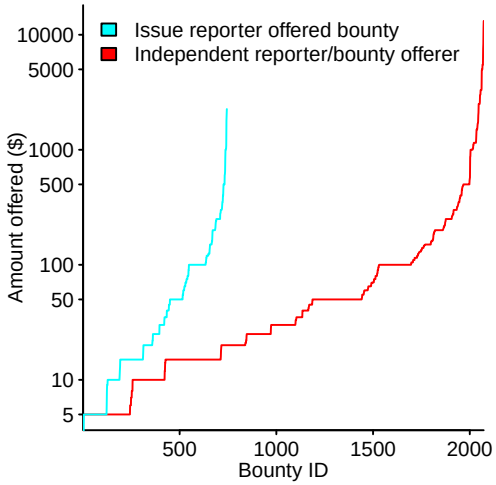


Figure 6.11: Value of bounties offered for 2,816 tasks addressing specified open issues of a Github project; pledges stratified by status of person reporting the pledge issue. Data from Zhou et al.[2021] Github–Local

6.3 Experiencing a fault

Unintended software behavior is the result of an interaction between a mistake in the code (or more rarely an incorrect translation by a compiler[1136]), and particular input values.

A program's source code may be riddled with mistakes, but if typical user input does not cause the statements containing these mistakes to be executed, the program may gain a reputation for reliability. Similarly, there may only be a few mistakes in the source code, but if they are frequently experienced the program may gain a reputation for being fault-ridden.

Almost all existing research on software reliability has focused on the existence of the mistakes in source code. This is convenience sampling, large amounts of Open source is readily available, while information on the characteristics of program input is very difficult to obtain.

The greater the number of people using a software system, the greater the volume and variety of inputs it is likely to process: consequently there are likely to be more reported fault experiences.

A study by Shatnawi[1681] investigated the impact of the number of sites using a release of telecommunication switch software, on the number of software failures reported. A regression model fitted to the data shows that reported fault experiences decreased over time, and increases with the number of installed sites; see Github–reliability/2014-04-13.R.

A study by Lucente[1165] investigated help desk incident reports, from 800 applications used by a 100,000 employee company with over 120,000 desktop machines. Figure 6.12

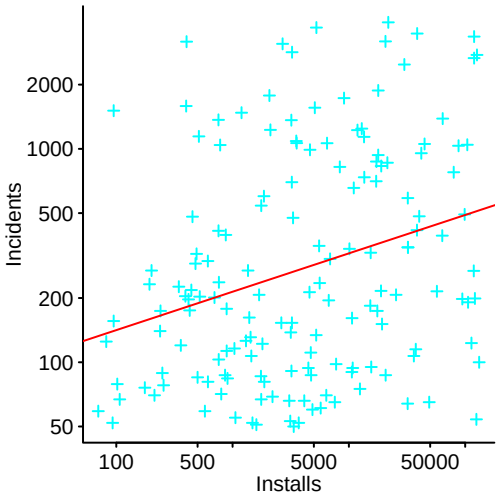


Figure 6.12: Number of incidents reported for each of 800 applications installed on over 120,000 desktop machines; line is fitted regression model. Data from Lucente.[1165] Github–Local

shows the number of incidents reported increasing with the number of installs (as is apparent from the plot, the number of installs only explains a small percentage of the variance).

A comparison of the number of fault experiences reported in different software systems[1489] might be used to estimate the number of people using the different systems; any estimate of system reliability has to take into account the volume of usage, and the likely distribution of input values.

The same user input can produce different fault experiences in different implementations of the same functionality, e.g., the POSIX library on 15 different operating systems.[488]

A study by Dey and Mockus[489] investigated the impact of number of users, and time spent using a commercial mobile application, on the number of exceptions the App experienced. The App used Google analytics to log events, which provides daily totals. On many days no exceptions were logged, and when exceptions did occur it is likely that the same set of faults were repeatedly experienced. The fitted regression models (with exceptions as the response variable) contain both user-uses and new-user-uses as power laws, with the exponent for new-user-uses being the largest, and the impact of the version of Android installed on the users' device varied over several orders of magnitude; see Github–reliability/2002-09989.R.

Figure 6.13 shows, for one application on prerelease, the number of exceptions per day for a given number of new users.

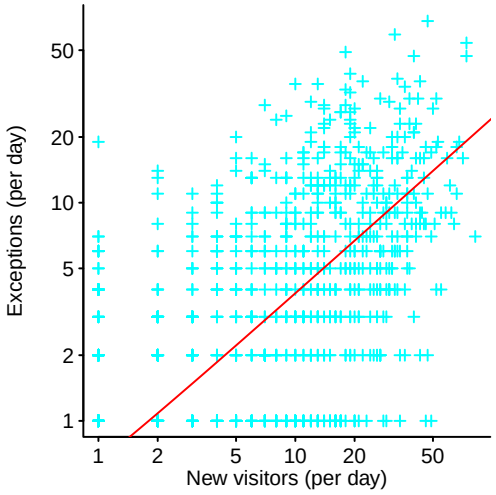


Figure 6.13: Number of exceptions experienced per day against number of new users of the application, for one application prior to its general release; line is a fitted regression model of the form: $Exceptions \propto newUserUses^{0.8}$. Data from Dey et al.[489] Github–Local

6.3.1 Input profile

The environments in which we live, and software systems operate, often experience regular cycles of activity; events are repeated with small variations at a variety of scales (e.g., months of the year, days of the week, and frequent use of the same words;[1106] also see section 2.4.4).

The input profile that results in faults being experienced is an essential aspect of any analysis of program reliability. For instance, when repeatedly adding pairs of floating-point values, with each value drawn from a logarithmic distribution, the likelihood of experiencing an overflow[590] may not be low enough to be ignored (the probability for a single addition overflowing is 5.3×10^{-5} for single precision IEEE and 8.2×10^{-7} for double[v]).

Undetected coding mistakes[vi] exist in shipped systems because the input values needed to cause them to generate a fault experience were not used during the testing process.

Address traces illustrate how the execution characteristics of a program can be dramatically changed by its input. Figure 6.14 shows the number of memory accesses made while executing `gzip` on two different input files. The small colored boxes representing 100,000 executed instruction on the x-axis, and successive 4,096 bytes of stack on the y-axis, the colors denote number of accesses within the given block (using a logarithmic scale).

Mistakes in code that interact with input values that are likely to be selected by developers, during testing, are likely to be fixed during development; beta testing is one method for discovering customer oriented input values that developers have not been testing against. Ideally the input profiles of the test and actual usage are the same, otherwise resources are wasted fixing mistakes that the customer is less likely to experience.

The test process may include automated generation of input values; see section 6.6.2.1.

Does the interaction between mistakes in the source code, and an input profile, generate any recurring patterns in the frequency of fault experiences?

One way of answering this question is to count the number of inputs successfully processed by a program between successive fault experiences.

A study by Nagel and Skrivan[1347] investigated the timing characteristics of fault experiences in three programs, each written independently by two developers. During execution, each program processed inputs selected from the set of permissible values, when a

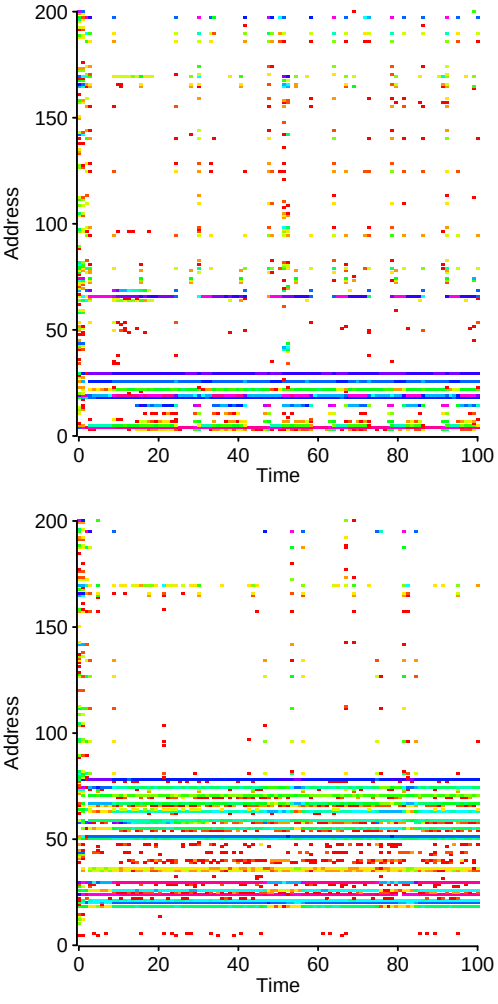


Figure 6.14: Number of accesses to memory address blocks, per 100,000 instructions, when executing `gzip` on two different input files. Data from Brigham Young[255] via Feitelson. Github–Local

[v] The general formula is: $\frac{\pi^2}{6\left(\ln \frac{\Omega}{\omega}\right)^2}$ where: Ω and ω are the largest and smallest representable values respectively); the probability of a subtraction underflowing has a more complicated form, but the result differs by at most 1 part in 10^{-9} from the addition formula.

[vi] Management may consider it cost effective to ship a system containing known mistakes.

fault was experienced its identity, execution time up to that point and number of input cases processed were recorded; the coding mistake was corrected and program execution continued until the next fault experience, until five or six faults had been experienced, or the mistake was extremely time-consuming to correct (the maximum number of input cases on any run was 32,808). This cycle was repeated 50 times, always starting with the original, uncorrected, program; the term *repetitive run modeling* was used to denote this form of testing. A later study by Nagel, Scholz and Skrivan[1346] partially replicated and extended this study.

Figure 6.15 shows the order in which distinct faults were experienced by implementation A2, over 50 replications; edge values show the number of times the n^{th} fault experience was followed by a particular fault experience. For example, starting in state A2−0 fault experience 1 was the first encountered during 37 runs, and this was followed by fault experience 3 during one run.

Figure 6.16, upper plot, shows the number of input cases processed before a given number of fault experiences, during the 50 runs of implementation A2; the lower plot shows the number of inputs processed before each of five distinct fault experiences.

What is the likely number of inputs that have to be processed by implementation A2 for the sixth distinct fault to be experienced? A regression model could be fitted to the data seen in the upper plot of figure 6.16, but a model fitted to this sample of five distinct fault experiences will have a wide confidence interval. There is no reason to expect that the sixth fault will be experienced after processing any number of inputs, there appears to be a change point after the fourth fault, but this may be random noise that has been magnified by the small sample size.

The time and cost of establishing, to a reasonable degree of accuracy, that users of a program have a very low probability of experiencing a fault[286] may not be economically viable.

A study by Dunham and Pierce[516] replicated and extended the work of Nagel and Skriva; problem 1 was independently reimplemented by three developers. The three implementations were each tested with 500,000 input cases, when a fault was experienced the number of inputs processed was recorded, the coding mistake corrected, and program execution restarted. This cycle was repeated four times, always starting with the original implementation, fixing and recording as faults were experienced.

Figure 6.17 shows the number of input cases processed, by two of the implementations (only one fault was ever experienced during the execution of the third implementation), before a given number of fault experiences, during each of the four runs. The grey lines are an exponential regression model fitted to each implementation; these two lines show that as the number of faults experienced grows, more input cases are required to experience another fault, and that code written by different developers has different fault experience rates per input.

A second study by Dunham and Lauterbach[515] used 100 replications for each of the three programs, and found the same pattern of results seen in the first study.

Some published fault experience experiments have used time (computer or user), as a proxy for the quantity of input data. It is not always possible to measure the quantity of input processed, and time may be more readily available.

A study by Wood[1977] analysed fault experiences encountered by a product Q/A group in four releases of a subset of products. Figure 6.18 shows that the fault experience rate is similar for the first three releases (the collection of test effort data for release 4 is known to have been different from the previous releases).

A study by Pradel[1519] searched for thread safety violations in 15 classes in the Java standard library and JFreeChart, that were declared to be thread safe, and 8 classes in Joda-Time not declared to be thread safe; automatically generated test cases were used. Thread safety violations were found in 22 out of the 23 classes; for each case the testing process was run 10 times, and the elapsed time to discover the violation recorded. Figure 6.19 illustrates the variability in the timing of the violations experienced.

A study by Adams[7] investigated reported faults in applications running on IBM mainframes between 1975 and 1980. Figure 6.20 shows that approximately one third of fault experiences first occurred on average every 5,000 months of execution time (over all uses of the product). Only around 2% of fault experiences first occurred after five months of execution time.

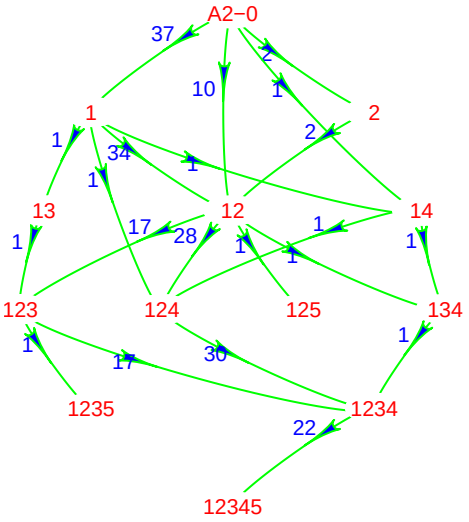


Figure 6.15: Transition counts of five distinct fault experiences in 50 runs of program A2; nodes labeled with each fault experienced up to that point. Data from Nagel et al.[1347] Github–Local

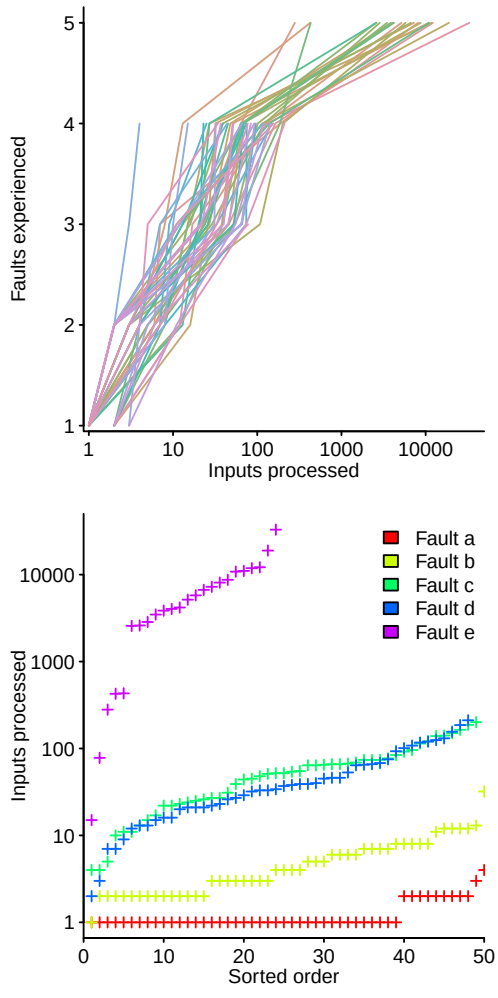


Figure 6.16: Number of input cases processed before a particular fault was experienced by program A2; the list is sorted for each distinct fault. Data from Nagel et al.[1347] Github–Local

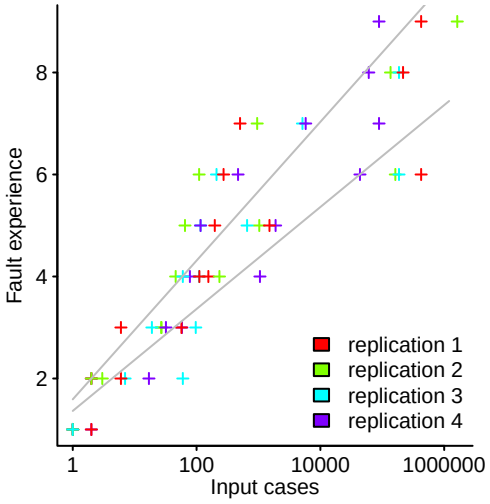


Figure 6.17: Number of input cases processed by two implementations before a fault was experienced, with four replications (each a different color); grey lines are a regression fit for one implementation. Data from Dunham et al.[516] Github–Local

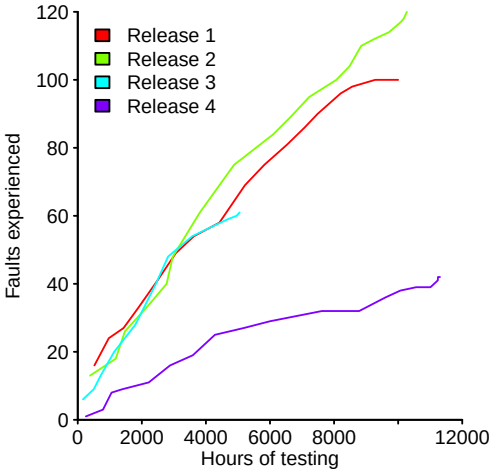


Figure 6.18: Faults experienced against hours of testing, for four releases of a product. Data from Wood.[1977] Github–Local

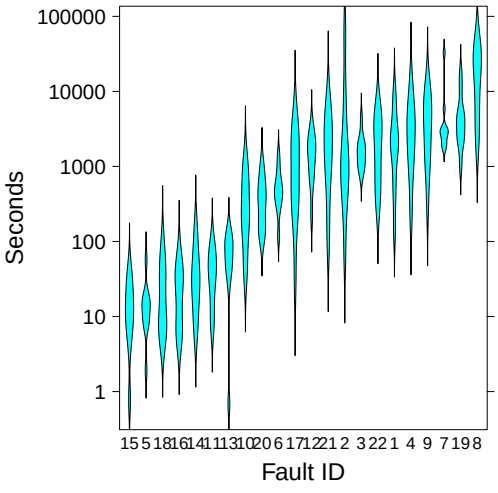


Figure 6.19: Time taken to encounter a thread safety violation in 22 Java classes, violin plots for 10 runs of each class. Data kindly supplied by Pradel.[1519] Github–Local

Multiple fault experiences produced by the same coding mistake provide information about the likelihood of encountering input that can trigger that fault experience. Regression models fitted using a biexponential equation (i.e., $a \times e^{b \times x} + c \times e^{d \times x}$, where x is the rank order of occurrences of each fault experience) have been fitted to a variety of program crash data; see fig 11.53.

A study by Zhao and Liu[2015] investigated the crash faults found by fuzzing the files processed by six Open source programs. Figure 6.21 shows the number of unique crash faults experienced by `convert` and `autotrace` (estimated by tracing back to a program location), along with lines fitted using biexponential regression models.

Why is a biexponential model such a good fit? A speculative idea is that the two exponentials are driven by the two independent processes that need to interact to produce a fault experience: the distribution of input values, and the mistakes contained in the source code.[vii]

6.3.2 Propagation of mistakes

The location in the code that triggers a fault experience may appear many executable instructions after the code containing the mistake (that is eventually modified to prevent further the fault experiences).

In some input values a coding mistake may not propagate from the mistake location, to a code location where it can trigger a fault experience. For instance, if variable `x` is mistakenly assigned the value `3`, rather than `2`, the mistake will not propagate past the condition: `if (x < 8)` (because the behavior is the same for both the correct and mistake value); for this case, an opportunity to propagate only occurs when the mistaken value of `x` changes the value of the conditional test.

How robust is code to small changes to the correct value of a variable?

A study by Danglot, Preux, Baudry and Monperrus[435] investigated the propagation of one-off perturbations in 10 short Java programs (42 to 568 LOC). The perturbations were created by modifying the value of an expression once during the execution of a program, e.g., by adding, or subtracting, one. The effect of a perturbation on program behavior could be to cause it to raise an exception, output an incorrect result, or have no observed effect, i.e., the output is unchanged. Each of a program's selected perturbation points were executed multiple times (e.g., the 41 perturbation points selected for the program `quicksort` were executed between 840 and 9,495 times, per input), with one modification per program execution (requiring `quicksort` to be executed 151,444 times, so that each possible perturbation could occur, for the set of 20 inputs used).

Figure 6.22 shows violin plots for the likelihood that an add-one perturbation has no impact on the output of a program; not all expressions contained in the programs were perturbed, so a violin plot is a visual artefact.

Studies[356] of the impact of soft errors (i.e., radiation induced bit-flips) have found that over 80% of bit-flips have no detectable impact on program behavior.

6.3.3 Remaining faults: closed populations

In a closed population no coding mistakes are added (e.g., no new code is added) or removed (i.e., reported faults are not fixed), and the characteristics of the input distribution remain unchanged.

After N distinct faults have been experienced, what is the probability that there exists new, previously unexperienced, faults?

Data on reported faults commonly takes two forms: incidence data (i.e., a record of the date of first report, with no information on subsequent reports involving the same fault experience), and abundance data, i.e., a record of every fault experience.

Software reliability growth has often been modeled as a nonhomogeneous Poisson process, with researchers fitting various formulae to small amounts of incidence data.[1127] Unfortunately, it is not possible to use one sample of incidence data to distinguish between

[vii] Working out which process corresponds to which exponential appearing in the plots is left as an exercise to the reader (because your author has no idea).

different exponential order growth models,[1283] i.e., this data does not contain enough information to do the job asked of it. It is often possible to fit a variety of equations to fault report data, using regression modeling: however, predictions about future fault experiences made using these models is likely to be very unreliable; see fig 11.50.

When abundance data is available, the modeling approach discussed in section 4.3.2 can be used to estimate the number of unique items within a population, and the number of new unique items likely to be encountered with additional sampling.

A study by Kaminsky, Eddington and Cecchetti[966] investigated crash faults in three releases of Microsoft Office and OpenOffice (plus other common document processors), produced using fuzzing. Figure 6.23 shows actual and predicted growth in crash fault experiences in the 2003, 2007 and 2010 releases of Microsoft Office, along with 95% confidence intervals. Later versions are estimated to contain fewer crash faults, although the confidence interval for the 2010 release is wide enough to encompass the 2007 rate.

Figure 6.24 shows the number of duplicate crashes experienced when the same fuzzed files were processed by the 2003, 2007 and 2010 releases of Microsoft Office. The blue/purple lines are the two components of fitted biexponential models for the three curves.

The previous analysis is based on information about faults that have been experienced. What is the likelihood of a fault experience, given that no faults have been experienced in the immediately previous time, T?

An analysis by Bishop and Bloomfield[202] derived a lower bound for the reliability function, R, for a program executing without experiencing a fault for time t, after it has executed for time T without failure; it is assumed that the input profile does not change during time $T+t$. The reliability function is:

$$R(t|T) \geq 1 - \frac{t}{T+t} e^{-\frac{T}{t}\log(1+\frac{t}{T})}$$

If t is much smaller than T, this equation can be simplified to: $R(t|T) \geq 1 - \frac{t}{(T+t)\times e}$

For instance, if a program is required to execute for 10 hours with reliability 0.9999, the initial failure free period, in hours, is:

$$0.9999 \geq 1 - \frac{10}{(T+10)\times e}$$

$$T \geq \frac{10}{(1-0.9999)\times e} - 10 \approx 36,778$$

If T is much smaller than t, the general solution can be simplified to: $R(t|T) \geq \frac{T}{t}$

How can this worst case analysis be improved on?

Assuming a system executes N times without a failure, and has a fixed probability of failing, p, the probability of one or more failures occurring in N executions is given by:

$$C = \sum_{n=1}^{N} p(1-p)^{n-1} = p\frac{1-(1-p)^N}{1-(1-p)} = 1-(1-p)^N$$

How many executions, without failure, need to occur to have a given confidence that the actual failure rate is below a specified level? Rearranging, gives: $N = \left\lceil \frac{\log(1-C)}{\log(1-p)} \right\rceil$

Plugging in values for confidence, $C = 0.99$, and failure probability, $p < 10^{-4}$, then the system has to execute without failure for 46,050 consecutive runs.

This analysis is not realistic because it assumes that the probability of failure, p, remains constant for all input cases; studies show that p can vary by several orders of magnitude.

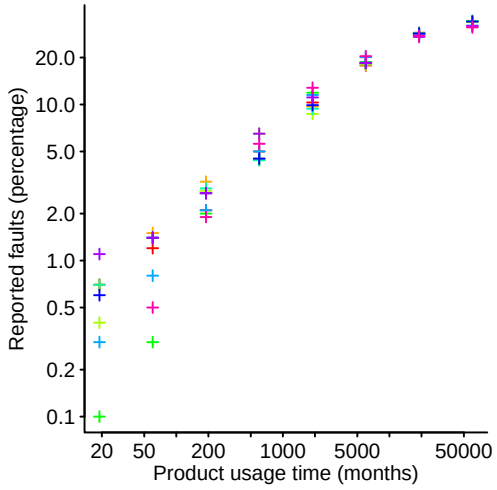


Figure 6.20: Percentage of fault experiences having a given mean time to first experience (in months, over all installations of a product), for nine products. Data from Adams.[7] Github–Local

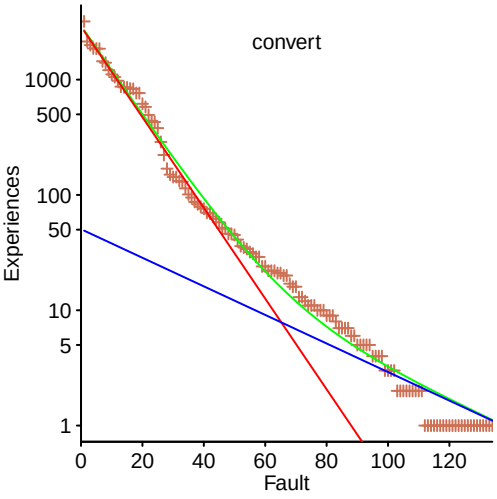


Figure 6.21: Number of times the same fault was experienced in one program, crashes traced to the same program location; with fitted biexponential equation (green line; red/blue lines the two components). Data kindly provided by Zhao.[2015] Github–Local

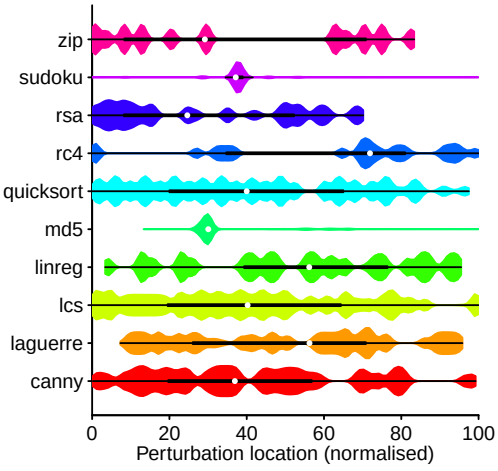


Figure 6.22: Violin plots of likelihood (local y-axis) that an add-one perturbation at a (normalised) program location will not change the output behavior. Data from Danglot et al.[435] Github–Local

6.3.4 Remaining faults: open populations

In an evolving system, existing coding mistakes are corrected and new ones are made; new features may be added that interact with existing functionality (i.e., there may be a change of behavior in the code executed for the same input), and the user base is changing (e.g., new users arrive, existing users leave, and the number of users running a particular version changes as people migrate to a newer release); the population of mistakes is open.

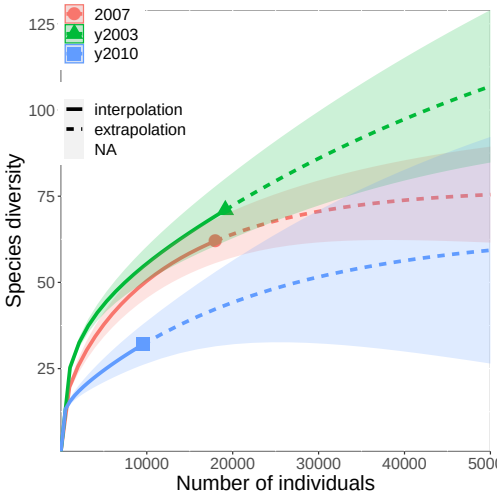


Figure 6.23: Predicted growth, with 95% confidence intervals, in the number of new crash fault experiences in the 2003, 2007 and 2010 releases of Microsoft Office. Data from Kaminsky et al.[966] Github–Local

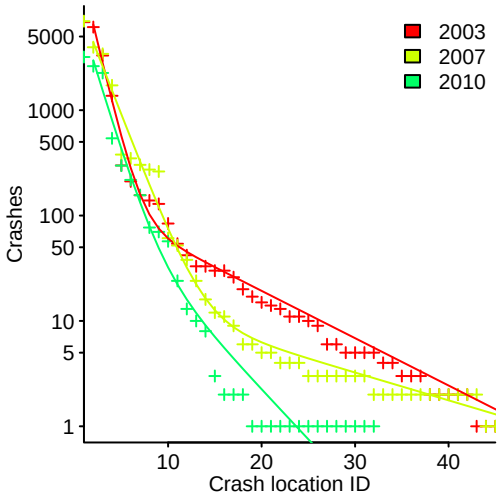


Figure 6.24: Number of crashes traced to the same executable location (sorted by number of crashes), in the 2003, 2007 and 2010 releases of Microsoft Office; lines are fitted biexponential regression models. Data from Kaminsky et al.[966] Github–Local

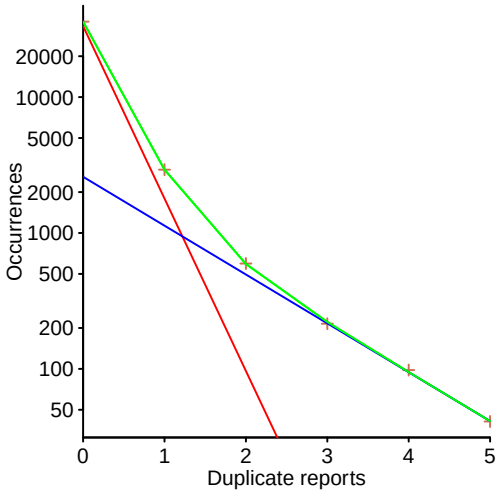


Figure 6.25: Number of occurrences of the same mistake responsible for a reported fault in GCC, with fitted biexponential regression model, and component exponentials. Data from Sun et al.[1793] Github–Local

Studies of fault reports in an open population that fail to take into account the impact of the time varying population[816] will not produce reliable results.

Section 4.3.2.2 discusses estimation in open populations.

What form of regression models can be fitted to data on fault reports from an open population?

A study by Sun, Le, Zhang and Su[1793] investigated the fault reports for GCC and LLVM. Figure 6.25 shows the number of times a distinct mistake has been responsible for a fault report in GCC (from 1999 to 2015), with a fitted biexponential, and its component exponentials.

A study by Sadat, Bener and Miranskyy[1625] investigated duplicate fault reports in Apache, Eclipse and KDE over 18-years. Figure 6.26 shows the number of times distinct faults reported in KDE, with a fitted triexponential (green) and the three component exponentials.

Being able to fit this form of model suggests a pattern that may occur in other collections of reported faults, but there is no underlying theory offering an explanation for the pattern seen.

Successive releases of a software system often include a large percentage of code from earlier releases. The collection of source code that is new in each release can be treated as a distinct population containing a fixed number of mistakes; these populations do not grow but can shrink (when code is deleted). The code contained in the first release is the foundation population.

If the only changes made to code are fixes of mistakes, the number of faults experiences caused by that code should decrease (assuming the input profile and number of users does not change).

A study by Massacci, Neuhaus and Nguyen[1217] investigated 899 security advisories in Firefox, reported against six major releases. Their raw data is only available under an agreement that does not permit your author to directly distribute it to readers; the data used in the following analysis was reverse engineered from the paper, or extracted by your author from other sources.

Table 6.2 shows the earliest version (columns) of Firefox containing a known mistake in the source code, and the latest version (rows) to which a corresponding fault report exists. For instance, 42 faults were discovered in version 2.0 corresponding to mistakes made in the source code written for version 1.0. Only corrected coding mistakes have been counted, unfixed mistakes are not included in the analysis. Each version of Firefox has a release, and retirement date, after which the version is no longer supported, i.e., coding mistakes are not corrected in retired versions.

	1.0	**1.5**	**2.0**	**3.0**	**3.5**	**3.6**
1.0	79					
1.5	71	108				
2.0	42	104	126			
3.0	97	15	22	67		
3.5	32			30	32	
3.6	13		1	5	41	14

Table 6.2: Number of reported security advisories in versions of Firefox; coding mistake made in version columns, advisory reported in version row. Data from Massacci et al.[1217]

The 1.0 column in table 6.2 shows that 97 security advisories reported in Firefox release 3.0 were fixed by changing code written for release 1.0, while 42 were reported in release 2.0. Is this increase in version 3.0 due to an increase in end-user usage, a change of input profile, or some other reason?

The Massacci study includes a break-down of the amount of source code contained in each released version of Firefox by the version in which it was first released.

The number of Firefox users can be estimated from the claimed internet usage over time (see Github–faults/internet-population.R), and Firefox market share. Figure 6.28 shows the market share of the six versions of Firefox between their official release and end-of-support dates. Estimated values appear to the left of the vertical grey line, values from measurements to the right; note: at their end-of-support dates, post 1.5 versions had a significant market share.

The end-user usage of source code originally written for a particular version of Firefox, over time, is calculated as follows (data is been binned by week): *number of lines of code originally written for a particular version contained within the code used to build a later version, or that particular version* (call this the build version) multiplied by *the market share of the build version* multiplied by *the number of Internet users* (based on the developed world user count); the units are $LOC \times Users$

Figure 6.29 shows the amount of end-user usage of the source code originally written for Firefox version 1.0. The yellow line is the code usage for version 1.0 code executing in Firefox build version 1.0, the green line the code usage for version 1.0 code executing in build version 1.5 and so on. The red points show version 1.0 code usage summed over all build versions.

Comparing end-user usage of version 1.0 source code in Firefox versions 2.0 and 3.0 (see figure 6.29) shows significantly greater usage in version 2.0, i.e., quantity of end-user usage is not the factor responsible for the increase in version 1.0 sourced security vulnerabilities seen in the 1.0 column of table 6.2.

Other possible reasons for the increase include a significant change in the input profile caused by changes in webpage content, starting around mid-2008, or an increase in researchers actively searching for vulnerabilities in browsers.

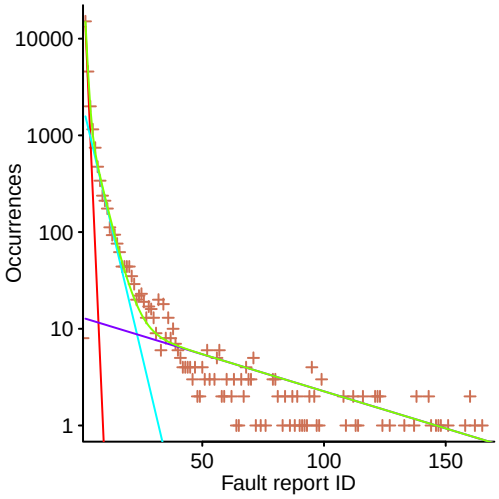


Figure 6.26: Number of instances of the same reported fault in KDE, with fitted triexponential regression model. Data from Sadat et al.[1625] Github–Local

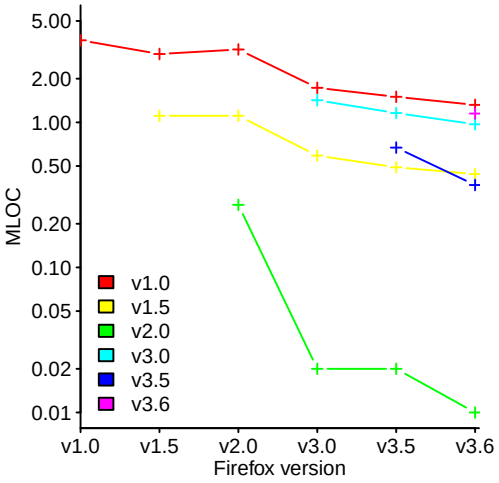


Figure 6.27: Lines of source in early versions of Firefox, broken down by the version in which it first appears. Data extracted from Massacci et al.[1217] Github–Local

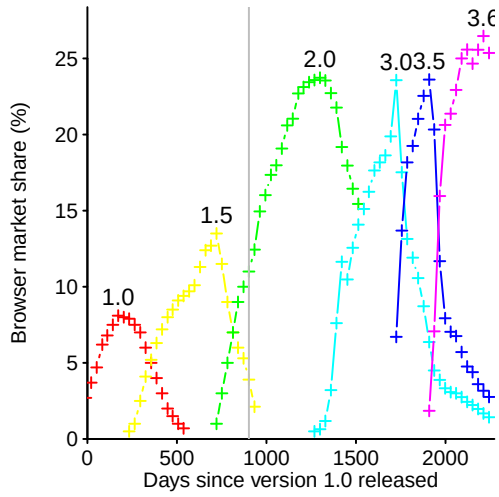


Figure 6.28: Market share of Firefox versions between official release and end-of-support (left of grey line are estimates, right are measurements). Data from Jones.[937] Github–Local

6.4 Where is the mistake?

Information about where mistakes are likely to be made can be used to focus problem-solving resources in those areas likely to produce the greatest returns. A few studies[424] have measured across top-level entities such as project phase (e.g., requirements, coding, testing, documentation), while others have investigated specific components (e.g., source code, configuration file), or low level constructs, e.g., floating-point.[490]

The root cause of a mistake, made by a person, may be knowledge based (e.g., lack of knowledge about the semantics of the programming language used), rule based (e.g., failure to correctly apply known coding rules), or skill based (e.g., fail to copy the correct value of a numeric constant in an assignment).[1566]

Mistakes in hardware[342, 505, 892] tend to occur much less frequently than mistakes in software, and mistakes in hardware are not considered here[viii]

The *user interface*, the interaction between people and an application, can be a source of fault experiences in the sense that a user misinterprets correct output, or selects a parameter option that produces unintended program behavior. User interface issues are not considered here.

Accidents where a large loss occurs (e.g., fatalities) are often followed by an accident investigation. The final report produced by the investigation may involve language biases that affect what has been written, and how it may be interpreted.[1892]

A study by Brown and Altadmri[268] investigated coding mistakes made by students, and the beliefs their professors had about common student mistakes; the data came from 100 million compilations across 10 million programming sessions using Blackbox (a Java programming environment). There was very poor agreement between professor beliefs and the actual ranked frequency of student mistakes; see Github–reliability/educators.R.

One study[1378] found that 36% of mistakes logged during development were made in phases that came before coding (Team Software Process was used, and many of the mistakes may have been minor); see Github–reliability/2018_005_defects.R.

Software supply chains are a target for criminals seeking to infect computers with malicious software;[1409] an infected software update may be downloaded and executed by millions of users

6.4.1 Requirements

The same situation can be approached from multiple viewpoints, depending on the role of the viewer; see fig 2.49. Those implementing a system may fail to fully appreciate all the requirements implied by the specification; context is important, see fig 2.48.

[viii] Your author once worked on a compiler for a cpu that was still on alpha release; the generated code was processed by a sed script to handle known problems in the implementation of the instruction set, problems that changed over time as updated versions of the cpu chip became available.

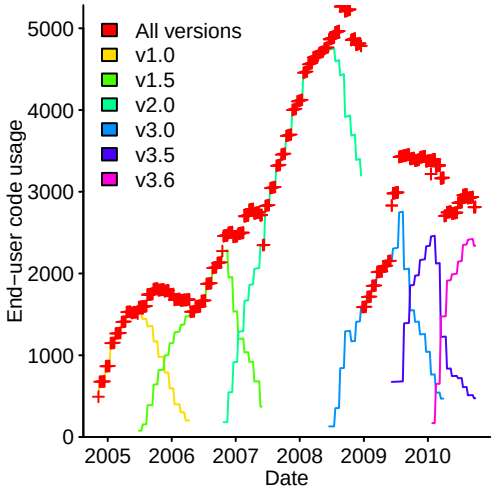


Figure 6.29: End-user usage of code originally written for Firefox version 1.0, by major released versions (in units of LOC*Users); red points show sum over all versions. Based on data from Jones[937] and extracted from Massacci et al.[1217] Github–Local

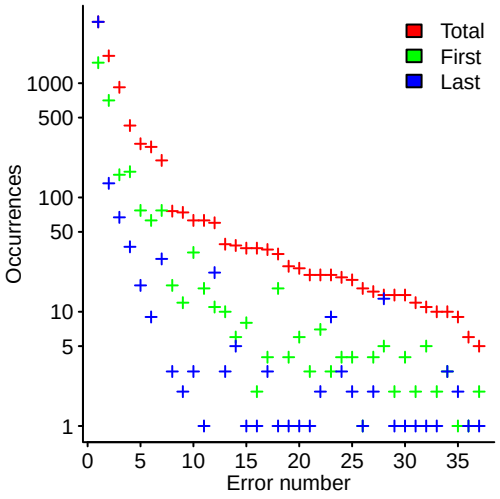


Figure 6.30: Total number of implementations in each of 36 equivalence classes, plus both first and last competitor submissions. Data from van der Meulen et al.[1877] Github–Local

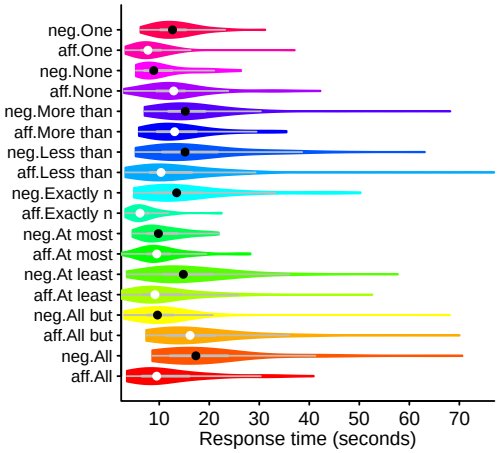


Figure 6.31: Violin plot of the time taken to response to a question about a requirement, for nine quantifiers paired by affirmative/negative. Data from Winter et al.[1971] Github–Local

A requirements mistake is made when one or more requirements are incorrect, inconsistent or incomplete; an ambiguous specification[189] contains potential mistakes. The number of mistakes contained in requirements may be of the same order of magnitude,[424] or exceed, the number of mistakes found in the code;[1551] different people bring different perspectives to requirements analysis, which can result in them discovering different problems.[1097]

Software systems are implemented by generating and interpreting language (human and programming). Reliability is affected by human variability in the use of language,[192] what individuals consider to be correct English syntax,[1752] and the interpretation of numeric phrases. Language issues are discussed in section 6.1.4 and section 6.4.2.

During the lifetime of a project, existing requirements are misinterpreted or changed, and new requirements are added.

Non-requirement mistakes may be corrected by modifying the requirements; see fig 8.28. In cases where a variety of behaviors are considered acceptable, modifying the requirements documents may be the most cost effective path to resolving a mistake.

A study by van der Meulen, Bishop and Revilla[1877] investigated the coding mistakes made in 29,000 implementations of the $3n+1$ problem (the programs had been submitted to a programming contest). All submitted implementations were tested, and programs producing identical outputs were assigned to the same equivalence class (competitors could make multiple submissions, if the first failed to pass all the tests). In many cases the incorrect output, for an equivalence class, could be explained by a failure of the competitor to implement a requirement implied by the problem being solved, e.g., failing to swap input number pairs, when the first was larger than the second.

Figure 6.30 shows the 36 equivalence classes containing the most members; the most common is the correct output, followed by always returning 0 (zero).

Studies[358] have found that people take longer to answer question involving a negation, and are less likely to give a correct answer.

A study by Winter, Femmer and Vogelsang[1971] investigated subject performance on requirements expressed using affirmative and negative quantifiers. Subjects saw affirmative (e.g., "All registered machines must be provided in the database.") and negative (e.g., "No deficit of a machine is not provided in the database.") requirements, and had to decide which of three situations matched the sentence. Affirmative wording had a greater percentage of correct answers in four of the nine quantifier combinations (negative wording had a higher percentage of correct answers for the quantifiers: `All but` and `More than`).

Figure 6.31 shows the response time for each quantifier, broken down by affirmative/negative. Average response time for negative requirements was faster for two, of the nine, quantifiers: `All but` and `None` (there was no statistical difference for `At most`).

The minimum requirements for some software (e.g., C and C++ compilers) is specified in an ISO Standard. The ISO process requires that the committee responsible for a standard maintain a log of potential defect submissions received, along with the committee's response. Figure 6.32 shows the growth of various kinds of defects reported against the POSIX standard.[899]

There have been very few studies[1692] of the impact of the form of specification on its implementation.

Two studies[363,926] have investigated the requirements' coverage achieved by two compiler validation suites.

Source code is written to implement a requirement, or to provide support for code that implements requirements. An `if-statement` represents a decision and each of these decisions should be traceable to a requirement or an internal housekeeping requirement. A project by Jones and Corfield[927] cross-referenced the `if-statements` in the source of a C compiler to every line in the 1990 version of the C Standard.[1637] Of the 53 files containing references to either the C Standard or internal documentation, 13 did not contain any references to the C Standard (for the 53 files the average number of references to the C Standard was 46.6). The average number of references per page of the language chapter of the Standard was approximately 14. For more details see Github–projects/Model-C/.

6.4.2 Source code

Source code is the focus of much software engineering research: it is the original form of an executable program that contains the mistakes that can lead to fault experiences, and

is usually what is modified by developers to correct a reported fault. From the research perspective it is now available in bulk, and techniques for analysing it are known, and practical to implement.

There are recurring patterns in the changes made to source code to correct mistakes,[1443] one reason for this is that some language constructs are used much more often than others.[930] The idea that there is an association between fault reports and particular usage patterns in source code, or program behavior, is popular; over 40 association measures have been proposed.[1167]

Errors of omission can cause faults to be experienced. One study[756] of error handling by Linux file systems found that possible returned error codes were not checked for 13% of function calls, i.e., the occurrence of an error was not handled. Cut-and-paste is a code editing technique that is susceptible to errors of omission, that is, failing to make all the necessary modifications to the pasted version;[169] significant numbers of cut-and-paste errors have been found in JavaDoc documentation.[1428]

Common patterns of mistakes are also seen in the use of programming language syntax and semantics. Figure 6.33 shows ranked occurrences of each kind of compiler message generated by Java and Python programs, submitted by students.

Proponents of particular languages sometimes claim that programs written in the language are more reliable (other desirable characteristics may also be claimed), than if written in other languages. Most experimental studies comparing the reliability of programs written in different languages have either used students,[691] or had low statistical power. A language may lack a feature that, if available and used, would help to reduce the number of mistakes made by developers, e.g., support for function prototypes in C,[1672] which were added to the language in the first ANSI Standard.

To what extent might programs written in some languages be more likely to appear to behave as expected, despite containing mistakes?

A study by Spinellis, Karakoidas and Lourida[1749] made various kinds of small random changes to 14 different small programs, each implemented in 10 different languages (400 random changes per program/language pair). The ability of these modified programs to compile, execute and produce the same output as the unmodified program was recorded.

Figure 6.34 shows the fraction of programs that compiled, executed and produced correct output, for the various languages. There appear to be two distinct language groupings, each having similar successful compilation rates; one commonality of languages in each group is requiring, or not, variables to be declared before use. One fitted regression model (see Github–reliability/fuzzer/fuzzer-mod.R) contains an interaction between language and program (the problems implemented did not require many lines of code, and in some cases could be solved in a single line in some languages), and a logarithmic dependency on program length, i.e., number of lines.

Other studies[146] have investigated transformations that modify a program without modifying its behavior.

A study by Aman, Amasaki, Yokogawa and Kawahara[49] investigated time-to-bug-fix events, in the source files of 50 projects implemented in Java and 50 in C++. The survival time of files (i.e., time to fault report causing the source to be modified) was the same for both languages, and number of developers, and number of project files; they had almost zero impact on a Cox model fitted to the data; see Github–survival/Profes2017-aman.R.

Various metrics have been proposed as measures of some desirable, or undesirable, characteristic of a unit of code, e.g., a function. Halstead's and McCabe's cyclomatic complexity are perhaps the most well-known such metrics (see section 7.1.4), both count the source contained within a single function. Irrespective of whether these metrics strongly correlate with anything other than lines of code,[1077] they can be easily manipulated by splitting functions with high values into two or more functions, each having lower metric values (just as it is possible to reduce the number of lines of code in a function, by putting all the code on one line).

The value of McCabe's complexity (number of decisions, plus one) for the following function is 5, and there are 16 possible paths through the function:

```
int main(void)
{
if (W) a(); else b();
if (X) c(); else d();
```

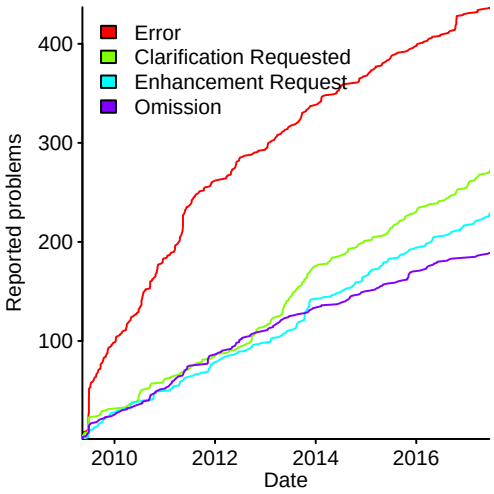


Figure 6.32: Cumulative number of potential defects logged against the POSIX standard, by defect classification. Data kindly provided by Josey.[1417] Github–Local

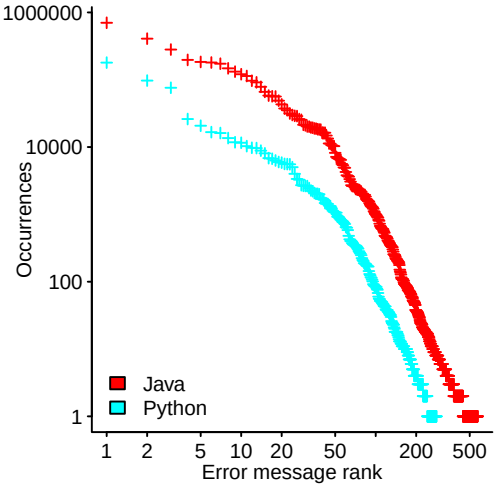


Figure 6.33: Ranked occurrences of compiler messages generated by student submitted Java and Python programs. Data from Pritchard.[1531] Github–Local

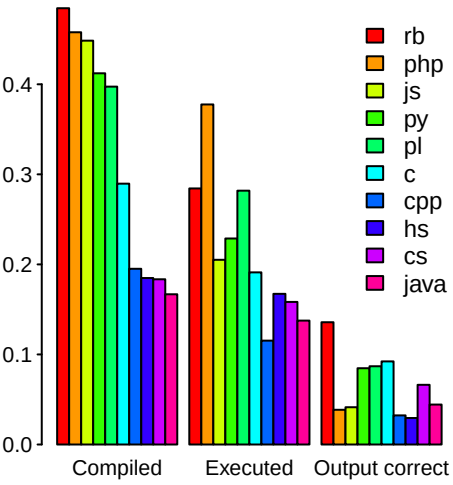


Figure 6.34: Fraction of mutated programs, in various languages, that successfully compiled/executed/produced the same output. Data from Spinellis et al.[1749] Github–Local

```
if (Y) e(); else f();
if (Z) g(); else h();
}
```

each `if...else` contains two paths and there are four in series, giving $2 \times 2 \times 2 \times 2$ paths. Restructuring the code, as below, removes the multiplication of paths caused by the sequences of `if...else`:

```
void a_b(void)
                {if (W) a(); else b();}
void c_d(void)
                {if (X) c(); else d();}
void e_f(void)
                {if (Y) e(); else f();}
void g_h(void)
                {if (Z) g(); else h();}

int main(void)
{
a_b();
c_d();
e_f();
g_h();
}
```

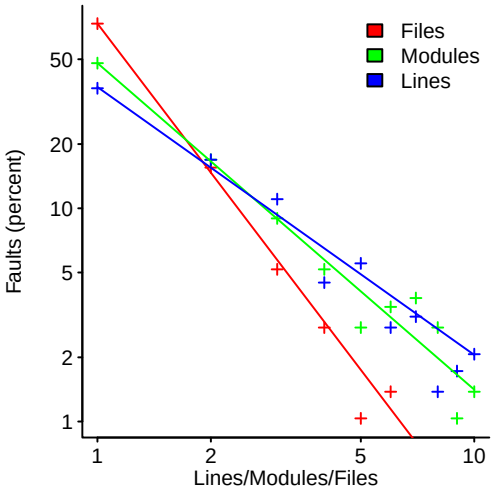


Figure 6.35: Number of fault reports whose fixes involved a given number of files, modules or lines in a sample of 290 faults in AspectJ; lines are fitted power laws. Data from Lucia.[1166] Github–Local

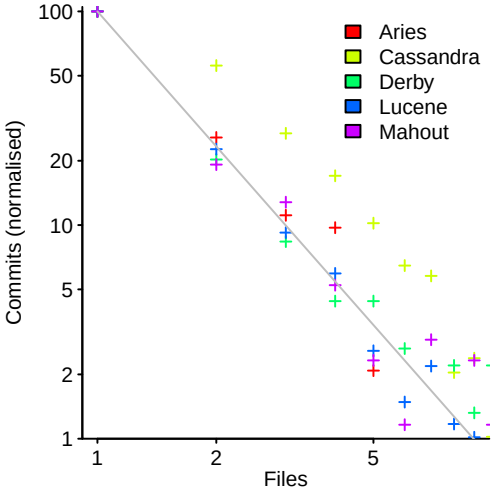


Figure 6.36: Normalized number of commits (i.e., each maximum is 100), made to address fault reports, involving a given number of files in five software systems; grey line is representative of regression models fitted to each project, and has the form: $Commits \propto Files^{-2.1}$. Data from Zhong et al[2019] via M. Monperrus. Github–Local

reducing the McCabe complexity of `main` to 1, with the four new functions each having a McCabe complexity of two. Where has the complexity that `main` once had, gone? It now *exists* in the relationship between the functions, a relationship that is not included in the McCabe complexity calculation; the number of paths that can be traversed, by a call to `main` at runtime, has not changed, but a function based count now reports one path.

A metric that assigns a value to individual functions (i.e., its value is calculated from the contents of single functions) cannot be used as a control mechanism (i.e., require that values not exceed some limit), because its value can be easily manipulated by moving contents into newly created functions. The software equivalent of what is known as *accounting fraud* in accounting.

Predictions are sometimes attempted[1696, 2029] at the file level of granularity, e.g., predicting which files are more likely to be the root cause of fault experiences; the idea being that the contents of highly ranked files be rewritten. Any reimplementation will include mistakes, and the cost of rewriting the code may be larger than handling the fault reports in the original code, as they are discovered.

The idea that there is an optimal value for the number of lines of code in a function body has been an enduring meme (when object-oriented programming became popular, the meme mutated to cover optimal class size). See fig 8.39 for a discussion of the U-shaped defect density paper chase; other studies[1037] investigating the relationship between reported fault experiences and number of lines of code, have failed to include program usage information (i.e., number of people using the software) in the model.

The fixes for most user fault reports involve changing a few lines in a single function, and these changes occur within a single file. A study[713] of over 1,000 projects for each of C, Java, Python and Haskell found that correcting most coding mistakes involved adding and deleting a few tokens.

A study by Lucia[1166] investigated fault localization techniques. Figure 6.35 shows the percentage of fault reports whose correction involved a given number of files, modules or lines; lines are power laws fitted using regression.

A study by Zhong and Su[2019] investigated commits associated with fault reports in five large Open source projects. Figure 6.36 shows the number of files modified while fixing reported faults, against normalized (i.e., each maximum is 100) number of commits made while making these changes.

When existing code is changed, there is a non-zero probability of a mistake being made.

A study by Purushothaman and Perry[1536] investigated small source code changes made to one subsystem of the 5ESS telephone switch software (4,550 files containing almost 2MLOC, with 31,884 modification requests after the first release changing 4,293 of these files). Figure 6.37 is based on an analysis of fault reports traced to updates, that involved

modifying/inserting a given number of lines, and shows the percentage of each kind of modification that eventually led to a fault report.

Using configuration files to hold literal values that would otherwise need to be present in the source code can greatly increase runtime flexibility. Mistakes in the use of configuration options can lead to fault experiences; see Github–reliability/fse15.R.

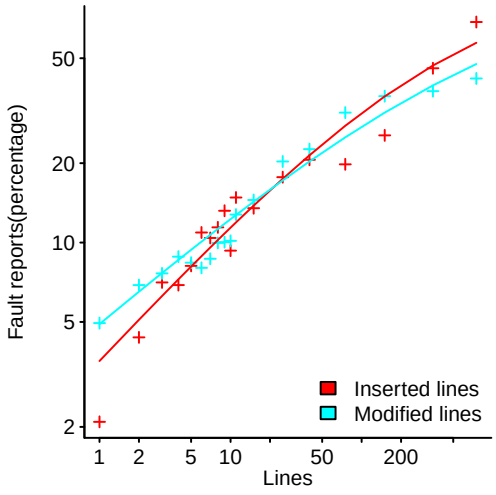


Figure 6.37: Percentage of insertions/modifications of a given number of lines resulting in a reported fault; lines are fitted beta regression models of the form: $percent\_faultReports \propto \log(Lines)$. Data from Purushothaman et al.[1536] Github–Local

6.4.3 Libraries and tools

Libraries provide functionality believed to be of use to many programs, or that is difficult to correctly implement without specialist knowledge, e.g., mathematical functions. Many language specifications include a list of functions that conforming implementations are required to support.

The implementation of some libraries requires domain specific expertise, e.g., the handling of branch cuts in some maths functions.[1700] Some library implementations may contain subtle, hard to detect mistakes: for instance, random number generators may generate sequences containing patterns[1101] that create spurious correlations in the results; even widely used applications can suffer from this problem, e.g., Microsoft Excel.[1236]

The availability of many open source libraries can make it more cost effective to use third-party code, rather than implementing a bespoke solution.

A study by Decan, Mens and Constantinou[470] investigated the time taken for security vulnerabilities in packages hosted in the `npm` repository to be fixed, along with the time taken to update packages that depended on a version of a package having a reported vulnerability. Figure 6.38 shows survival curves (with 95% confidence bounds), for high and medium severity vulnerabilities, of time to fix a reported vulnerability (Base), and time to update a dependency (Depend) to a corrected version of a package.

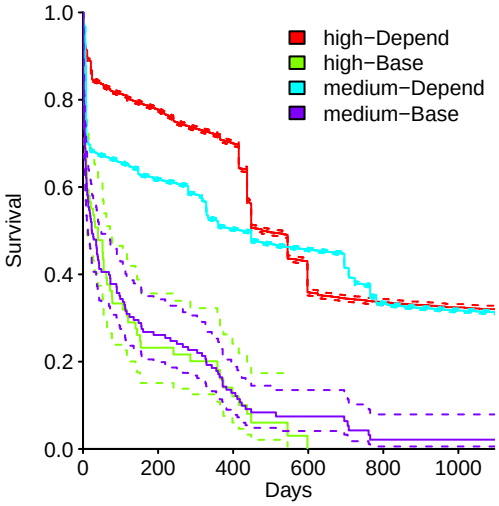


Figure 6.38: Survival curve (with 95% confidence bounds) of time to fix vulnerabilities reported in `npm` packages (Base) and time to update a package dependency (Depend) to a corrected version (i.e., not containing the reported vulnerability); for vulnerabilities with severity high and medium. Data from Decan et al.[470] Github–Local

The mistake that leads to a fault experience may be in the environment in which a program is built and executed. Many library package managers support the installation of new packages via a command line tool. One study[1849] made use of typos in the information given to command line package managers to cause a package other than the one intended to be installed.

Compilers, interpreters and linkers are programs, and contain mistakes (e.g., see fig 6.25), and the language specification may also be inconsistent or under specified, e.g., ML.[961]

Different support tools may produce different results, e.g., statement coverage,[1990] and call graph construction; see fig 13.1.

6.4.4 Documentation

Documentation is a cost paid today, that is intended to provide a benefit later for somebody else, or the author.

If user documentation specifies functionality that is not supported by the software, the vendor may be liable to pay damages to customers expecting to be able to perform the documented functionality.[972] Non-existent documentation is not unreliable, but documentation that has not been updated to match changes to the software is.

There have been relatively few studies of the reliability of documentation (teachers of English as a foreign language study the language usage mistakes of learners[257]). A study by Rubio-Gonzalez and Libit[1618] investigated the source code of 52 Linux file systems, which invoked 42 different system calls and returned 30 different system error codes. The 871 KLOC contained 1,784 instances of undocumented error return codes; see Github–projects/err-code_mismatch.R. A study by Ma, Liu and Forin[1179] tested an Intel x86 cpu emulator and found a wide variety of errors in the documentation specifying the behavior of the processor.

The Microsoft Server protocol documents[1277] sometimes specify that the possible error values returned by an API function are listed in the Windows error codes document, which lists over 2,500 error codes.

6.5 Non-software causes of unreliability

Hardware contains moving parts that wear out. Electronic components operate by influencing the direction of movement of electrons; when the movement is primarily in one direction atoms migrate in that direction, and over time this migration degrades device operating characteristics.[1755] Fabricating devices with smaller transistors decreases mean time to failure; expected chip lifetimes have dropped from 10 years to 7, and continue to decrease.[1947]

As the size of components shrinks, and the number of components on a device increases, the probability that thermal noise will cause a bit to change state increases.[1011]

Faulty hardware does not always noticeably change the behavior of an executing program; apparently correct program execution can occur in the presence of incorrect hardware operation, e.g., image processing.[1392] Section 6.3.2 discusses studies showing that many mistakes have no observable impact on program behavior.

For a discussion of system failure traced to either cpu or DRAM failures see table 10.7, and for a study investigating the correlation between hardware performance and likelihood of experiencing intermittent faults see section 10.2.3.

A software reliability problem rarely encountered outside of science fiction, a few decades ago, now regularly occurs in modern computers: cosmic rays (plus more local sources of radiation, such as the materials used to fabricate devices) flipping the value of one or more bits in memory, or a running processor. Techniques for mitigating the effects of radiation induced events have been proposed.[1339]

The two main sources of radiation are alpha-particles generated within the material used to fabricate and package devices, and Neutrons generated by Cosmic-rays interacting with the upper atmosphere. The data in figure 6.39, from a study by Autran, Semikh, Munteanu, Serre, Gasiot and Roche,[94] comes from monitoring equipment located in the French Alps; either, 1,700 m under the Fréjus mountain (i.e., all radiation is generated by the device), or on top of the Plateau de Bure at an altitude of 2,552 m (i.e., radiation sources are local and Cosmic).[93] For confidentiality reasons, the data has been scaled by a small constant.

Figure 6.39 shows how the number of bit-flips increased over time (measured in Megabits per hour), for SRAM fabricated using 130 nm, 65 nm and 40 nm processes. The 130 nm and 65 nm measurements were made underground, and the lower rate of bit-flips for the 65 nm process is the result of improved materials selection, that reduced alpha-particle emissions; the 40 nm measurements were made on top of the Plateau de Bure, and show the impact of external radiation sources.

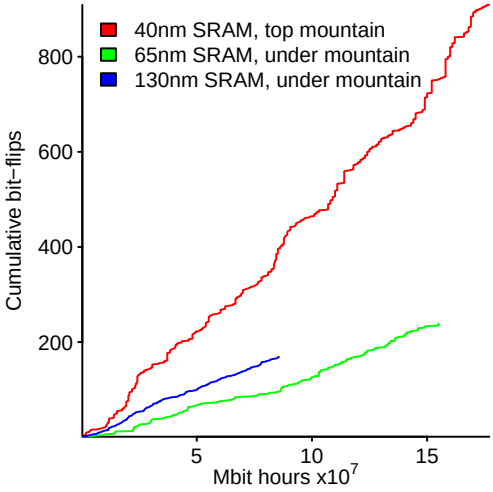


Figure 6.39: Number of bit-flips in SRAM fabricated using various processes, with devices on top of, or under a mountain in the French Alps. Data kindly provided by Autran.[94] Github–Local

The soft error rate is usually quoted in FITs (Failure in Time), with 1 FIT corresponding to 1 error per 10^9 hours per megabit, or 10^{-15} errors per bit-hour. Consider a system with 4 GB of DRAM (1000 FIT/Mb is a reasonable approximation for commodity memory,[1825] which increases with altitude, being 10 times greater in Denver, Colorado), the system has an MTBF of $1000 \times 10^{-15} \times 4.096 \times 10^9 \times 8 = 3.2 \times 10^{-2}$ hours (around once every 33 hours). Soft errors are a regular occurrence for installations containing hundreds of terabytes of memory.[880]

The Cassini spacecraft experienced an average of 280 single bit memory errors per day[1804] (in two identical flight recorders containing 2.5G of DRAM); also see fig 8.31. The rate of double-bit errors was higher than expected (between 1.5 and 4.5%) because the incoming radiation had enough energy to flip more than one bit.

Uncorrected soft errors place a limit on the maximum number of computing nodes that can be usefully used by one application; at around 50,000 nodes, a system would spend half its time saving checkpoints, and restarting from previous checkpoints after an error occurred.[1584]

Error correcting memory reduces the probability of an uncorrected error by several orders of magnitude, but with modern systems containing terabytes the probability of an error adversely affecting the result remains high.[880] The Cray Blue Waters system at the National Center for Supercomputing Applications experienced 28 uncorrected memory errors (ECC and Chipkill parity hardware checks corrected 722,526 single bit errors, and 309,359 two-bit errors, a 99.995% success rate).[491] Studies[1258] have investigated assigning variables deemed to be critical to a subset of memory that is protected with error correcting hardware, along with various other techniques.[1173]

Calculating the FIT for processors is complicated.[1128]

Redundancy can be used to continue operating after experiencing a hardware fault, e.g., three processors performing the same calculation, and a majority vote used to decide which outputs to accept.[1993] Software only redundancy techniques include having the compiler generate, for each source code sequence, two or more independent machine code sequences[1574] whose computed values are compared at various check points, and replicating computations across multiple cores[2013] (and comparing outputs). The overhead of duplicated execution can be reduced by not replicating those code sequences that are less affected by register bit flips[592] (e.g., the value returned from a bitwise AND that extracts 8 bits from a 32-bit register is 75% less likely to deliver an incorrect result than an operation that depends on all 32 bits). Optimizing for reliability can be traded off against performance,[1295] e.g., ordering register usage such that the average interval between load and last usage is reduced.[1986]

Developers don't have to rely purely on compiler or hardware support, reliability can be improved by using algorithms that are robust in the presence of *faulty* hardware. For instance, the traditional algorithms for two-process mutual exclusion are not fault tolerant; a fault tolerant mutual exclusion algorithm using $2f + 1$ variables, where a single fault may occur in up to f variables is available.[1316] Researchers are starting to investigate how best to prevent soft errors corrupting the correct behavior of various algorithms.[262]

Bombarding a system with radiation increases the likelihood of radiation induced bit-flips,[1275] and can be used for testing system robustness.

The impact of level of compiler optimization on a program's susceptibility to bitflips is discussed in section 11.2.2.

Vendor profitability is driving commodity cpu and memory chips towards cheaper and less reliable products, just like household appliances are priced low and have a short expected lifetime.[1721]

A study by Dinaburg[501] found occurrences of bit-flips in domain names appearing within HTTP requests, e.g., a page from the domain `ikamai.net` being requested rather than from `akamai.net` (the 2.10^{-9} bit error rate was thought to occur inside routers and switches). Undetected random hardware errors can be used to redirect a download to another site,[501] e.g., to cause a maliciously modified third-party library to be loaded.

If all the checksums involved in TCP/IP transmission are enabled, the theoretical error rate is 1 in 10^{17} bits; which for 1 billion users visiting Facebook on average once per day and downloading 2M bytes of Javascript per visit, gives an expected bit flip rate of once every 5 days for a single Facebook user.

6.5.1 System availability

A system is only as reliable as its least reliable critical subsystem, and the hardware on which software runs is a critical subsystem that needs to be included in any application reliability analysis; some applications also require a working internet connection, e.g., for database access.

Before cloud computing became a widely available commercial service, companies built their own clustered computer facilities (low usage rates of such systems[895] is what can make cloud providers more cost effective).

The reliability of Internet access to the services provided by other computers is currently not high enough for people to overlook the possibility that failures can occur;[185] see the example in section 10.5.4.

Long-running applications need to be able to recover from hardware failures, if they are to stand a reasonable chance of completing. A process known as *checkpointing* periodically stores the current state of every compute unit, so that when any unit fails, it is possible to restart from the last saved state, rather than restarting from the beginning. A tradeoff has to be made[1989] between frequency of checkpointing, which takes resources away from completing execution of the application but reduces the total amount of lost calculation, and infrequent checkpointing, which diverts less resources but incurs greater losses when a fault is experienced. Calculating the optimum checkpoint interval[430] requires knowing the distribution of node uptimes; see figure 6.40.

The Los Alamos National Laboratory (LANL) has made public, data from 23 different systems installed between 1996 and 2005.[1084] These systems run applications that " . . .

perform long periods (often months) of CPU computation, interrupted every few hours by a few minutes of I/O for check-pointing." Figure 6.40 shows the 10-hour binned data fitted to a zero-truncated negative binomial distribution for systems 2 and 18.

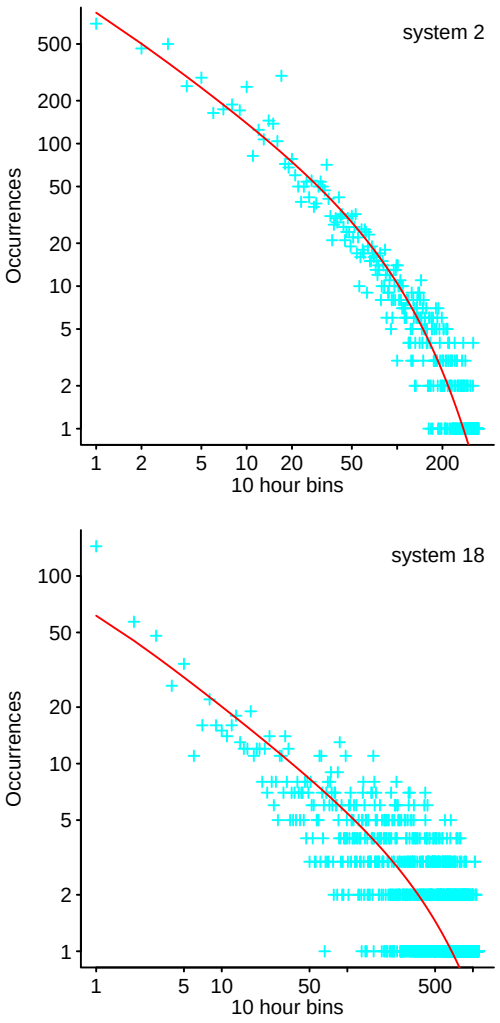


Figure 6.40: For systems 2 and 18, number of uptime intervals, binned into 10 hour intervals, red lines are both fitted negative binomial distributions. Data from Los Alamos National Lab (LANL). Github–Local

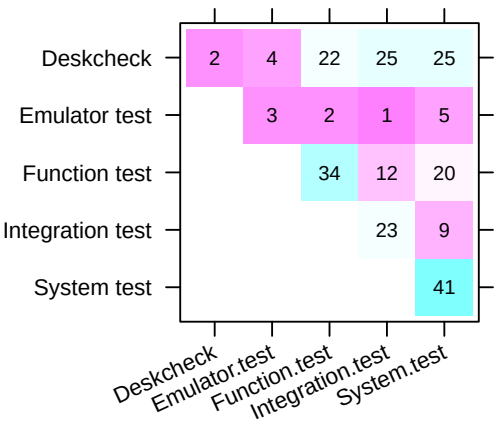


Figure 6.41: Fault slip throughs for a development project at Ericsson; y-axis lists phase when fault could have been detected, x-axis phase when fault was found. Data from Hribar et al.[864] Github–Local

Operating systems and many long-running programs sometimes write information about a variety of events to one or more log files. One study[1997] found that around 1 in 30 lines of code in Apache, Postgresql and Squid was logging code; this information was estimated to reduce median diagnosis time by a factor of 1.4 to 3. The information diversity of system event logs tends to increase, with new kinds of information being added, with the writing of older information not being switched off (because it might be useful); log files have been found to contain[1413] large amounts of low value information, more than one entry for the same event, changes caused by software updates, poor or no documentation, and inconsistent information structure within entries.

6.6 Checking for intended behavior

The two main methods for checking that code behaves as intended, are: analyzing the source code to work out what it does, and reviewing the behavior of the code during execution, e.g., testing. Almost no data is available on the kinds of mistakes found, and the relative cost-effectiveness of the various techniques used to find them.

So-called *formal proofs* of correctness are essentially a form on N-version programming, with $N = 2$. Two programs are written, with one nominated to be called the specification; one or more tools are used to analyse both programs, checking that their behavior is consistent, and sometimes other properties. Mistakes may exist in the specification program or the non-specification program.[622,668] One study[1320] of a software system that had been formally proved to be correct, found at least two mistakes per thousand lines, remained.

The further along in the development process a mistake is found, the more costly it is likely to be to correct it; possible additional costs include having to modify something created between the introduction of the mistake and its detection, and having to recheck work. This additional cost does not necessarily make it more cost effective to detect problems as early as possible. The relative cost of correcting problems vs. detecting problems, plus practical implementation issues, decide where it is most cost effective to check for mistakes during the development process.

A study by Hribar, Bogovac and Marinčić[864] investigated *Fault Slip Through* by analyzing the development phase where a fault was found compared to where it could have been found. Figure 6.41 shows the number of faults found in various test phases (deskcheck is a form of code review performed by the authors of the code), and where the fault could have been found (as specified on the fault report); also see Antolić.[66]

The cost of correcting problems will depend on the cost characteristics of the system containing the software; developing software for a coffee vending machine is likely to be a lot cheaper than for a jet fighter, because of, for instance, the cost of the hardware needed for testing. Data from NASA and the US Department of Defense, on the relative costs of fixing problems discovered during various phases of development are large, because of the very high cost of the hardware running the software systems developed for these organizations.

To reduce time and costs, the checking process may be organized by level of abstraction, starting with basic units of code (or functionality), and progressively encompassing more of the same, e.g., unit testing is performed by individual developers, integration testing checks that multiple components or subsystems work together, and systems testing is performed on the system as a whole.

A study by Nichols, McHale, Sweeney, Snavely and Volkman[1378] investigated the economics of detecting mistakes in system development (the organizations studies all used Team Software Process). One of the organizations studied developed avionics software, which required performing manual reviews and inspections of the high-level design, design and coding phases, followed by testing.

Figure 6.42 shows the reported time taken to correct 7,095 mistakes (for one avionics project), broken down by phase introduced/corrected, against the number of major phases between its introduction and correction (x-axis). Lines are fitted exponentials, with fix times less than 1, 5 and 10-minutes excluded (72% of fixes are recorded as taking less than 10-minutes); also see fig 3.33.

Many software systems support a range of optional constructs, and support for these may be selected by build time configuration options. When checking for intended behavior, a decision has to be made on the versions of the system being checked; some systems support so many options, that checking whether all possible configurations can be built requires an unrealistic investment of resources[766] (algorithms for sampling configurations are used[1254]).

6.6.1 Code review

Traditionally a code review (other terms include *code inspection* and *walkthroughs*[631]) has involved one or more people reading another developer's code, and then meeting with the developer to discuss what they have found. These days the term is also applied to reviewing code that has been pushed to a project's version control system, to check whether it is ok to merge the changes into the main branch; with geographically disperse teams, online reviews and commenting have become a necessity.

Review meetings support a variety of functions, including: highlighting of important information between project members (i.e., ensuring that people are kept up to date with what others are doing), and uncovering potential problems before changes becomes more expensive. Detecting issues may not even be the main reason for performing code reviews,[104] keeping teams members abreast of developments and creating an environment of shared ownership of code may be considered more important.

A variety of different code review techniques have been proposed, including: Ad-hoc (no explicit support for reviewers), Checklist (reviewers work from a list of specific questions that are intended to focus attention towards common problems), Scenarios-based (each reviewer takes on a role intended to target a particular class of problems), and Perspective-based reading (reviewers are given more detailed instructions, than they are given in Scenario-based reviews, about how to read the document; see section 13.2 for an analysis). The few experimental comparisons of review techniques have found that the relative performance of the techniques is small compared to individual differences in performance.

The range of knowledge and skills needed to review requirements and design documents may mean that those involved focus on topics that are within their domain of expertise.[534] Many of the techniques used for estimating population size assume that capture sites (i.e., reviews) have equal probabilities of flagging an item; estimates based on data from meetings where reviewers have selectively read documents will be biased; see Github–reliability/eickt1992.R.

Most published results from code review studies have been based on small sample sizes. For instance, Myers,[1340] investigated the coding mistakes detected by 59 professionals, using program testing and code walkthroughs/inspections, for one PL/1 program containing 63 statements and 15 known mistakes; see Github–reliability/myers1978.R. Also, researcher often use issues-found as the metric for evaluating review meetings, in particular potential fault experiences found during code reviews. Issues found is something that is easy to measure, code is readily available, and developers to review it are likely to be more numerous than people with the skills needed to review requirements and design documents (which do not always exist, as such).

Studies where the data is available include:

- Hirao, Ihara, Ueda, Phannachitta and Matsumoto[833] investigated the impact of positive and negative code reviews on patches being merged or abandoned (for Qt and OpenStack). A logistic regression model found that for Qt positive votes were more than twice as influential, on the outcome, as negative votes, while for, OpenStack negative votes were slightly more influential; see Github–reliability/OSS2016.R.
- Porter, Siy, Mockus and Votta[1507] recorded code inspection related data from a commercial project over 18 months (staffed by six dedicated developers, and five developers who also worked on other projects). The best fitting regression model had the number of mistakes found proportional to the log of the number of lines reviewed, and the log of meeting duration; this study is discussed in section 13.2, also see fig 11.33.
- Finifter[603] investigated mistakes found and fault experiences, using manual code review and black box testing, in nine implementations of the same specification. Figure 6.43 shows the number of vulnerabilities found by the two techniques in the nine implementations; some of the difference is due to the variation in the abilities and kinds of

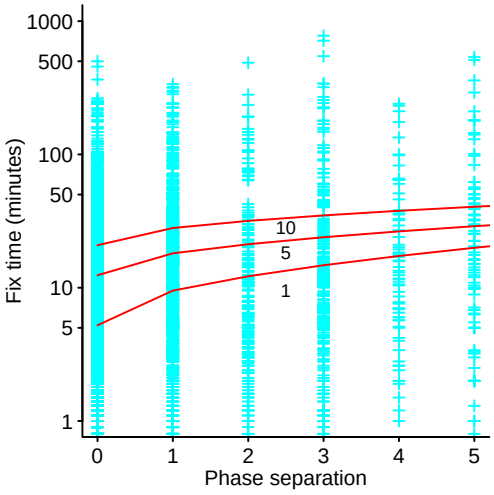


Figure 6.42: Reported time taken to correct 7,095 mistakes (in one project), broken down by phase the mistake was introduced/corrected (y-axis), against number of phases between introduction/correction (x-axis); lines are fitted regression models of the form: $Fix\_time \propto e^{\sqrt{phase\_sep}}$, with fix times less than 1, 5 and 10-minutes excluded. Data from Nichols et al.[1378] Github–Local

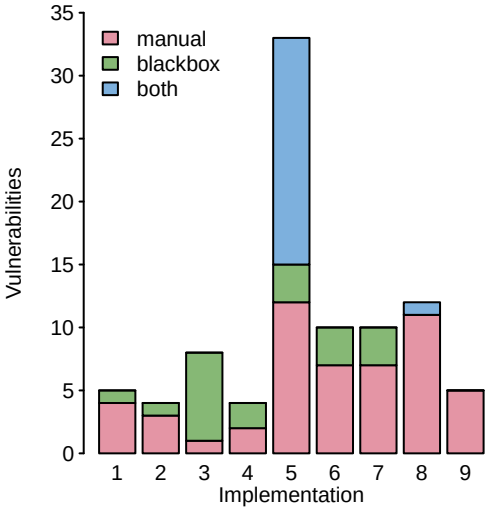


Figure 6.43: Number of vulnerabilities found using black-box testing, and manual code review of nine implementations of the same specification. Data from Finifter.[603] Github–Local

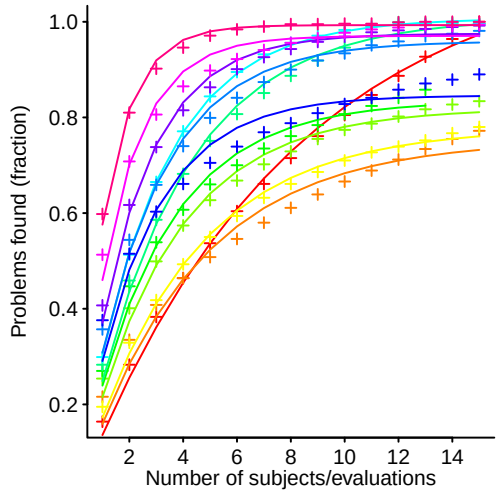


Figure 6.44: Fraction of usability problems found by a given number of subjects/evaluations in 12 system evaluations; lines are fitted regression model for each system. Data extracted from Nielsen et al.[1380] Github–Local

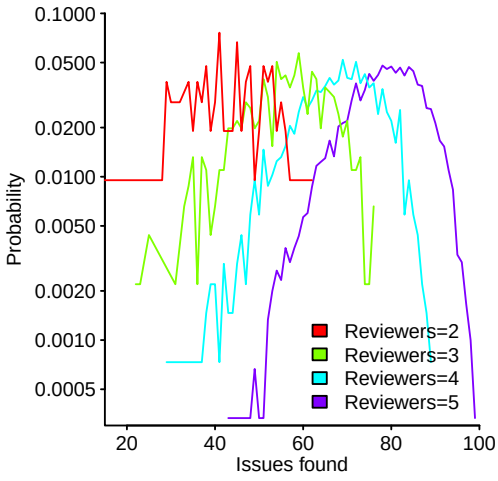


Figure 6.45: Probability (y-axis) of a given number of issues being found (x-axis), by a review group containing a given number of people (colored lines). Data from Lewis.[1123] Github–Local

mistakes made by different implementers, plus skill differences in using the programming languages.

While there has been a lot of activity applying machine learning to fault prediction, the models have not proved effective outside the data used to build them, or even between different versions of the same project;[2028] see Github–faults/eclipse/eclipse-pred.R. Noisy data is one problem, along with a lack of data on program usage; see section 6.1.3.

During a review, there is an element of chance associated with the issues noticed by individual reviewers, and some issues may only be noticed by reviewers with a particular skill or knowledge. If all reviewers have the same probability, p, of finding a problem, and there are N issues available to be found, by S reviewers, then the expected number of issues found is: $N\left[1-(1-p)^S\right]$.

A study by Nielsen and Landauer[1380] investigated the number of different usability problems discovered, as the number of subjects increased, based on data from 12 studies. Figure 6.44 shows how the number of perceived usability problems increased as the number of subjects increased; lines show the regression model fitted by the above equation (both N and p are constants returned by the fitting process).

When the probability of finding a problem varies between reviewers, there can be a wide variation in the number of problems reported by different groupings of individuals.

A study by Lewis[1123] investigated usability problem-discovery rates; the results included a list of the 145 usability problems found by 15 reviewers. How many problems are two of these reviewers likely to find, how many are three likely to find? Figure 6.45 is based on the issues found by every pair, triple (etc, up to five) of reviewers. The mean of the number of issues found increases with review group size, as does the variability of the number found. Half of all issues were only found by one reviewer, and 15% found by two reviewers.

Some coding mistakes occur sufficiently often that it can be worthwhile searching for known patterns. Ideally coding mistakes flagged by a tool are a potential cause of a fault experience (e.g., reading from an uninitialized variable), however the automated analysis performed may not be sophisticated enough to handle all possibilities[1639] (e.g., there may be some uncertainty about whether the variable being read from has been written to), or the usage may simply be suspicious (e.g., use of assignment in the conditional expression of an `if-statement`, when an equality comparison was intended, i.e., the single character = had been typed, instead of the two characters ==). The issue of how developers might respond to false positive warnings is discussed in section 9.1.

A study of one tool[2017] found a strong correlation between mistakes flagged, and faults experienced during testing, and faults reported by customers (after the output of the tool had been cleaned by a company specializing in removing false positive warnings from static analysis tool output).

6.6.2 Testing

The purpose of testing is to gain some level of confidence that software behaves in a way that is likely to be acceptable to customers. For the first release, the behavior may be specified by the implementation team (e.g., when developing a product to be sold to multiple customers), or the customer, e.g., the vendor is interested in meeting the criteria for acceptance, so they get paid. Subsequent releases usually include checks that the behavior is consistent with previous releases.

During testing, a decrease in the number of previously unseen fault experiences, per unit of test effort, is sometimes taken as an indication that the software is becoming more reliable; other reasons for a decrease in new fault experiences is replacement of existing testers by less skilled staff, or repetition of previously used input values. The extent to which reliability improves, as experienced by the customer, depends on the overlap between the input distribution used during testing, and the input distribution provided in real world use.

A study by Stikkel[1782] investigated three industrial development projects. Figure 6.46 shows the number of faults discovered per man-hour of testing, averaged over a week, for these projects (each normalised to sum to 100). The sharp decline in new fault experiences may be due to there being few mistakes remaining, a winding down of investment in the closing weeks of testing (i.e., rerunning the same tests with the same input), or some other behavior.

An example of how the input used for random testing can be unrepresentative of customer input is provided by a study[367] that performed random testing of the Eiffel base library. The functions in this library contain extensive pre/post condition checks, and random testing found twice as many mistakes in these checks as the implementation of the functionality; the opposite of the pattern seen in user fault reports.

To what extent are fault experiences generated by fuzzers representative of faults experienced by users of the software?

A study by Marcozzi, Tang, Donaldson and Cadar[1204] investigated the extent to which fault experiences obtained using automated techniques are representative of the fault experiences encountered by code written by developers. The source code involved in the fixes of 45 reported faults in the LLVM compiler were instrumented to log when the code was executed, and when the condition needed to trigger the fault experience occurred; the following is an example of instrumented code:

```
warn ("Fixing patch reached");
if (Not.isPowerOf2()) {
   if (!(C-> getValue().isPowerOf2()  // Check needed to fix fault
         && Not != C->getValue()))
      {
      warn("Fault possibly triggered");
      }
   else { /* CODE TRANSFORMATION */ } } // Original, unfixed code
```

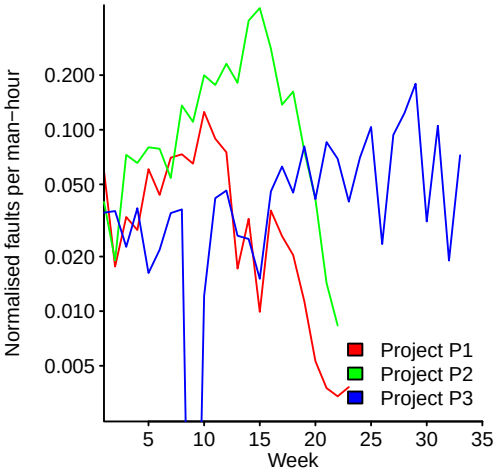


Figure 6.46: Number of faults experienced per unit of testing effort, over a given number of weeks (each normalised to sum to 100). Data from Stikkel.[1782] Github–Local

The instrumented compiler was used to build 309 Debian packages (around 10 million lines of C/C++), producing possibly miscompiled versions of the packages; the build process included running each package's test suite. A package built from miscompiled code may successfully pass its test suite.

A bitwise compare of the program executables generated by the unfixed and fixed compilers was used to detect when different code was generated.

Table 6.3 shows a count, for each fault detector (Human, fuzzing tools, and one formal verifier), of fix locations reached, fix condition triggered, bitwise difference of generated code and failed tests (build tests are not expected to fail). One way of measuring whether there is a difference between faults detected (column 1) in human and automatically generated code is to compare number of fault triggers encountered (column 4).

Detector	Faults	Reached	Triggered	Bitwise-diff	Tests failed
Human	10	1,990	593	56	1
Csmith	10	2,482	1,043	318	0
EMI	10	2,424	948	151	1
Orange	5	293	35	8	0
yarpgen	2	608	257	0	0
Alive	8	1,059	327	172	0

Table 6.3: Fault detector, number of source locations fixed, number of fix locations reached, number of fix condition triggered, number of programs having a bitwise difference of generated code and number of failed tests. Data from Marcozzi et al.[1204]

Comparing the counts for the number of trigger occurrences experienced for each of the 10 fixes in each of the Human, Csmith and EMI detected source mistakes, finds that the differences between the counts is not statistically different across method of detection; see Github–reliability/OOPSLA-compiler.R.

The behavior of some software systems is sufficiently important to some organizations that they are willing to fund the development of a test suite intended to check behavior: cases include:

- the economic benefits of being able to select from multiple hardware vendors is dependent on being able to port existing software to the selected hardware; at a minimum, different compilers must be capable of processing the same source code to produce the same behavior. The US Government funded the development of validation suites for Cobol and Fortran,[1414] and later SQL, POSIX and Ada;[5] a compiler testing service was also established,[10]
- there were a sufficient number of C compiler vendors that several companies were able to build a business supplying test suites for this language, expanding to support C++

when this language started to become popular,[ix]

- the "Write once, run anywhere" design goal for Java required Sun Microsystems to fund the development of a conformance test suite, and to litigate when licensees shipped products that did not conform.[1319,1950]

While manual tests can be very effective, creating them is very time-consuming[1414] and expensive. Various kinds of automatic test generation are available, including exhaustive testing of short sequences of input[2009] and fuzzing.[2004]

Testing that a program behaves as intended requires knowledge of the intended behavior, for a given input. While some form of automatic input generation is possible, in only a few cases[61] is it possible to automatically predict the expected output from the input, independently of the software being tested. One form of program behavior is easily detected: abnormal termination, and some forms of fuzz testing use this case as their test criteria; see fig 6.23.

When multiple programs supporting the same functionality are available, it may be possible to use differential testing to compare the outputs produced from a given input (a difference being a strong indicator that one of the systems is behaving incorrectly).[345]

There are ISO Standards that specify methods for measuring conformance to particular standards[900,901] and requirements for test laboratories.[898] However, very few standards become sufficiently widely used for it to be commercially viable to offer conformance testing services.

Like source code, tests can contain mistakes.[1868]

To what extent do test suites change over time? A study by Marinescu, Hosek and Cadar[1207] measured the statement and branch coverage of six open source programs over time, using the test suite distributed with the program's source. Figure 6.47 shows that for some widely used programs the statement coverage of the test suite did not vary much over five years.

One study[1999] found no correlation between the growth of a project and its test code; see Github–time-series/argouml_complete.R.

The application build process may require the selection of a consistent set of configuration options. The Linux 2.6.33.3 kernel supports 6,918 configuration options, giving over $10^{23,563}$ option combinations. One study[1139] using a random sample of 1,000,000 different option combinations failed to find any that were valid according to the variability model; a random sampling of the more than $10^{1,377}$ possible option combinations supported by OpenSSL found that 3% were considered valid. Various techniques have been proposed for obtaining a sample of valid configuration options,[965] which might then be used for testing different builds of an application, or analyzing the source code.[1910]

Regular expressions are included within the syntax of some languages (e.g., awk and SNOBOL 4[743]), while in others they are supported by the standard library.[333] A given regular expression may match (or fail to match) different character sequences in different languages[443] (e.g., support different escape sequences, and different disambiguation policies; PCRE based libraries use a leftmost-greedy disambiguation, while POSIX based libraries use the leftmost-longest match[182]).

A study by Wang and Stolee[1921] investigated how well 1,225 Java projects tested the regular expressions appearing in calls to system libraries (such as `java.lang.String.matches`); there were 18,426 call sites. The regular expression methods were instrumented to obtain the regular expression and the input strings passed as arguments. When running the associated project test suite 3,096 (16.8%) of call sites were evaluated; method argument logging during test execution obtained 15,096 regular expressions and 899,804 test input strings (at some call sites, regular expressions were created at runtime).

A regular expression can be represented as a deterministic finite state automata (DFA), with nodes denoting states and each edge denoting a basic subcomponent of the regular expression. Coverage testing of a regular expression involves counting the number of nodes and edges visited by the test input.

Figure 6.48 shows a violin plot of the percentage of regular expression components having a given coverage. The nodes and edges of the DFA representation of each of the 15,096 regular expressions are the components measured, using the corresponding test input strings for each regex; coverage if measured for both matching and failing inputs.

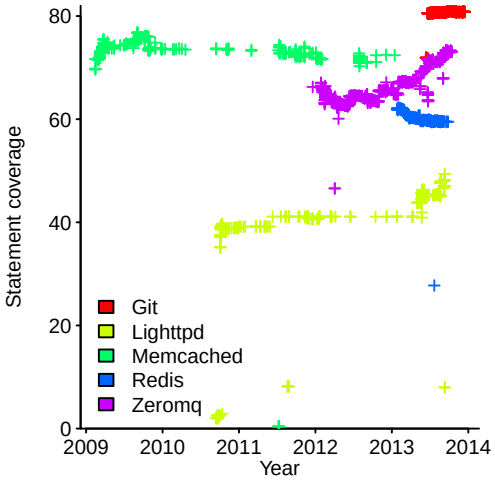


Figure 6.47: Statement coverage achieved by the respective program's test suite (data on the sixth program was not usable). Data from Marinescu et al.[1207] Github–Local

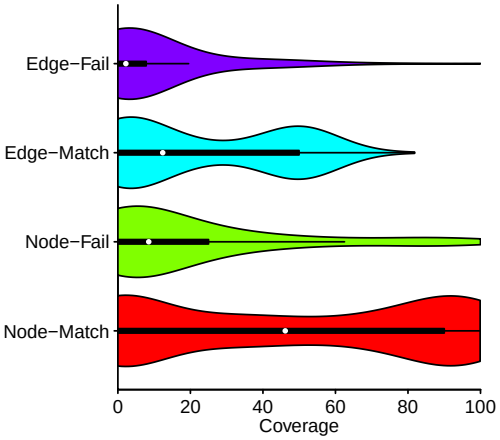


Figure 6.48: Violin plots of percentage of regular expression components having a given coverage, (measured using the nodes and edges of the DFA representation of the regular expression, broken down by the match failing/succeeding) for 15,096 regular expressions, when passed the corresponding project test input strings. Data kindly provided by Wang.[1921] Github–Local

[ix] A vendor of C/C++ validation suites (selling at around $10,000), once told your author they had over 150 licensees; a non-trivial investment for a compiler vendor.

6.6.2.1 Creating tests

Traditionally tests were written by people employed to test software; some companies have Q/A (quality assurance) departments. The tests developers write to check their code may become part of the systems formal test suite; there are development methodologies in which include testing is a major component of implementation, e.g., test driven development.

Automated test generation can reduce the time and costs associated with testing software. A metric often used by researchers for evaluating test generation tools is the number of fault experiences produced by the generated tests, i.e., one of the factors involved in gaining confidence that program behavior is likely to be acceptable.

Automated test generation techniques include (there is insufficient evidence to evaluate the extent to which an automatic technique is the most cost effective to use in a particular development):

- random modification of existing tests: so-called *fuzzing* makes random changes to existing test inputs, and little user input is required to test for one particular kind of fault experience, abnormal termination; see fig 6.23. Some tools use a fuzzing selection strategy intended to maximise the likelihood of generating a file that causes a crash fault, e.g., CERT's BFF uses a search strategy that gives greater weight to files that have previously produced faults[861] (i.e., it is a biased random process),
- source code directed random generation: this involves a fitness function, such as number of statements covered or branches executed.

  A study by Salahirad, Almulla and Gay[1630] investigated the ability of eight fitness functions, implemented in the EvoSuite tool, to generate tests that produced fault experiences for 516 known coding mistakes; a test generation budget of two and ten-minutes per mistake was allocated on the system used. The branch coverage fitness function was found to generate tests that produced the most fault experiences,
- input distribution directed random generation: the generation process uses information about the characteristics of the expected input (e.g., the probability of an item appearing, or appearing in sequence with other items) to generate tests having the same input characteristics.

  A study by Pavese, Soremekun, Havrikov, Grunske and Zeller[1455] used the measured characteristics of input tests to create a probabilistic grammar that generated tests having either the same distribution or were uncommon inputs (created by inverting the measured input item probabilities).

Automated test generation techniques are used to find vulnerabilities by those seeking to hijack computer systems for their own purposes. To counter this threat, tools and techniques have been created to make automatic test generation less cost effective.[960]

A program's input may be include measurements of a set of different items (e.g., the time of day, the temperature and humidity), and within the code there may be an interaction between these different items (these items are sometimes called *factors*). A coding mistake may only be a source of a fault experience when two different items each take a particular range of values, and not when just one of the items is in this range.

Combinatorial testing involves selecting patterns of input that are intended to detect situations where a fault experience is dependent on a combination of different item input values. The generation of the change patterns used in combinatorial testing can be very similar to those used in the design of experiments, and the same techniques can be used to help minimise the number of test cases; see section 13.2.5.

A study by Kuhn, Kacker and Lei[1053] investigated the percentage of fault experiences likely to require a given number of factors, in some combination. Figure 6.49 shows the cumulative growth in the percentage of known faults experienced, for tests involving combinations of a given number of factors (x-axis).

A study by Czerwonka[429] investigated the statement and branch coverage achieved, in four Microsoft Windows utilities, by combinatorial tests. The tests involved values for a single factor, and interaction between factors (from two to five factors). The results found that most of the variance in the measurements of branch and statement coverage could be explained by models fitted using the *log* of the number of combination of factors and the *log* of the number of tests; see Github–reliability/Coverage-Combin.R.

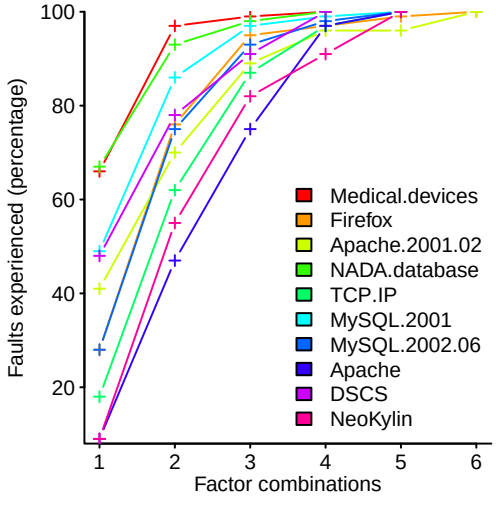


Figure 6.49: Percentage of known faults experienced for tests involving a given number of combinations of factors (x-axis), for ten programs. Data from Kuhn et al.[1053] Github–Local

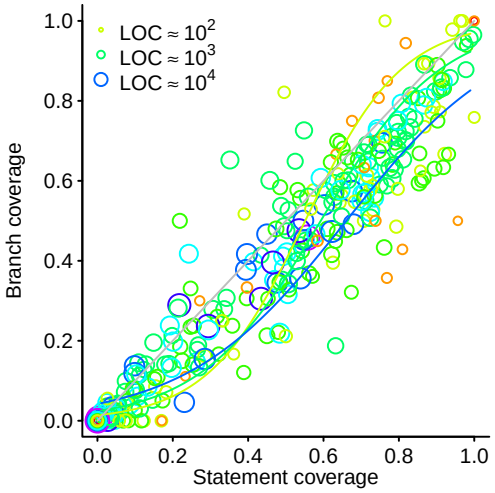


Figure 6.50: Statement coverage against branch coverage for 300 or so Java projects; colored lines are fitted regression models for three program sizes (see legend), equal value line in grey. Data from Gopinath et al.[712] Github–Local

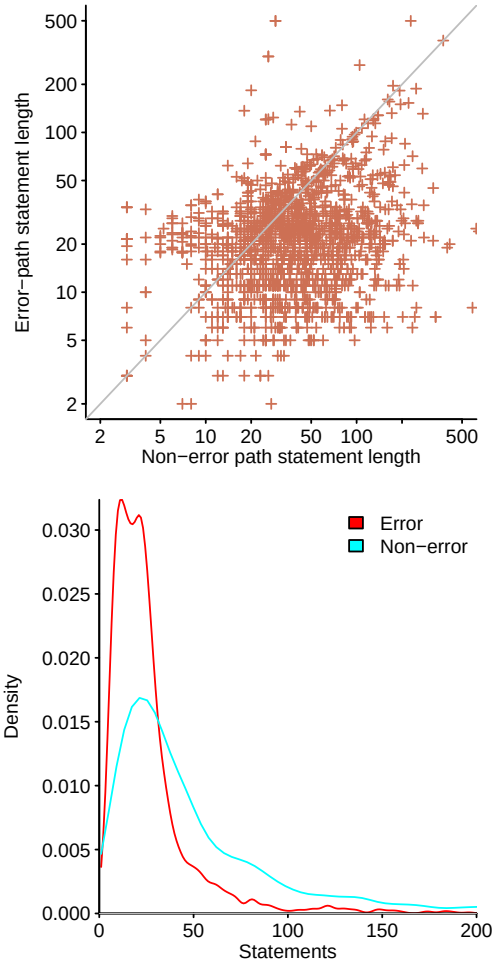


Figure 6.51: Number of statements executed along error and non-error paths within a function (top), and density plots of the number of statements along error and non-error paths. Data kindly provided by Kang.[974] Github–Local

6.6.2.2 Beta testing

The input profiles generated by the users of a software system may be very different from those envisaged by the developers who tested the software. Beta testing is a way of discovering problems with software (e.g., coding mistakes and incomplete requirements), when processing the input profiles of the intended users. The number of problems found during beta testing, compared to internal testing provides feedback on the relevance of the usage profile that drives the test process.[1201]

Beta testing is also a form of customer engagement.

6.6.2.3 Estimating test effectiveness

Does the project test process provide a reliable estimate, to the desired level of confidence, that the software is likely to be acceptable to the customer?

A necessary requirement for checking the behavior of code is executing it, every statement not executed is untested. Various so called *coverage* criteria are used, for instance percentage of program methods or statements executed by the test process (the coverage achieved by a test suite is likely to vary between platforms, different application configurations,[1544] and even the compiler used[651]).

The conditional expression in `if-statements` controls whether a block of statements is executed, or not. Branch coverage simply counts the number of branches executed, e.g., one branch for each of the true and false arms of an `if-statement`. More thorough coverage criteria measure the coverage of the decisions involved in the conditional expression, which can be an involved process; an expression may involve decisions conditions (e.g., `x && (y || z)`), with each subcondition derived from a different requirement. Modified Condition and Decision Coverage (MC/DC)[352] is one measure of coverage of the combination of possible decisions that may be involved in the evaluation of a conditional expression. For this example, the MC/DC requirements are met by `x`, `y` and `z` (assumed to take the values `T` or `F`) taking the values: `TFF`, `TTF`, `TFT`, and `FTF`, respective.

The probability of tests written to the MC/DC requirements detecting an incorrect condition is always at least 93.75%;[353] see Github–reliability/MCDC_FP.R.

One weakness of MC/DC is its dependence on the way conditions are expressed in the code. In the previous example, assigning a subexpression to a variable: `a=(y || z);`, simplifies the original expression to: `x && a`, and reduces the number of combinations of distinct values that need to be tested to achieve 100% MC/DC coverage.[x] One study[660] was able to restructure code to achieve 100% MC/DC coverage using 50% fewer tests than the non-restructured code (in some cases even fewer tests were needed). Achieving MC/DC coverage is often a requirement for software used within safety critical applications.

A study by Inozemtseva and Holmes[891] investigated test coverage of five very large Java programs. The results showed a consistent relationship between percentage statement coverage, *sc*, and percentage branch coverage, *bc* (i.e., $bc \propto sc^{1.2}$), and percentage modified condition coverage, *mc* (i.e., $mc \propto sc^{1.7}$); see Github–reliability/coverage_icse-2014.R.

The most common use of branches is as a component of the conditional expression in an `if-statement`, which decides whether to execute the statements in the enclosed compound statement. Most compound statements contain a few statements,[930] so a close connection between branch and statement coverage is to be expected.

A study by Gopinath, Jensen and Groce[712] investigated the characteristics of coverage metrics for the test suites of 1,023 Java projects. Figure 6.50 shows the fraction of statement coverage against branch coverage; each circle is data from one project. The various lines are fitted regression models, which contain a non-simple interaction between coverage and $\log(KLOC)$.

In figure 6.50, why does branch coverage tend to grow more slowly than statement coverage? Combining the findings from the following two studies suggest a reason:

- A study by Kang, Ray and Jana[974] investigated the number of statements encountered along the execution paths, within a function, executed after a call to a function that

[x] Some tools track variables appearing in conditionals that have previously been assigned expressions whose evaluation involved equality or relational operators.

could return an error, i.e., the error and non-error paths. Figure 6.51, lower plot, shows that non-error paths often contain more statements than the error paths; in the upper plot most of the points are below the line of equal statement-path length, i.e., there are a greater number of longer non-error paths.

- A study by Čaušević, Shukla, Punnekkat and Sundmark[308] found that developers wrote almost twice as many positive tests as negative tests,[xi] for the problem studied; see Github–reliability/3276-TDD.R. This behavior may be confirmation bias; see section 2.2.1.

If a test suite contains more positive than negative tests, and positive tests involve more statements than negative tests, then statement coverage would be expected to grow faster than branch coverage.

A basic-block is a sequence of code that has one entry point and one exit point (a function call would be an exit point, as would any form of goto statement). In figure 6.52, the fitted regression line shows a linear relationship between basic-block coverage and decision coverage, i.e., the expected relationship (grey line shows *Decision* = *Block*).

A study by McAllister and Vouk[1228] investigated the coverage of 20 implementations of the same specification. Two sets of random tests were generated using different selection criteria, along with 796 tests designed to provide full functional coverage of the specification. Figure 6.53 shows the fraction of basic-blocks covered as the number of tests increases, for the 20 implementations (sharing the same color), and three sets of tests (the different colors); the lines are fitted regression models of the form: $coverage_{BB} = a \times (1 - b \times \log(tests)^c)$, where: a, b and c are constants (c is between -0.35 and -1.7, for this specification).

Another technique for estimating the effectiveness of a test suite, in detecting coding mistakes, is to introduce known mistakes into the source and measure the percentage detected by the test suite, i.e., the mistake produces a change of behavior that is detected by the test process; the modified source is known as a *mutant*. For this technique to be effective, the characteristics of the mutations have to match the characteristics of the mistakes made by developers; existing mutant generating techniques don't appear to produce coding mistakes that mimic the characteristics of developer coding mistakes.[710,713] A test suite that kills a high percentage of mutants (what self-respecting developer would ever be happy just detecting mutants?) is considered to be more effective than one killing a lesser percentage.

Figure 6.54 shows statement coverage against percentage of mutants killed. The various lines are fitted regression models, which contain a non-simple interaction between coverage and $\log(KLOC)$.

A test suite's mutation score converges to a maximum value, as the number of mutants used increases. For programs containing fewer than 16K executable statements, E, the number of mutants needed has been found to grow no faster than $O(E^{0.25})$,[2008] a worst case confidence interval for the error in the mutation score can be calculated.[711]

6.6.3 Cost of testing

Testing costs include the cost of creating the tests, subsequent maintenance costs (e.g., updating them to reflect changes in program behavior), and the cost of running the tests.

For some kinds of test creation, automatic test generation tools may be more cost effective than human written tests;[336] see Github–reliability/1706-01636a.R.

The higher the cost of performing system testing, the fewer opportunities there are likely to be to check software at the system level. Figure 6.55 shows the relationship between the unit cost of a missile (being developed for the US military), and the number of development test flights made.

When does it become cost effective to stop testing software? Some of the factors involved in stopping conditions are discussed in section 13.2.4. Studies[331,1988] have analysed stopping rules for testing after a given amount of time, based on number of faults experienced.

A study by Do, Mirarab, Tahvildari and Rothermel[504] investigated test case prioritization, and attempted to model the costs involved; multiple versions of five programs (from 3 to 15 versions) and their respective regression suites were used as the benchmark. The performance of six test prioritization algorithms were compared, based on the number of

[xi] Sometimes known as error and non-error tests.

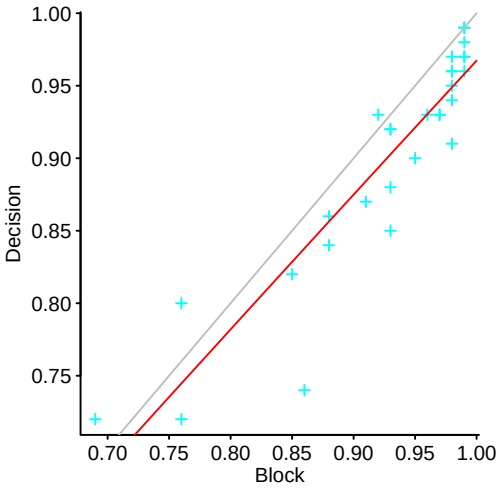


Figure 6.52: Basic-block coverage against branch coverage for a 35 KLOC program; lines are a regression fit (red) and *Decision* = *Block* (grey). Data from Gokhale et al.[693] Github–Local

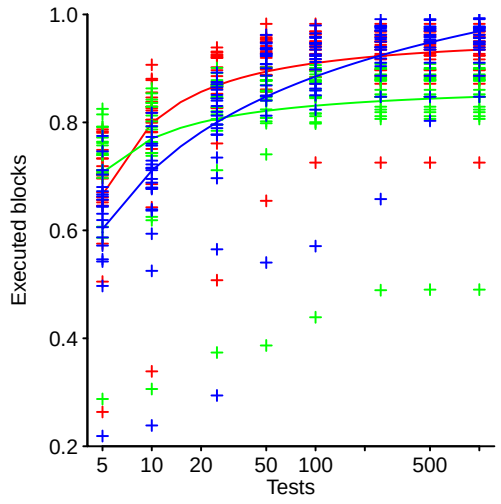


Figure 6.53: Fraction of basic-blocks executed by a given number of tests, for 20 implementations using three test suites. . Data from McAllister et al.[1228] Github–Local

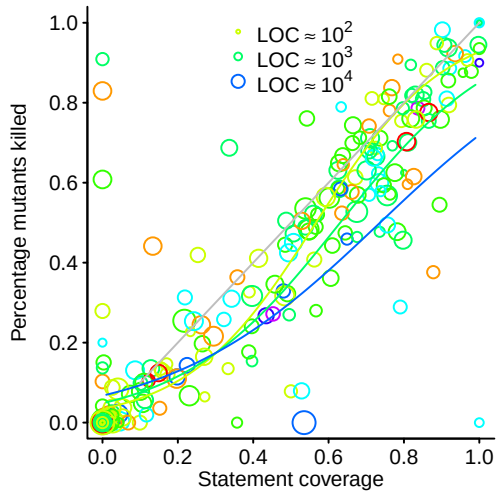


Figure 6.54: Statement coverage against mutants killed for 300 or so Java projects; colored lines are fitted regression models for three program sizes, equal value line in grey. Data from Gopinath et al.[712] Github–Local

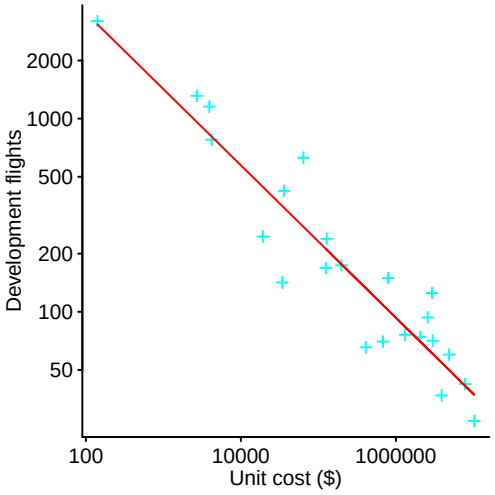


Figure 6.55: Unit cost of a missile, developed for the US military, against the number of development test flights carried out, with fitted power law. Data extracted from Augustine.[88] Github–Local

seeded coding mistakes detected when test resource usage was limited (reductions of 25, 50 and 75% were used). The one consistent finding was that the number of faults experienced decreased as the amount of test resource used decreased, there were interactions between program version and test prioritization algorithm; see Github–reliability/fse-08.R.

Chapter 7

Source code

7.1 Introduction

Source code is the primary deliverable product for software development. The purpose of studying source code is to find ways of reducing the resources needed to create and maintain it, and resources such as the developer cognitive effort needed to process code.

The limiting resource during software development is the experience and cognitive effort deliverable by the people involved; the number of people involved may be limited by the funding available, and their individual experience and cognitive firepower is limited to those people that could be recruited.

A study by Ikutani, Kubo, Nishida, Hata, Matsumoto, Ikeda and Nishimoto[887] investigated the neural basis for programming expertise. Subjects (10 top ranking, 10 middle ranking, 10 novices, on the AtCoder competitive programming contest website) categorized 72 Java code snippets into one of four categories (maths, search, sort, string); each snippet was presented three times, for a total of 216 categorization tasks per subject. The tasks were performed while each subject was in an fMRI scanner. A searchlight analysis[562] of the fMRI data was used to locate brain areas having higher levels of activity; figure 7.1 is a composite image of all active locations found over all 30 subjects.[i]

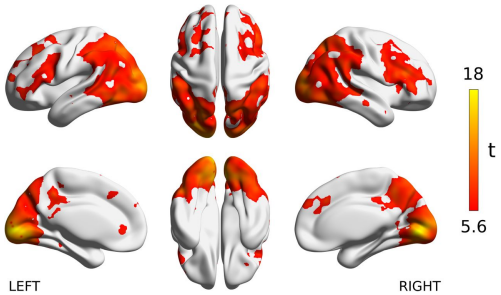


Figure 7.1: Composite image of brain areas active when 30 subjects categorized Java code snippets; colored scale based on t-value of the decoding accuracy of source code categories from the MRI signals. Image from Ikutani et al.[887]

Building a software system involves arranging source code in a way that causes the desired narrative to emerge, as the program is executed, when processing user inputs. As more complicated problems are solved and more functionality is added to existing systems, the quantity of source code contained in software systems has grown. Figure 7.2 shows the lines contained in the source of the implementations of the collected algorithms published by the "Transactions on Mathematical Software";[1841] the fitted regression line has an annual rate of growth of 11%.

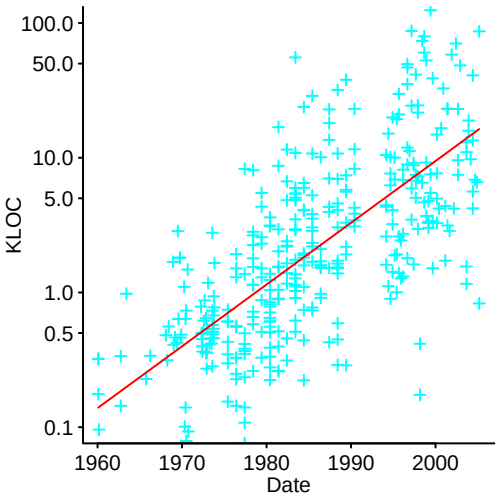


Figure 7.2: Lines of code in published implementations of the collected algorithms of the Transactions on Mathematical Software; line is a fitted regression of the form: $LOC \propto e^{0.0003Day}$. Data extracted by your author. Github–Local

Source code is a form of communication, written in a programming language, whose purpose is often to communicate a sequence of actions, or required results[ii] to a computer, and sometimes a person.

There are many technical similarities between programming languages and human languages, however, the ways in which they are used to communicate are very different (differences are visible in medical imaging of brain activity[617]). While there has been a lot of research investigating the activities involved in processing written representations of human language[441] (i.e., prose[iii]; see section 2.3), there have been few studies of the activities involved in the processing source code by people. For instance, eye-tracking is commonly used to study reading prose, but is only just starting to be used to study reading code; see fig 2.17. This chapter draws on findings from the study of prose (see section 2.3.1), highlighting possible patterns of behavior that might apply to source code.

The influential work of Noam Chomsky[357] led to widespread studying of language based on the products of language (e.g., words and sentences) abstracted away from the context

[i] The color scale is a measure (using t-contrasts) of the explanatory power of source code categorization of the recorded MRI signals.

[ii] In a declarative language, such as SQL, the intended result is specified (e.g., return information matching the specified conditions), while code written in an imperative language specifies actions to be performed (with the author being responsible for sequencing the actions to produce an intended result).

[iii] English prose has been the focus of most studies, with prose written in other languages not yet so extensively studied.

in which they are used. The cognitive linguistics[1552] approach is based around how people use language, with context taken into account, e.g., intentions and social normals, as in the work of Grice.[739] This chapter takes a cognitive linguistics approach to source code, including:

- human language use is a joint activity, based on common ground between speaker and listener[iv]. What a speaker says is the evidence used by a listener to create an interpretation of the speaker's intended meaning; a conversation involves cooperation and coordinating activities.

  Grice's maxims[739] provide a model of human communication, these are underpinned by speaker aims and listener assumptions. Perhaps the most important aspect of human language communication is the assumption of relevance[369,1747] (and even optimal relevance). A speaker says something because they believe it is worth the effort, with both speaker and listener assuming that what is said is relevant to the listener, and worth the listener investing effort to understand,
- communicating with a computer, using source code, is a take-it or leave-it transaction, what the speaker *said* is treated as definitive by the listener, intent is not part of the process. There is no cooperative activity, and the only coordination involves the speaker modifying what has been *said* until the desired listener response is achieved.

  Human readers of source code may attempt to extract an intent behind what has been written.

  Creating a program requires explaining, as code, everything that is needed. The implementation language may support implicit behavior (e.g., casting operands to a common type), or be based on a specific model of the world, e.g., domain specific languages. Commonly occurring narrative subplots may be available as library functions.

Experienced developers are aware of the one-sided nature of communicating with a computer, and have learned a repertoire of techniques for adjusting their approach to communication; novices have to learn how to communicate with a listener who is not aware of their intent. One aspect of learning to program is learning to communicate with an object that will not make any attempt to adapt to the speaker; patterns have been found in the ways novices misunderstanding the behavior of code.[1458]

Source code is used to build programs and libraries (or packages). There are a variety of different kinds of text that might be referred to as *source code*, e.g., text that is compiled to produce an executable program, text that is used to direct the process of building programs and system distributions (such as Makefiles and shell scripts), text that resembles prose in configuration files, and README files[886] containing installation and usage examples.

A study by Pfeiffer[1479] investigated the kinds of files contained in 23,715 Github repositories. Files were classified into four high-level categories (i.e., code, data, documentation, and other), and 19 lower level categories, e.g., script, binary code, build code, video, font, legalese. Figure 7.3 shows the fraction of files in each high-level category within repositories containing a given total number of files, averaged over repositories having the given total numbers of files (the data was smoothed using a rolling mean, with a window width of three).

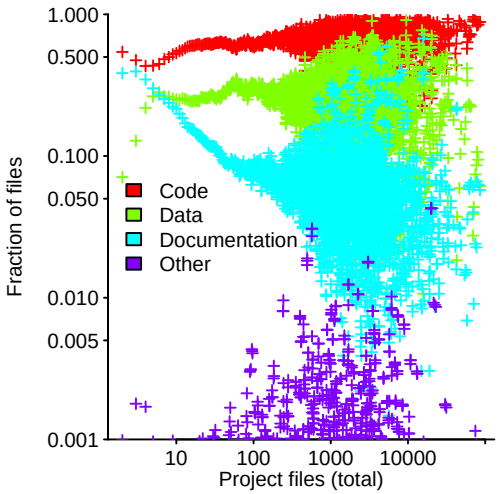


Figure 7.3: Fraction of files in high-level categories for 23,715 repositories containing a given number of files (averaged over all repositories containing a given number of files). Data from Pfeiffer.[1479] Github–Local

Source code is the outcome of choices made in response to implementation requirements, the culturally derived beliefs and experiences of the developers writing the code (coupled with their desire for short-term gratification[630]), available resources, and the functionality provided by the development environment.

In some application domains, effective ways of organizing implementation components have become widely known. For instance, in the compiler writing domain, the production-quality compiler-compiler project[1119] created a way of organizing an optimizing compiler, as a sequence of passes over a tree, that has been widely used ever since.[v]

High-level implementation plans have to be broken down into smaller components, and so on down, until the developer recognises how a solution that can be directly composed using code.

Developers have a collection of beliefs, and mental models, about the semantics of the programming languages they use, as well as a collection of techniques they have become practiced at, and accustomed to using.

[iv] "vaj laH: pejatlh lion ghaH yajbe' maH." Wittgenstein.

[v] Recent research has been investigating subcomponent sequencing selected machine learning, on a per-compilation basis.[82]

The low-level patterns of usage found in source code arise from the many coding choices made in response to immediate algorithmic and implementation requirements. For instance, implementing the $3n+1$ problem requires testing whether n is odd or even. In one study,[1876] the expressions used for this test included: `n % 2` and `n & 1` (89% and 8% of uses respectively), along with less common sequences such as: `(n >> 1) << 1 ==n` and `(n/2)*2 ==n`; around 50% of the conditions returned a non-zero value (i.e., true), when n is even. Over 95% of the expressions appeared as the condition of an `if-statement`, with most of the others appearing as the first operand of a ternary operator, e.g., `n=(n%2)? 3*n+1 :n/2`.

There are a huge variety of different ways of implementing the same functionality (see fig 5.23), and neutral variants of existing programs can be created[779] (i.e., small changes that do not affect the external behavior); the style of some developers source code is sufficiently different from other developers that it can be used to distinguish them from other developers.[292]

If, after executing the two statements: `x=1;y=x+2;`, the statement: `x=3;`, is executed, is the value of `y` now 5? In many programming languages the value of `y` remains unchanged by the second assignment to `x`, but in reactive programming languages[118] (found in spreadsheets) statements can express a relationship, not a one time assignment of a value (novice developers have been found to give a wide variety of interpretations to the assignment operator[223]).

While spreadsheets support the specification of formula and relationships between named memory locations, they have not traditionally been treated as part of software engineering. One reason has been the lack of large samples of usage to study; legal proceedings against large companies are helping to change this situation.[815]

Acquiring an understanding of the behavior of a program, by reading its source code, is not an end in itself; one reason for making an investment to acquire this understanding, is to be able to predict a program's behavior sufficiently well to be able to change it. By reading source code, developers acquire beliefs about it, which are a means to an end; *understanding a program* is a continuum, not a yes/no state.

The complexity and inter-connectedness found in software systems can also be found in non-software systems. A study by Braha and Bar-Yam[239] investigated the network connections in a 16-story hospital, pharmaceutical facilities design, a General Motors vehicle design facility and Linux. Table 7.1 lists the values of various properties of the networks and from a component network perspective software looks middle of the road.

	Nodes	Edges	Average path length	Clustering coeff	Degree	Density
Hospital	889.00	8178.00	3.12	0.11	18.40	0.02
Pharmaceutical	582.00	3689.00	2.63	0.12	12.68	0.02
Software	466.00	1245.00	3.70	0.14	5.34	0.01
Vehicles	120.00	417.00	2.88	0.13	6.95	0.06

Table 7.1: Values of various attributes of the communication network graphs for various organizations. Data from Braha et al.[239] Github–Local

So-called *end-user* programming involves non-developers writing code to perform simple tasks. Trigger-Action-Programs, written in languages such as IFTTT (If This Then That), can be created by users to control IoT devices (e.g., smart speakers);[1273] spreadsheets[815] are used by a wide range of non-developers; and, Scratch[828] is used for teaching children.

Are the factors driving non-developer written applications sufficiently different from the factors driving developer written applications, that the characteristics of the code written is noticeably different?

This question is outside the scope of this book, which focuses on professional developers.

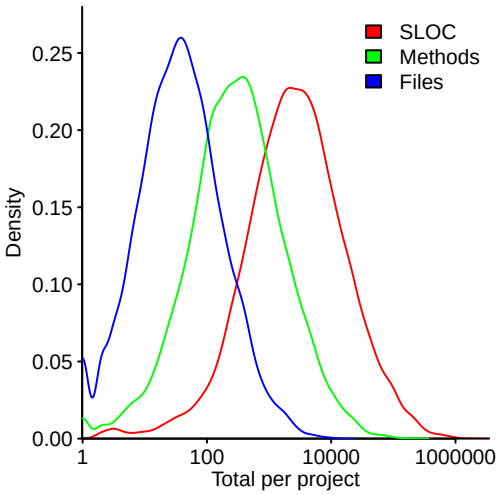


Figure 7.4: Number of source files, methods, and lines of code within methods, contained in each of 13,103 Java projects; lines are kernel density plots. Data kindly provided by Landman.[1077] Github–Local

7.1.1 Quantity of source

The size of software systems is sometimes used to obtain an estimate of the cost of its production, the time taken to implement it, and number of mistakes it contains; see fig 5.1 and fig 5.3. Given the large number of variables in the production process (e.g., the large variation in the quantity of code written by different developers to implement the same

functionality, see fig 5.23), size only has any likelihood of good enough accuracy for comparisons within the same project team working on the system.

What is the size distribution of software systems, and to what extent do the characteristics of subcomponents vary with size?

The size of a software system is often measured by lines of source code. While building a program from source invariably involves a variety of configuration files (e.g., makefiles), such files are not always categorized as source files; counts may specify the kinds of file counted, e.g., those having a given file suffix. Figure 7.4 shows the number of source files, methods and SLOC (within each method) for 13,103 Java projects (y-axis shows density, rather than a count).

to	SLOC	Methods	Classes	Files
SLOC		$13^{+20}_{-8} \times Methods^{0.97\pm0.05}$	$50^{+100}_{-30} \times Classes^{1.02\pm0.08}$	$64^{+300}_{-50} \times Files^{1.04\pm0.13}$
Methods	$0.11^{+0.2}_{-0.07} \times SLOC^{0.98\pm0.04}$		$4.5^{+10}_{-3} \times Classes^{1.03\pm0.08}$	$7^{+20}_{-5} \times Files^{1.05\pm0.11}$
Classes	$0.054^{+0.07}_{-0.04} \times SLOC^{0.86\pm0.03}$	$0.42^{+0.6}_{-0.3} \times Methods^{0.86\pm0.04}$		
Files	$0.051^{+0.1}_{-0.04} \times SLOC^{0.84\pm0.08}$	$0.23^{+0.4}_{-0.2} \times Methods^{0.89\pm0.07}$		

Table 7.2: Equations that map between total number of Java source constructs in a software system (fitted using quantile regression, the bounds are derived from 95% and 5% quantile regression models). Data from Lopes et al,[1158] and the Files data is from Landman et al.[1077] Github–Local

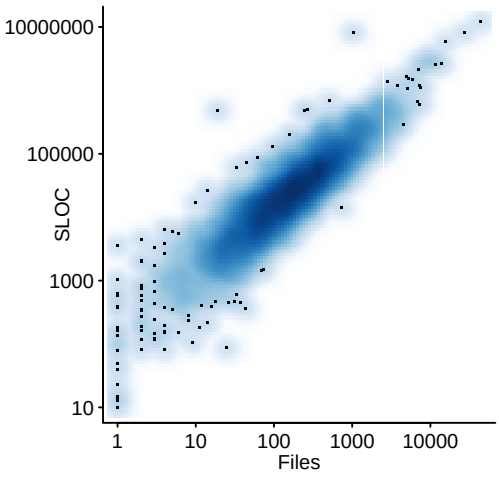


Figure 7.5: Number of files and lines of code in 3,782 projects hosted on Sourceforge. Data from Herraiz.[817] Github–Local

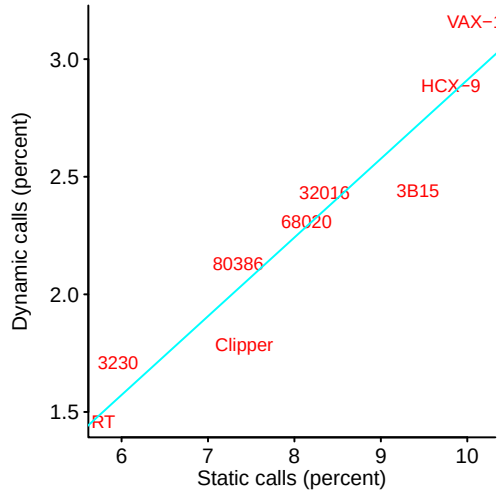


Figure 7.6: Percentage of call instructions contained in code generated from the same C source, against call execution percentage for various processors; grey line is fitted regression model. Data from Davidson et al.[440] Github–Local

There is sufficient consistency in source code layout for lines to be treated as a good enough unit of measure. In many languages a *method* (also known as a *function*, *procedure* or *subroutine*) is the smallest self-contained unit of source code, with larger units of measurement including classes and files. These characteristics are interchangeable in the sense that, when the value of one of them is known, an order of magnitude approximation of the other values can be calculated.

Table 7.2 shows the equations obtained by fitting quantile regression models to measurements of Java systems containing 10 or more methods (the study by Lopes and Ossher[1158] counted SLOC in the classes of 27,063 Java systems, and Landman, Serebrenik, Bouwers and Vinju[1077] counted SLOC in the methods of 12,628 Java systems). The listed equation uses the median, with uncertainty bounds derived from fitting the 95% and 5% quantiles.

More accurate models can be used when information on more characteristics is available, e.g., $SLOC = e^{1.98} Methods^{1.12} Files^{-0.13}$, and $SLOC = e^{1.78} MethodsInClasses^{0.8} Classes^{-0.2}$.

To what extent do the size characteristics of source code written in other languages follow the patterns seen in Java (i.e., power law with an exponent close to one)? The pattern seen in figure 7.16 shows that different size characteristics occur in at least one other language.

A study by Herraiz[817] measured the number of files and SLOC in a snapshot of 3,781 projects hosted on SourceForge (as of June 2006, having at least three developers and one year of history). Figure 7.5 shows number of files against SLOC, as a density plot; the fitted quantile regression model, with 95% bounds is: $SLOC = 167^{+563}_{-134} \times Files^{0.98\pm0.09}$.

Sometimes the only information available about a program is contained in its executable form. Studies[1796] have built models that estimate the length of the original source from the compiled machine code (the language in which the source is written can have a large impact on machine code characteristics[289]).

The relationship between static and dynamic counts of machine code, compiled from the same source, can be noticeably affected by processor characteristics and the compiler used. Figure 7.6 shows static and dynamic percentage of call instruction opcodes, in the compiled form of various C programs targeting various processors.

The quantity of source has been found to be an approximate predictor of compile time. One study[1282] of Ada compilers found that compile-time was proportional to the square root of the number of tokens; see Github–sourcecode/ADA177652.R. A study by Auler and Borin[89] investigated the time taken by a JIT compiler to generate code for functions in the SPEC CPU2006 benchmark. Figure 7.7 shows the time taken to generate machine code for 71,200 functions, containing a given number of LLVM instructions (an intermediate code). The two trends in the data are: compile-time not depending on function size and compile-time increasing linearly (once functions contain more than 100 instructions).

7.1.2 Experiments

Human experiments are the primary technique for unravelling developer performance interactions with source code. Usage patterns in source code are the emergent outcome of developer behavior, application requirements and the characteristics of the development environment. Focusing on common usage patterns can be an efficient use of research resources.

Obtaining reliable experimental results requires controlling all the variables likely to affect the outcome. Asking subjects to solve variations of a specific simple question is one method of excluding extraneous factors. This is a bottom up approach to understanding behavior.

Subjects will vary in ability and questions will vary in difficulty; item response theory can be used to model this kind of performance data.

A study by Chapman, Wang and Stolee[332] investigated the accuracy with which 180 workers on Mechanical Turk (who had to correctly answer 4-5 questions about regular expressions before being accepted) matched regular expressions against particular character sequences (and also constructing a character sequence to match a given regular expression). A set of 41 equivalent regex pairs (equivalent in that the same character sequences were matched by different regexs) was used to construct 60 problems, with 10 randomly selected for each worker to answer.

Figure 7.8 shows the probability that a worker with a given ability (x-axis) will correctly answer a particular problem (numbered colored lines).[vi] Unpicking the various ability/difficulty response patterns requires further work.

Some coding constructs can be incrementally made more difficult to answer, or support alternative representations.

A study by Ajami, Woodbridge and Feitelson[24] investigated the performance of 222 professional developers when answering questions about the behavior of code snippets. The questions contained 28 `if-statement` snippets (out of 41), whose conditions tested against a disjoint range of values, expressed either as a single expression, or as a sequence of nested `if-statements`. For instance:

```
// linear form:
       if (x>0 && x<10 || x>20 && x<30 || x>40 && x<50) {
          print("1");
       } else {
          print("2");
       }
// equivalent nested form:
       if (x<10) {
          if (x>0) {
             print ("1");
          }
       }
       else if (x<30)
          ...
```

The test performed by each snippet involved between two and four discrete ranges (in one condition, or via nested `if-statements`), and some tests involved negated subexpressions. Modeling subject response time finds that this increases as the number of ranges checked increases (by around 15%, or 3 seconds in an additive model), and decreases for nested `if-statements` (by around 10%, or 2 seconds in an additive model). Modeling answer correctness finds that this decreases as the number of ranges checked increases; see Github–sourcecode/Complexity18EmpSE.R.

The result from this experiment (if replicated) provides one set of inputs into the analysis of the factors involved in the use of `if-statements`.

Linear reasoning is discussed in section 2.6.2.

A condition testing for inclusion within a continuous single range can be written in many ways, including the following:

```
if (x > u && x < e) ...
if (u < x && x < e) ...
if (x < e && u > x) ...
```

[vi] For unknown reasons, the response to problem 51 is the opposite of that expected.

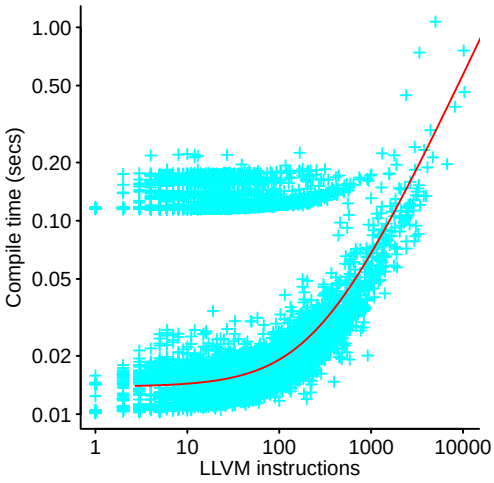


Figure 7.7: Time to compile, using -O3 optimization, each of 71,200 function (in the SPEC benchmark) containing a given number of LLVM instructions; line shows fitted regression model for one trend in the data. Data kindly provided by Auler.[89] Github–Local

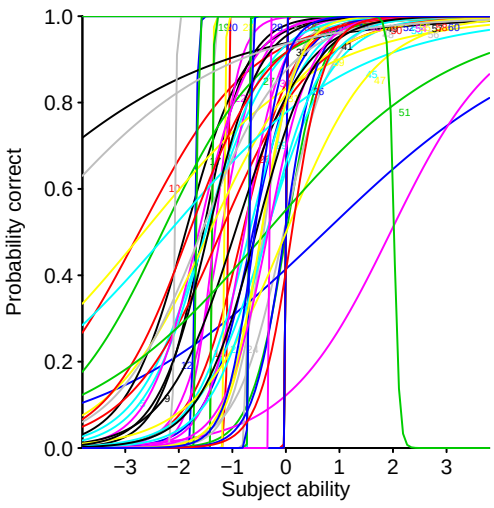


Figure 7.8: Probability that a worker having a given ability (x-axis) will correctly answer a given question (numbered colored lines); fitted using item response theory. Data from Chapman et al.[332] Github–Local

A study by Jones[929] investigated developer performance when answering questions about the relative value of two variables, given information about their values relative to a third variable. A total of 844 answers were given by 40 professional developers, with 40 incorrect answers (no timing information). With so few incorrect answers it is not possible to distinguish performance differences due to the form of the condition.

7.1.3 Exponential or Power law

Many source code measurements can be well fitted by an exponential and/or power law equation. Ideally, the choice of equation is driven by the theory describing the processes that generated the measurements, but such a theory may not be available. Sometimes, the equation chosen may be based on the range of values considered to be important.

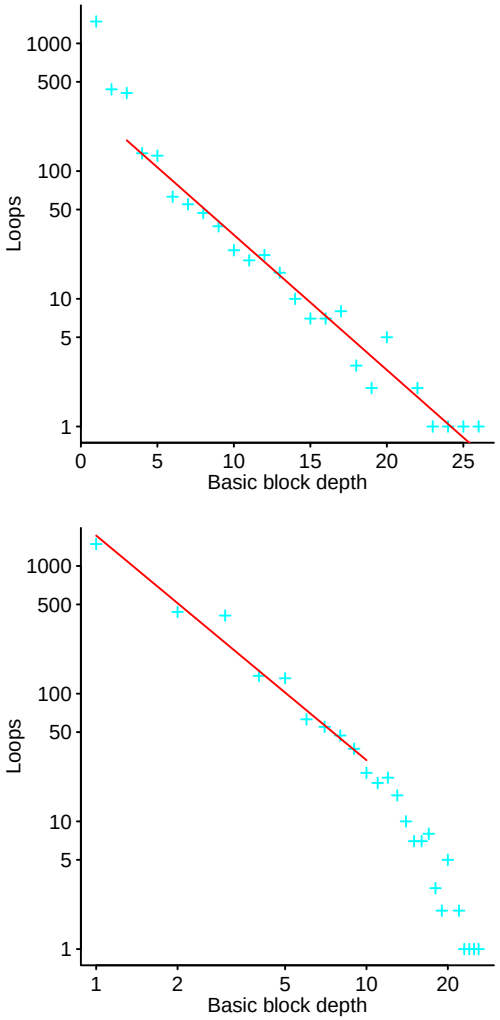


Figure 7.9: Number of for-loops, in C source, whose enclosed compound-statement contained basic blocks nested to a given depth; with fitted exponential (upper) and power law (lower). Data kindly provided by Pani.[1444] Github–Local

Figure 7.9 shows the same data fitted by an exponential (upper) and power law (lower); note different x-axis scales are used. The choice of equation to fit might be driven by the range of nesting depths considered to be important (or perhaps the range considered to be unimportant), and the desire to use a simple, brand-name, equation, i.e., the complete range of data may be fitted by a more complicated, and less well-known, equation.

In figure 7.39 an exponential was fitted, based on the assumption that the more deeply nested constructs were automatically generated, and that the probability of encountering a `selection-statement` in human written code did not vary with nesting depth.

Section 11.5.1 discusses the analysis needed to check whether it is statistically reasonable to claim that data is fitted by a power law.

7.1.4 Folklore metrics

Two source code metrics, proposed in the 1970s, have become established folklore within many software development communities: Halstead's metric and McCabe's cyclomatic complexity metric. Use of these metrics persists, despite the studies[902,1077,1191] finding that they have predictive performance that is comparable to that of lines of code, when used to model various program characteristics. Why do developers and researchers continue to use them? Perhaps counting lines of code is thought to lack the mystique that the market craves, or is the demand for more accurate metrics insufficient to motivate a search for something better?

Ptolomy's theory (i.e., the Sun and planets revolved around the Earth) was not discredited because it gave inaccurate answers, but because Copernicus's theory eventually[vii] made predictions that had a better fit to experimental measurements (once observing equipment became more accurate).

The studies involving what became known as Halstead's complexity metric were documented by Halstead in a series of technical reports[278,635,714,770,771,1427] published in the 1970s. The later reports compared theory with practice, using tiny datasets; Halstead's early work[771] is based on experimental data obtained for other purposes,[2031] and later work[635] used a data set containing nine measurements of four attributes (see Github–faults/halstead-akiyama.R), others contained 11 measurements,[714] or used measurements from nine modules.[1427]

Halstead gave formula for what he called *program length* (with source code tokens as the unit of measurement), *difficulty*, *volume* and *effort*. A dimensional analysis shows that the first three each have units of *length*, and *effort* has units of *area* (i.e., $length^2$), i.e., the names suggest that different quantities are being calculated, but they all calculate the same quantity in different ways.

Researchers at the time found that Halstead's metric did not stand up to scrutiny: Magidin and Viso[1191] used slightly larger datasets (50 randomly selected algorithms), and found many mistakes in the methodology used in Halstead's papers; a report by Shen, Conte and Dunsmore,[1688] Halstead's colleagues at Purdue, written four years after Halstead's flurry of papers, contains a lot of background material, and points out the lack of any theoretical foundation for some of the equations, that the analysis of the data was weak, and that a more thorough analysis suggests theory and data don't agree.

The Shen et al report explains that Halstead originally proposed the metrics as a way of measuring the complexity of algorithms not programs, explains the background to

[vii] Thanks to Isaac Newton.

Halstead's uses of the term *impurities*, and the discussion for the need for *purification* in his early work. Halstead points out[278] that the value of metrics for *algorithms written by students* are very different from those for the equivalent programs published in journals, and goes on to list eight classes of impurity that need to be purified (i.e., removing or rewriting clumsy, inefficient or duplicate code) in order to obtain results that agree with the theory, i.e., cleaning data in order to obtain results that more closely agree with his theory.

A study by Israeli and Feitelson[902] investigated the evolution of the Linux kernel. Figure 7.10 shows lines of code against Halstead's volume and McCabe's cyclomatic complexity, for the 62,365 C functions containing at least 10 lines, in Linux version 2.6.9. Explanations for the value of the exponents of the fitted power laws include: for Halstead's volume: the number of tokens per line increases with function size, and for McCabe's complexity: that the number of control structures decreases as function size increases.

The paper in which McCabe[1229] proposed what he called *complexity measures* contains a theoretical analysis of various graph-theoretic complexity measures, and some wishful thinking that these might apply to software; no empirical evidence is given. Studies[1077] have found a strong correlation between lines of code and McCabe's complexity metric. This metric also suffers from the problem of being very easy to manipulate, i.e., is susceptible to software accounting fraud; see section 6.4.2.

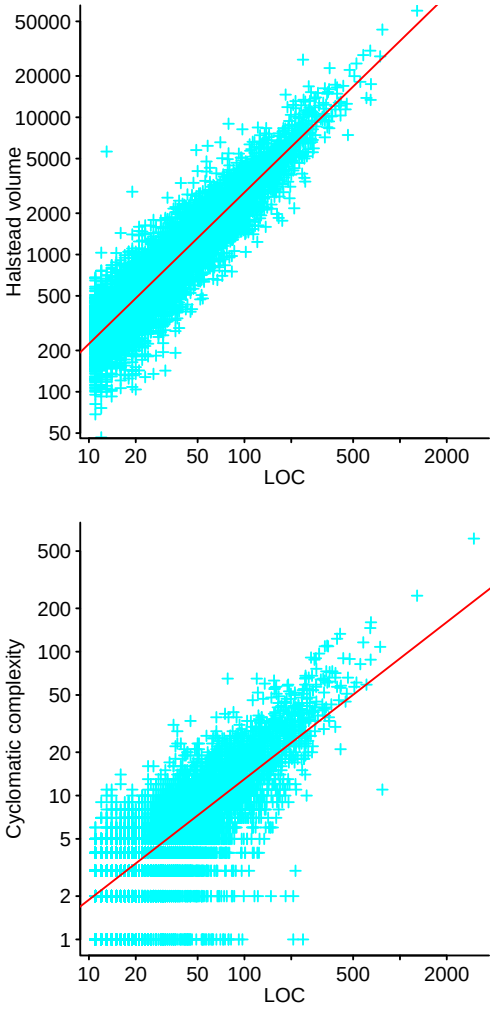


Figure 7.10: Lines of code, Halstead's volume and McCabe's cyclomatic complexity of the 62,365 C functions containing at least 10 lines, in Linux version 2.6.9; fitted regression lines have the form: $Halstead\_volume \propto KLOC^{1.1}$ and $McCabe\_complexity \propto KLOC^{0.8}$. Data from Israeli et al.[902] Github–Local

7.2 Desirable characteristics

What characteristics are desirable in source code?

This section assumes that desirable characteristics are those that minimise one or more developer resources (e.g., cost, time), and developer mistakes (figure 6.9 suggests that most of the code containing mistakes is modified/deleted before a fault is reported). Desirable characteristics include:

- developers' primary interaction with source code is reading it to obtain the information needed to get a job done. The resources consumed by this interaction can be reduced by:
  - reducing the amount of code that needs to be read, or the need to remember previously read code,
  - increasing the memorability of the behavior of code reduces the cost of rereading it to obtain a good enough understanding,
  - reducing the cognitive effort needed to extract a good enough understanding of the behavior of what has been read,
  - being consistent to support the use of existing information foraging[1491] skills, and reducing the need to remember exceptions,
- use of constructs whose behavior is the same across implementations (which has to be weighed against the benefits provided by use of implementation specific constructs):
  - reduces familiarisation costs for developers new to a project,
  - recruitment is not restricted to developers with experience of a specific implementation; see section 3.3.2,
- interacts with people's propensity to make mistakes in a fail-safe way:
  - robust in the sense that any mistake made is less likely to result in a fault experience,
  - use of constructs that contain redundant information, which provides a mechanism for consistency checking,
  - use of constructs that are fragile and likely to noticeably fail unless used correctly,
- executes within acceptable resource limits. While performance bottlenecks may arise from interaction between the characteristics of the input and algorithm choices, there are domains where individual coding constructs can have a noticeable performance impact, e.g., SQL queries,[248] or might be believed to have such an impact; see fig 7.30 and fig 11.22.

Use of the terms *maintainability*, *readability* and *testability* are often moulded around the research idea, or functionality, being promoted by an individual researcher, i.e., they are essentially a form of marketing.

The production of books, reports, memos and company standards containing suggestions and/or recommendations for organizing source code, and the language constructs to avoid (or use), so-called *coding guidelines*, is something of a cottage industry, e.g., for C.[428,629,640,784,858,909,989,1032,1232,1291,1496,1497,1545,1549,1557,1665,1748,1753,1789]

Stylistically, guideline documents are often more akin to literary criticism than engineering principles, i.e., they express personal opinions that are not derived from evidence (other than perhaps episodes in a person's life). Recommendations against the use of particular language constructs are sometimes based on the construct repeatedly appearing in fault reports; however, the possibility that the use of alternative constructs will produce more/fewer reported faults is rarely included in the analysis, i.e., the current usage may be the least bad of the available options. The C preprocessor is an example of frequently criticised functionality,[1255] that is widely used because of the useful functionality it uniquely provides.

While several ways of implementing the required functionality may be possible, at the time of writing there is little if any evidence available showing that any construct is more/-less likely to have some desirable characteristic, compared to another, e.g., less costly to modify or to understand by future readers of the code, or less likely to be the cause of mistakes.

7.2.1 The need to know

How might source code be organized to minimise the expenditure of cognitive effort per amount of code produced?[viii]

One technique for reducing the expenditure of cognitive effort is to reduce the amount of code that a developer needs to process to get a job done, e.g., reading code to understanding it and writing new code.

How might a developer know whether it is necessary to understand code without first having some understanding of it?

Many languages support functionality for breaking source up into self-contained units/-modules,[ix] each having a defined interface; these self-contained units might be functions/methods, classes, files, etc. Language vary in the functionality they provide to allow developers to control the visibility of identifiers defined within a module, e.g., `private` in Java.

The concept of *information hiding* is sometimes used in connection with creating interfaces and associated implementations. This term misrepresents the primary item of interest, which is what developers need to know, not how much information is hidden.

Modularization is a technique used in other domains that build systems from subcomponents, including:

- hardware systems use modularization to make it easier/cheaper to replace broken or worn out components, and to simplify the manufacturing process (as well as manufacturing costs). These issues are not applicable to software, which does not wear out, and has essentially zero manufacturing costs, however, the interface to the world in which the software operates may change in a way that creates a need to modify the software,

- biological systems where connections between components have a cost (e.g., they cause delays in a signalling pathway), and modularity is an organizational method that reduces the number of connections needed;[374] modularity as a characteristic that makes it easier to adapt more quickly when the environment changes may be an additional benefit, rather than the primary driver towards modularity. Simulations[1263] have found that, for non-trivial systems, a hierarchical organization reduces the number of connections needed (for a viable implementation).

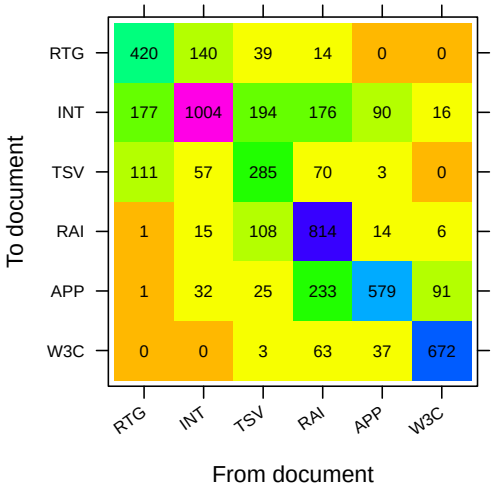


Figure 7.11: Number of citations from Standard documents within protocol level, to documents in the same and other levels (RTG routing, INT internet, TSV transport, RAI realtime applications and infrastructure, APP Applications, W3C recommendations). Data from Simcoe.[1707] Github–Local

[viii] There is too much uncertainty around measuring quantity of functionality provided to make this a viable end-point.

[ix] The term *module* is given a specific meaning in some languages.

Minimizing the need to know is one component of the larger aim of minimising the expenditure of cognitive effort per amount of code produced, which is one component of the larger aim of maximizing overall ROI, i.e., an increase in the need to know is a cost that may be a worthwhile trade-off for a greater benefit elsewhere.

A study by Simcoe[1707] investigated the modularity of communication protocol standards involved in the implementation of the Internet. Figure 7.11 shows the number of citations from IEFT and W3C Standard documents, grouped by protocol layer, that reference Standard documents in the same and other layers. Treating citations as a proxy for dependencies: 89% are along the main diagonal (a uniform distribution would produce 17%); dependencies discovered during implementation may substantially change this picture.

Dependencies between units of code can be used to uncover possible clusters of related functionality. One dependency is function/method calls between larger units of code, with units of code making many of the same calls likely to have something in common.

A study by Almossawi[43] investigated the architectural complexity of early versions of Firefox. The source code of the `gfx` module in Firefox version 20 is contained in 2,664 files, and makes 14,195 calls from/to methods in these files. Figure 7.12 shows one clustering of files based on the number of from/to method calls.

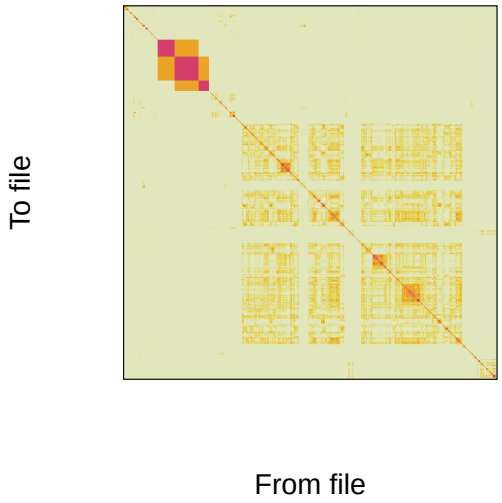


Figure 7.12: A clustering of the 2,664 files containing from/to method calls in the `gfx` module of Firefox version 20. Data kindly provided by Almossawi.[43] Github–Local

7.2.2 Narrative structures

People are inveterate storytellers, and narrative is another way of interpreting source code. People tell stories about their exploits, and a culture's folktales are passed on to each new generation. The narrative structures present in the folktales told within cultures have many features in common.[1532]

The Aarne-Thompson-Uther Index (ATU) is a classification of 2,399 distinct folktale templates (based on themes, plots and characters). A study by Tehrani[1819] investigated the phylogeny of the European folktale "Little Red Riding Hood" (ATU 333), a very similar folktale from Japan, China and Korea known as "The Tiger Grandmother" which some classify as ATU 123 (rather than ATU 333), and other similar folktales not in the ATU index. Figure 7.13 shows a phylogenetic tree of 58 folktales, based on 72 story characteristics, with 18 classified as ATU 333 (red), 20 as ATU 123 (blue), and 20 not classified (green).

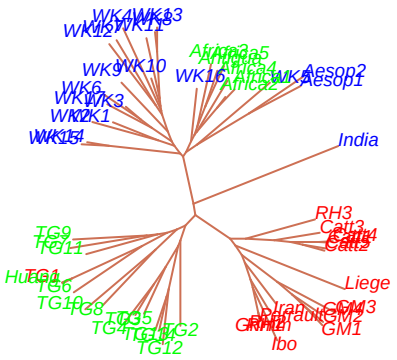


Figure 7.13: Phylogenetic tree of 58 folktales, based on 72 story characteristics; 18 classified as ATU 333 (red), 20 as ATU 123 (blue), and 20 unclassified (green). Data from Tehrani.[1819] Github–Local

The ability of some narrative structures to survive through many retellings, and to spread (or be locally created), suggests they have characteristics that would be desirable in source code, e.g., memorability and immunity to noise (such as introduction of unrelated subplots).

A program's narrative structure (i.e., its functionality) emerges from the execution of selective sequences of code, which may be scattered throughout the source files used to build a program. The source code of programs implemented using an Interactive Fiction approach is essentially the program narrative.[1211] The term *programming plans* is used in some studies.[1968]

A study by Wong, Gokhale and Horgan[1976] investigated the execution of basic blocks, within the 30 source files making up the SHARPE program, when fed inputs that exercised six of the supported features (in turn). Figure 7.14 shows the fraction of basic blocks in each file executed when processing input exercising a particular feature.

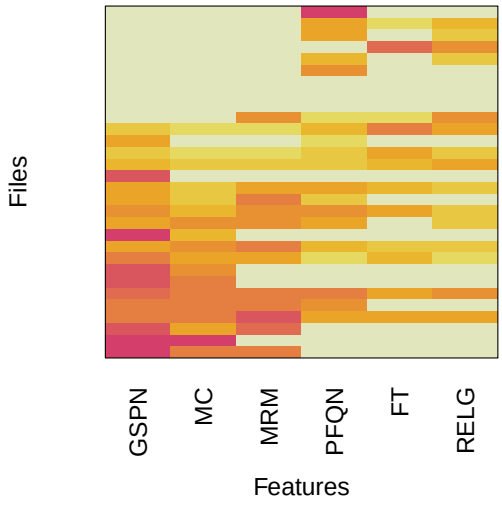


Figure 7.14: Heat map of the fraction of each of 30 files' basic blocks executed when performing a given feature of the SHARPE program. Data from Wong et al.[1976] Github–Local

The narratives of daily human activity are constrained by the cost of moving in space, and the feasibility of separating related activities in time. The same constraints apply to mechanical systems, along with the ability to be manufactured at a profit.

The narratives achievable in a software system are constrained by the cognitive capacity, and knowledge of the people who implemented it, along with the ability to use the system within the available storage and processing capacity.

Languages support a variety of constructs for creating narrative components that can be fitted together, e.g., functions/methods, classes, and generics/templates. The purpose of generics/templates is to provide a means of specifying a general behavior that can later be instantiated for specific cases, e.g., the generic behavior is to return the maximum of a list of values, and a specific instance involves the values having an integer type.

A study by Chen, Wu, Ma, Zhou, Xu and Leung[343] investigated the use of C++ templates, such as the definition and use of new templates by developers, and the use of templates defined in the Standard Template Library. For the five largest projects: around 25% of

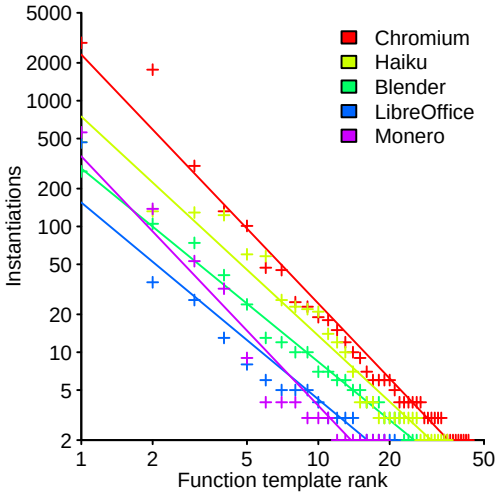


Figure 7.15: Sorted number of instantiations of each developer-defined C++ function template; fitted regression lines have the form: $Instantiations \propto template\_rank^{-K}$, where K is between 1.5 and 2. Data from Chen et al.[343] Github–Local

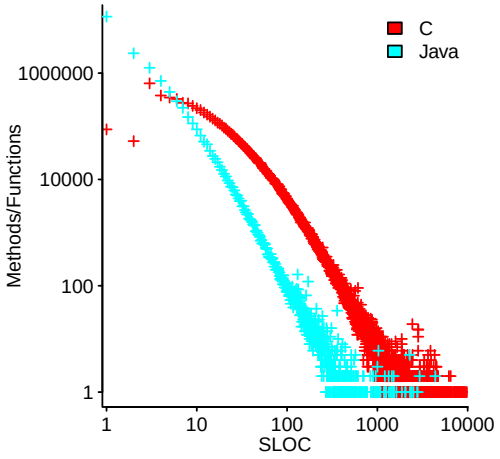


Figure 7.16: Number of methods/functions containing a given number of source lines; 17.6M methods, 6.3M functions. Data kindly provided by Landman.[1077] Github–Local

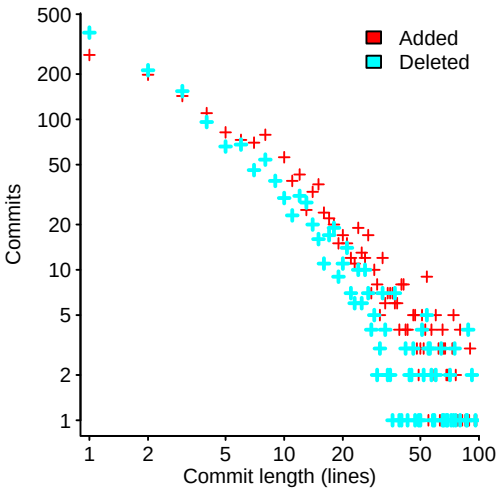


Figure 7.17: Number of commits of a given length, in lines added/deleted to fix various faults in Linux file systems. Data from Lu et al.[1164] Github–Local

developer-defined function templates and 15% of class templates were instantiated once, and there were seventeen times as many instantiations of function templates defined in the STL compared to developer-defined function templates (149,591 vs. 8,887). Figure 7.15 shows that a few developer-defined function templates account for most of the template instantiations in a project.

Reasons for the greater use of STL templates include: the library supports the commonly required functionality, and documentation on the available templates is readily available. Developer-defined templates are likely to be application specific, and documentation on them may not be readily available to other developers. The study found that most templates were defined by a few project developers, which may be an issue of developer education, or applications only needing specific templates in specific cases.

Studies of the introduction of generics in C♯[997] and Java[1447] found that while developers made use of functionality defined in libraries using generics, they rarely defined their own generic classes. Also, existing code in most projects was not refactored to use generics, and the savings from refactoring (measured in lines of code) was small.

How much source code appears in the implementation of distinct components of a narrative?

A study by Landman, Serebrenik, Bouwers and Vinju[1077] measured 19K open source Java projects, and the 9.6K packages (written in C) contained in a Linux distribution. Figure 7.16 shows the number Java methods and C functions containing a given number of source lines. While a power law provides a good fit, over some range, of both sets of data, the majority of C functions have a size distribution that differs from Java; see Github–sourcecode/Landman_m_ccsloc.R.

Figure 7.16 shows that most Java methods are very short, while the size range of the majority C functions is much wider (i.e., four to ten lines); figure 7.25 shows that 50% of Java source occurs within methods containing four lines or fewer, while in C 50% of source appears in functions containing 114 lines, or less.

One study[324] of function calls in eight C programs found, statically, that function calls in some programs were mostly to functions defined in other files, while calls in the other programs were to functions defined in the same file; the same static intra/inter file call predominance tended to occur at runtime.

Narratives are created and motivated by pressures such as:

- startups seeking to bring a saleable narrative to market as quickly as possible (i.e., a minimum viable product), to enable them to use customer feedback to enrich and extend the narrative,
- companies with established systems seeking to evolve the software to keep it consistent with the real-world narrative within which it has become intertwined,
- open source developers creating narratives for personal enjoyment.

Language tokens (such as identifiers, keywords and integer literals) are not the source code equivalent of words, but more like the phonemes (a distinct unit of sound) that are used to form a word. Most lines only contain a few tokens (see fig 8.4), and might form a distinct unit of thought or act as a connection between the adjacent lines.

7.2.3 Explaining code

Before a developer can successfully complete a source code related task, they have to invest in obtaining a good enough explanation of the behavior of the appropriate code. The nature of the task is likely to drive the approach the developer takes, to the explanation process[857] (the term *understanding* is often used by software developers, and *comprehension* is used in prose related research).

Human reasoning is discussed in section 2.6.

A small modification may only require understanding how the code implements the functionality it provides (e.g., algorithms used), while a larger change may require searching for existing code that could be impacted by the change; fixing a reported fault is a search process that often involves localised understanding.

Figure 7.17 shows the size of commits involved in fixing reported faults in Linux; also see fig 8.15.

For some tasks the cost-effectiveness of understanding the behavior of a complete program will decrease as the size of the program increases (because the task can be completed with an understanding of a subset of the program's behavior). One study[1154] based around a 250-line Fortran program found some developers used an as-needed strategy, and others attempted to understand the complete program.

To what extent do the contents of existing files affect the future coding habits of developers (because they read and learn from the source code contained in one or more of these files)?

A lower bound on the number of times a file has been written is provided by version control check-in history. It is not known, at the time of writing, whether check-outs can be used as a good-enough proxy for the number of times a file is read.

A study by Taylor[1817] investigated author contribution patterns to Eclipse projects. Figure 7.18 shows the number of files modified by a given number of people.

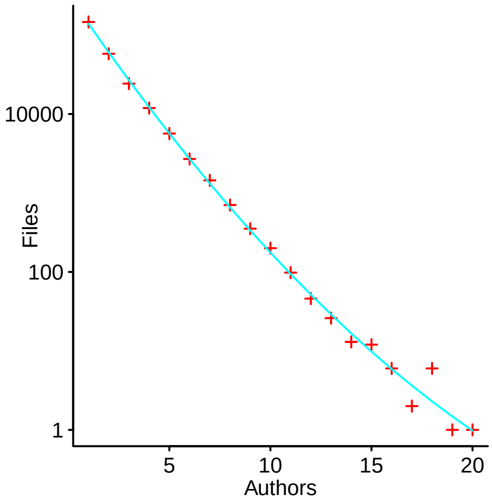


Figure 7.18: Number of files, in Eclipse projects, that have been modified by a given number of people; line is a fitted regression model of the form: $Files \propto e^{-0.87authors+0.01authors^2}$. Data from Taylor.[1817] Github–Local

Developers do not understand programs, as such, they acquire beliefs about program behavior; a continuous process involving the creation of new beliefs and the modification of existing ones, with no well-defined ending. The beliefs acquired are influenced by existing beliefs about the programming language it is written in, general computing algorithms, and the application domain.[1674]

People search for meaning and explanations.[982] Developers may infer an *intended meaning* of source code, i.e., a belief about what the meaning that the original author of the code intended to implement. Code understanding is an exercise in obtaining an intended meaning that is assumed to have existed.

Activities that appear to be very complicated, can have a simple, but difficult to discover, explanation. For instance, some hunting rituals intended to select the best hunting location are actually randomization algorithms,[1676] whose effect is to reduce the likelihood of the community over-hunting any location.

What can be done to reduce the cognitive effort that needs to be invested to obtain a good-enough interpretation of the behavior of code?

Source code is an implementation of application requirements. An understanding of the kinds of activities involved within the application domain provides a framework for guiding an interpretation of the intended behavior of a program's source.

A study by Bransford and Johnson[246] investigated the impact of having a top-level description on the amount of information subjects' remembered about a task. Try to give a meaning to the task described in the outer margin, while remembering what is involved (taken from the study).

The procedure is really quite simple. First you arrange things into different groups depending on their makeup. Of course, one pile may be sufficient depending on how much there is to do. If you have to go somewhere else due to lack of facilities that is the next step, otherwise you are pretty well set. It is important not to overdo any particular endeavor. That is, it is better to do too few things at once than too many. In the short run this may not seem important, but complications from doing too many can easily arise. A mistake can be expensive as well. The manipulation of the appropriate mechanisms should be self-explanatory, and we need not dwell on it here. At first the whole procedure will seem complicated. Soon, however, it will become just another facet of life. It is difficult to foresee any end to this task in the immediate future, but then one never can tell.

Table 7.3 shows that subjects' recalled over twice as much information, if they were given a meaningful phrase (the topic), before reading the passage. The topic of the passage in the margin is washing clothes.

	No Topic Given	Topic Given After	Topic Given Before	Maximum Score
Comprehension	2.29 (0.22)	2.12 (0.26)	4.50 (0.49)	7
Recall	2.82 (0.60)	2.65 (0.53)	5.83 (0.49)	18

Table 7.3: Mean comprehension rating and mean number of ideas recalled from passage (standard deviation in parentheses). Adapted from Bransford and Johnson.[246]

In one study[1466] investigating subject performance, answering questions about the code contained in a 200-line program they had studied; developers who had built an application domain model from the code performed best.

Some form of understanding may be achieved by assembling basic units of information into a higher level representation. In human languages, native speakers effortlessly operate on words, which are a basic unit of understanding. The complexity of human languages, which have to be processed in real-time while listening to the speaker, is constrained by the working memory capacity of those involved in the communication activity.[674,792] The capacity limits that make it difficult for speakers to construct complicated sentences, in real-time, are a benefit for listeners (who share similar capacity limits).

A study by Futrell, Mahowald and Gibson[641] investigated dependency length (the distance between words, in a sentence, that depend on each other; see figure 7.19) in the use of

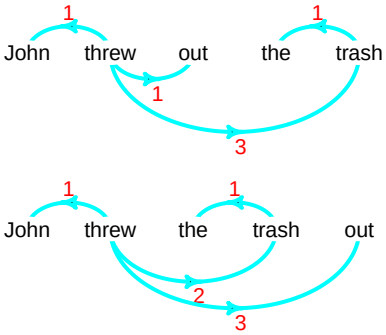


Figure 7.19: Two sentences, with their dependency representations; upper sentence has total dependency length six, while in the lower sentence it is seven. Based on Futrell et al.[641] Github–Local

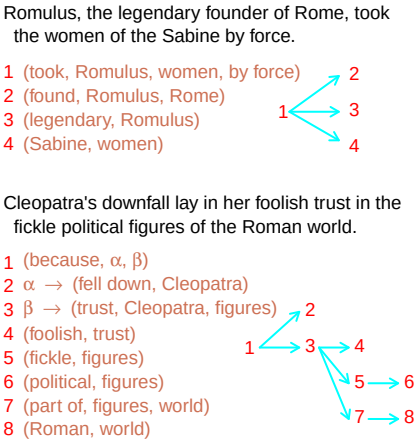


Figure 7.20: One sentence containing four, and the other eight propositions, along with their propositional analyses. Based on Kintsch et al.[1006] Github–Local

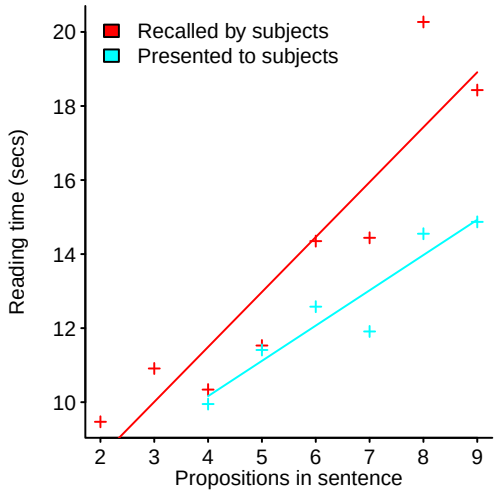


Figure 7.21: Mean reading time (in seconds) for sentences containing a given number of propositions, and as a function of the number of propositions recalled by subjects; with fitted regression models. Data extracted from Kintsch et al.[1006] Github–Local

37 human languages. The results suggest that speakers attempt to minimise dependency length.

Sentence complexity[349] has a variety of effects on human performance. A study by Kintsch and Keenan[1006] asked subjects to read single sentences, each containing the same number of words, but varying in the number of propositions they contained; see figure 7.20. The time taken to read each sentence and recall it (immediately after reading it), was measured.

Figure 7.21 shows reading time (in seconds) for sentences containing a given numbers of propositions (blue), and reading time for when a given number of propositions were recalled by subjects (red); with fitted regression models. A later study[1007] found that reader performance was also affected by the number of word concepts in the sentence, and the grammatical form of the propositions (subordinate or superordinate).

7.2.4 Memory for material read

Increasing the likelihood that information extracted from code will be accurately recalled later reduces the costs associated with having to reextract the information.

What do people remember about the material they have read (human memory systems are discussed in section 2.4)?

Again, the only detailed experimental data available is from studies based on human language prose.

Studies[245] have found that in the short term, syntax is remembered (i.e., words), while over the longer term mostly semantics is remembered (i.e., the meaning); explicit verbatim memory for text does occur.[759] Readers might like to try the following test (based on Jenkins[916]); part 1: A line at a time, 1) read the sentence on the left, 2) look away and count to five, 3) answer the question on the right, and 4) repeat process for the next line.

The girl broke the window on the porch.	Broke what?
The hill was steep.	What was?
The cat, running from the barking dog, jumped on the table.	From what?
The tree was tall.	Was what?
The old car climbed the hill.	What did?
The cat running from the dog jumped on the table.	Where?
The girl who lives next door broke the window on the porch.	Lives where?
The car pulled the trailer.	Did what?
The scared cat was running from the barking dog.	What was?
The girl lives next door.	Who does?
The tree shaded the man who was smoking his pipe.	What did?
The scared cat jumped on the table.	What did?
The girl who lives next door broke the large window.	Broke what?
The man was smoking his pipe.	Who was?
The old car climbed the steep hill.	The what?
The large window was on the porch.	Where?
The tall tree was in the front yard.	What was?
The car pulling the trailer climbed the steep hill.	Did what?
The cat jumped on the table.	Where?
The tall tree in the front yard shaded the man.	Did what?
The car pulling the trailer climbed the hill.	Which car?
The dog was barking.	Was what?
The window was large.	What was?

You have now completed part 1. Please do something else for a minute, or so, before moving on to part 2 (which follows immediately below).

Part 2: when performing this part, do not look at the sentences above, from part 1; look at the sentences below. Now, a line at a time, 1) read the sentence on the left, 2) if you think that sentence appeared as a sentence in part 1 express your confidence level by writing a number between one and five (with one expressing very little confidence, and five expressing a lot of confidence in the decision) next to **old**, otherwise write a number representing your confidence level next to **new**, and 3) repeat process for the next line.

The car climbed the hill.	old___, new ___
The girl who lives next door broke the window.	old___, new ___
The old man who was smoking his pipe climbed the steep hill.	old___, new ___
The tree was in the front yard.	old___, new ___

```
The window was on the porch.                                        old___, new ___
The barking dog jumped on the old car in the front yard.            old___, new ___
The cat was running from the dog.                                   old___, new ___
The old car pulled the trailer.                                     old___, new ___
The tall tree in the front yard shaded the old car.                 old___, new ___
The scared cat was running from the dog.                            old___, new ___
The old car, pulling the trailer, climbed the hill.                 old___, new ___
The girl who lives next door broke the large window on the porch.   old___, new ___
The tall tree shaded the man.                                       old___, new ___
The cat was running from the barking dog.                           old___, new ___
The cat was old.                                                    old___, new ___
The girl broke the large window.                                    old___, new ___
The car climbed the steep hill.                                     old___, new ___
The man who lives next door broke the window.                       old___, new ___
The cat was scared.                                                 old___, new ___
```

You have now completed part 2. Count the number of sentences you judged to be **old**.

The surprise is that all the sentences are new.

What is thought to happen is that while reading, people abstract and remember the general ideas contained in sentences. In this case, they are based on the four *idea sets*: 1) "The scared cat running from the barking dog jumped on the table.", 2) "The old car pulling the trailer climbed the steep hill.", 3) "The tall tree in the front yard shaded the man who was smoking his pipe.", and 4) "The girl who lives next door broke the large window on the porch.".

A study by Bransford and Franks[245] investigated subjects' confidence of having previously seen a sentence. Sentences contained either one idea unit (e.g., "The cat was scared."), two idea units (e.g., "The scared cat jumped on the table."), three idea units (e.g., "The scared cat was running from the dog."), or four idea units (e.g., "The scared cat running from the barking dog jumped on the table."). Subjects saw 24 sentences, after a 4-5 minute break they were shown 28 sentences (24 of which were new sentences), and asked to rank their confidence of having previously seen the sentence (on a 1 to 5 scale).

Figure 7.22 shows that subjects' confidence of having previously seen a sentence increases with the number of idea units it contains. New sentences contained one or more idea units contained in previously seen sentences. The results are consistent with subject confidence level being driven by the number of previously seen idea units in a sentence, rather than the presence of new idea units.

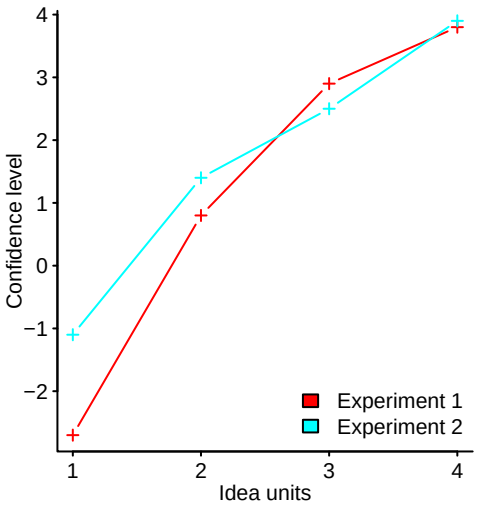


Figure 7.22: Subject confidence level, on a one to five scale (yes positive, no negative), of having previously seen a sentence containing a given number of idea units (experiment 2 was a replication of experiment 1, plus extra sentences). Data extracted from Bransford et al.[245] Github–Local

People use their experience with the form and structure of often repeated sequences of actions, to organize the longer-term memories they form about them. The following studies illustrate the effect that a person's knowledge of the world can have on their memory for what they have read, particularly with the passage of time, and their performance in interpreting sequences of related facts they are presented with:

- A study by Bower, Black and Turner[233] gave subjects a number of short stories describing various activities to read, such as visiting the dentist, attending a class lecture, going to a birthday party, i.e., scripts. Each story contained about 20 actions, such as looking at a dental poster, having teeth X-rayed, etc. After a 20-minutes interval, subjects were asked to recall actions contained in the stories.

  The results found that around a quarter of recalled actions might be part of the script, but were not included in the written story. Approximately 7% of recalled actions were not in the story, and would not be thought to belong to the script.

  A second experiment involved subjects reading a list of actions, which in the real world, would either be expected to occur in a known order or not be expected to have any order, e.g., the order of the float displays in a parade. The results showed that, within ordered scripts, actions that occurred at their expected location were recalled 50% of the time, while actions occurring at unexpected locations were recalled 18% of the time at that location. The recall rate for unordered scripts (i.e., the controls) was 30%.

- A study by Graesser, Woll, Kowalski and Smith[725] read subjects stories representing scripted activities, e.g., eating at a restaurant. The stories contained actions that varied in the degree to which they were typical of the script, e.g., Jack sat down at the table, Jack confirmed his reservation, and Jack put a pen in his pocket.

  Table 7.4 shows the results; recall was not affected by typicality over short periods of time, but after one week recall of atypical actions dropped significantly. Recognition

Memory Test	Typical (30 mins)	Atypical (30 mins)	Typical (1 week)	Atypical (1 week)
Recall (correct)	0.34	0.32	0.21	0.04
Recall (incorrect)	0.17	0.00	0.15	0.00
Recognition (correct)	0.79	0.79	0.80	0.60
Recognition (incorrect)	0.59	0.11	0.69	0.26

Table 7.4: Probability of subjects recalling or recognizing, typical or atypical actions present in stories read to them, at two time intervals (30 minutes and 1 week) after hearing them. Based on Graesser et al.[725]

performance (i.e., subjects were asked if a particular action occurred in the story) for typical vs. atypical actions was less affected by the passage of time.

- A study by Dooling and Christiaansen[506] asked subjects to read a short biography containing 10 sentences. The only difference between the biographies was that in some cases the name of the character was fictitious (i.e., a made up name), while in other cases it was the name of an applicable famous person. For instance, one biography described a ruthless dictator, and used either the name Gerald Martin or Adolph Hitler.

  After 2-days, and then after 1-week, subjects were given a list of 14 sentences (seven sentences that were included in the biography they had previously read, and seven that were not included), and asked to specify, which sentences they had previously read.

  To measure the impact of subjects' knowledge about the famous person, on recognition performance, some subjects were given additional information. In both cases the additional information was given to the subjects who had read the biography containing the made up name, e.g., Gerald Martin. The *before* subjects were told just before reading the biography that it was actually a description of a famous person and given that persons name, e.g., Adolph Hitler. The *after* subjects were told just before performing the recognition test that the biography was actually a description of a famous person and given that persons name (they were given one minute to think about what they had been told).

  Figure 7.23 shows that the results are consistent with the idea that remembering is constructive. After a week subjects memory for specific information in the passage was lost, and under these conditions sentence recognition is guided by subjects' general knowledge. Variations in the time between reading the biography, and identity of a famous character being revealed, affected the extent to which subjects integrated this information.

These results suggest that it is a desirable characteristic (i.e., more information, more accurately recalled), for the contents of scripted stories to be consistent with readers' prior knowledge and expectations, e.g., events that occur and their relative ordering. The extent to which source code can be organised in this way will depend on the application requirements and any demands for algorithmic efficiency.

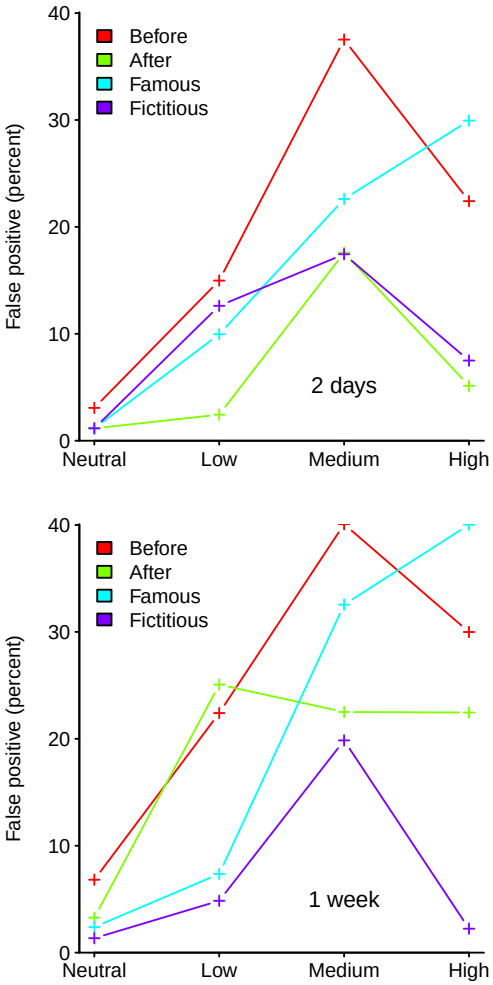


Figure 7.23: Percentage of false-positive recognition errors for biographies having varying degrees of thematic relatedness to the famous person, in *before*, *after*, *famous*, and *fictitious* groups. Data extracted from Dooling et al.[506] Github–Local

7.2.5 Integrating information

Units of source code (e.g., statements) are sequenced in ways that result in a behavioral narrative emerging during program execution. To modify an existing narrative a developer needs to acquire a good enough understanding of how the units of code are sequenced to produce the emergent behavior. Information has to be extracted and integrated into a mental model of program operation.

Which factors have the largest impact on the cognitive effort needed to integrate source code information into a mental model? Studies[1005,1246] of prose comprehension provide some clues.

The process of integrating two related items of information involves acquiring the first item, and keeping it available for recall while processing other information, until the second item is encountered; it is then possible to notice that the two items are related, and act on this observation.

A study by Daneman and Carpenter[432] investigated the connection between various measures of subjects' working memory span, and their performance on a reading comprehension task. The two measures of working memory used were the *word span* and *reading span*. The word span test is purely a measure of memory usage. In the reading span test subjects have to read, out loud, sequences of sentences while remembering the last word

of each sentence, which have to be recalled at the end of the sequence. In the test, the number of sentences in each sequence is increased until subjects are unable to successfully recall all the last words.

The reading comprehension test involves subjects reading a narrative passage containing approximately 140 words, and then answering questions about facts and events described in the passage. Passages are constructed such that the distance between the information needed to answer questions varies. For instance, the final sentence of the passage might contain a pronoun (e.g., she, her, he, him, or it) referring to a noun appearing in a previous sentence, with different passages containing the referenced noun in either the second, third, fourth, fifth, sixth, or seventh sentence before the last sentence.

In the excerpt: " . . . river clearing . . . The proceedings were delayed because the leopard had not shown up yet. There was much speculation as to the reason for this midnight alarm. Finally, he arrived, and the meeting could commence." the question: "Who finally arrived?" refers to information contained in the last and third to last sentence; the question: "Where was the meeting held?" requires the recall of a fact.

Figure 7.24 show the relationship between subject performance in the reading span test and the reading comprehension test. A similar pattern of results was obtained when the task involved listening, rather than reading. A study by Turner and Engle[1857] found that having subjects verify simple arithmetic identities, rather than a reading comprehension test, did not alter the results. However, altering the difficulty of the background task (e.g., using sentences that required more effort to comprehend) reduced performance.

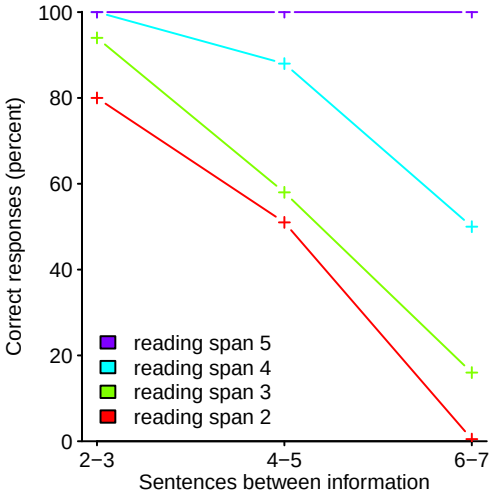


Figure 7.24: Percentage of correct responses in a reading comprehension test, for subjects having a given reading span, using the pronoun reference questions as a function of the number of sentences (x-axis) between the pronoun and the referent noun. Data extracted from Daneman et al.[432] Github–Local

As a coding example, given the following three assignments, would moving the assignment to x after the assignment to y, reduce the cognitive effort needed to comprehend the value of the expression assigned to z?

```
x = ex_1 + ex_2;            /* Could be moved to after assignment to y. */
y = complicated_expression; /* No dependencies on previous statement.   */
z = y + ex_1;
```

This question assumes that ex_2 does not appear prior to the assignment to x, in which case there may be a greater benefit to this assignment appearing close to the prior usage, rather than close to the assignment to z; at the time of writing there is little if any evidence available that might be used to help answer these questions.

Is reader cognitive effort reduced by having a single complex statement, rather than several simpler statements?

A study by Daneman and Carpenter[433] investigated subjects performance when integrating information within a single sentence, compared to across two sentences, e.g., "There is a sewer near our home who makes terrific suits" (this is what is known as a *garden path* sentence), and "There is a sewer near our home. He makes terrific suits." The results found that a sentence boundary can affect comprehension performance. It was proposed that this performance difference was caused by readers purging any verbatim information they held in working memory, about a sentence, on reaching its end. The availability of previously read words, in the single sentence case, making it easier to change an interpretation, based of what has already been read.

Putting too much information in one sentence has costs. A study by Gibson and Thomas[675] found that subjects were likely to perceive complex ungrammatical sentences as being grammatical. Subjects handled complex sentence that exceeded working memory capacity by forgetting parts of the syntactic structure of the sentence, to create a grammatically correct sentence.

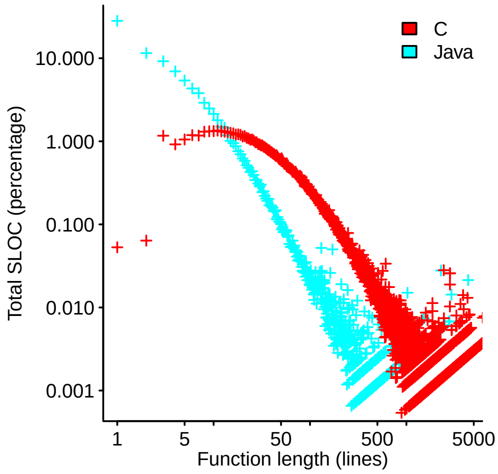
A study by Kintsch, Mandel, and Kozminsky[1008] investigated the time taken to read and summarize 1,400 word stories. The order of the paragraphs (not the sentences) in the text seen by some subjects was randomized. The results showed that while it was not possible to distinguish between the summaries produced by subjects reading ordered vs. randomised stories, reading time for randomly ordered paragraphs was significantly longer (9.05 minutes vs. 7.34).

A study by Ehrlich and Johnson-Laird[533] asked subjects to draw diagrams depicting the spatial layout of everyday items specified by a sequence of sentences. The sentences varied in the extent to which an item appearing as the object (or subject, or not at all) in one sentence appeared as the subject (or object, or not at all) in the immediately following sentence. For instance, there is referential continuity in the sentence sequence "The knife is in front of the pot. The pot is on the left of the glass. The glass is behind the dish.", but

not in the sequence "The knife is in front of the pot. The glass is behind the dish. The pot is on the left of the glass."

The results found that when the items in the sentence sequences had referential continuity 57% of the diagrams were correct, compared to 33% when there was no continuity. Most of the errors for the non-continuity sentences were items missing from the diagram drawn, and subjects reported finding it difficult to remember the items as well as the relationship between them.

Most functions contain a few lines (see fig 7.16), and figure 7.25 shows that, depending on language, most of a program's code appears in the shorter functions.



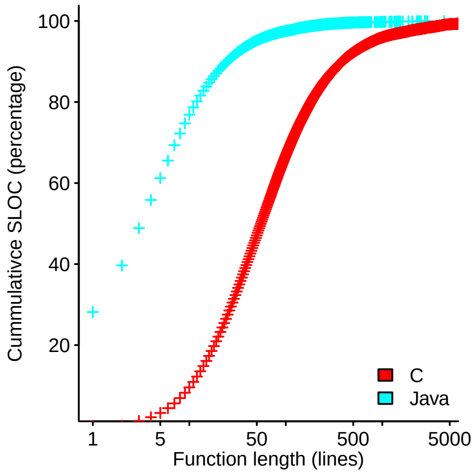


Figure 7.25: Lines of code (as a percentage of all lines of code in the language measured) appearing in C functions and Java methods containing a given number of lines of code (upper); cumulative sum of SLOC percentage (lower). Data kindly provided by Landman.[1077] Github–Local

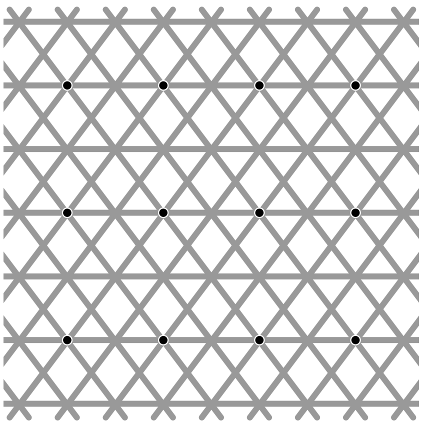


Figure 7.26: Hermann grid, with variation due to Ninio and Stevens[1384] to create an extinction illusion. Github–Local

7.2.6 Visual organization

The human brain contains several regions that perform specific kinds of basic processing of the visual input, along with regions that use this processed information to create higher level models of the visual scene; see section 2.3.

High level visual attention is a limited resource, as illustrated by some of the black circles in figure 7.26 not being visible when not directly viewed. How might the visual layout of source code be organized to reduce the need for conscious attention, by making use of the lower level processing capability that is available (e.g., indenting the start of adjacent lines to take advantage of preattentive detection of lines)?

People's ability to learn means that, with practice, they can adapt to handle a wide variety of disparate visual organizations of character sequences. The learning process requires practice, which takes time. Using a visual organization likely to be familiar to developers reduces the start-up cost of adapting to a new layout, i.e., prior experience is used to enable developer performance to start closer to their long-term performance.

How quickly might people achieve a reading performance, using a new text layout, that is comparable to that achieved with a familiar layout (based on reading and error rate)?

A study by Kolers and Perkins[1034] investigated the extent to which practice improved reading performance. Subjects were asked to read pages of text written in various ways, and the time taken for subjects to read a page of text having a particular orientation was measured; the text could be one of: normal, reversed, inverted, or mirrored text, as in the following:

- Expectations can also mislead us; the unexpected is always hard to perceive clearly. Sometimes we fail to recognize an object because we...
- .ekam ot detcepxe eb thgim natiruP dnalgnE weN a ekatsim fo dnik eht saw tI .eb ot serad eh sa yzal sa si nam yreve taht dias ecno nosremE
- These are but a few of the reasons for believing that a person cannot be conscious of all his mental processes. Many other reasons can be...
- Several years ago a professor who teaches psychology at a large university had to ask his assistant, a young man of great intelligence...

Figure 7.27 shows the time taken to read a page containing text having a particular orientation. In a study[1033] a year later, Kolers measured the performance of the same subjects, as they read more pages. Performance improved with practice, but this time the subjects had prior experience, and their performance started out better and improved more quickly.

Eye-tracking is used in studies of reading prose to find out where subjects are looking, and for how long; this evidence-based approach is only just starting to be used to study the visual processes involved in processing source code (see fig 2.17), and the impact of factors such as indentation.[148]

7.2.7 Consistency

People subconsciously learn and make use of patterns in events that occur within their environment; see section 2.5. Consistently following patterns of behavior, when writing source code, creates an opportunity for readers to make use of this implicit learning ability. Individual developers write code in a distinct style,[292] even if they cannot articulate every pattern of behavior they follow.

A study by Lewicki, Hill and Bizot[1120] investigated the impact of implicit learning on subjects' performance, in a task containing no overt learning component. While subjects watched a computer screen, a letter was presented in one of four possible locations; subjects had to press the button corresponding to the location of the letter as quickly as possible. The sequence of locations used followed a consistent, but complex, pattern. The results showed subjects' response times continually improving as they gained experience. The presentation was divided into 17 segments of 240 trials (a total of 4,080 letters). The pattern used to select the sequence of locations was changed after the 15th segment, but subjects were not told about the existence of any patterns of behavior. After completing the presentation subjects were interviewed to find out if they had been aware of any patterns in the presentation; they had not.

Figure 7.28 shows the mean response time for each segment. The consistent improvement, after the first segment, is interrupted by a decrease in performance after the pattern changes on the 15th segment.

A study by Buse and Weimer[284] investigated Computer Science students' opinions of the readability of short snippets of Java source code, rating them on a scale of 1 to 5. The students were taking first, second and third/fourth year Computer Science degree courses or were postgraduates at the researchers' University.

Subjects were not given any instructions on how to rate the snippets for readability, and the attributes that subjects were evaluating when selecting a rating is not known, e.g., were subject ratings based of how readable they personally found the snippets to be, or based on the answer they would expect to give when tested in an exam.

The results show that the agreement between students readability ratings, for short snippets of code, improved as students progressed through course years 1 to 4 of a computer science degree; see Github–developers/readability. The study can be viewed as an investigation of implicit learning, i.e., students learned to rate code against what they had been told were community norms of a quantity called readability.

A study by Corazza, Maggio and Scanniello[400] investigated semantic relatedness, which they called *coherence*, between a method's implementation and any associated comment, i.e., did the method implement the intent expressed in the comment. Five Java programs, containing a total of 5,762 methods, were manually evaluated; the results found that coherence was positively correlated with $\log(comment\_lines)$, and negatively correlated with method *LOC*; see Github–sourcecode/SQJ_2015.R.

A study by Martinez and Monperrus[1213] investigated the kind of changes made to source to fix reported faults. The top five changes accounted for 30% of all changes, i.e., add method call, add `if-statement`, change method call, delete method call, and delete `if-statement`. Figure 7.29 shows kind of source changes ranked by percentage occurrence, and exponentials fitted over a range of ranks (red lines).

Software is created within a particular development culture, and differences can exist between different cultures. Consistency of culture (as in unchanging) is only a good thing while it provides good fit for the environment in which it operates.

Embedded software runs on resource limited hardware, which is often mass-produced, and saving pennies per device can add up to a lot of money. Systems are populated with the smallest amount of memory needed to run the code, and power consumption is reduced by using the slowest possible clock speeds, e.g., closer to 1 MHz than 1 GHz.

Experienced embedded developers are aware of hardware performance limitations they have to work within. Many low-cost processors have a very simple architecture with relatively few instructions and parameter passing, to a function, can be very expensive (in execution time and code size) compared to passing values to functions in global variables on some processors.

A study by Engblom[545] investigated differences in the characteristics of embedded C software, and the SPECint95 benchmark. Figure 7.30 shows the percentage of function definitions containing a given number of parameters, for embedded software the SPECint95 benchmark and desktop software measured by Jones.[930] A Poisson distribution provides a reasonable fit to both sets of data; for desktop software, the Poisson distribution fitted to function definitions having a given number of parameters has $\lambda = 2$, while for embedded developers $\lambda = 0.8$.

These measurements were of source code from the late 1990s; have embedded systems processor characteristics changed since then?

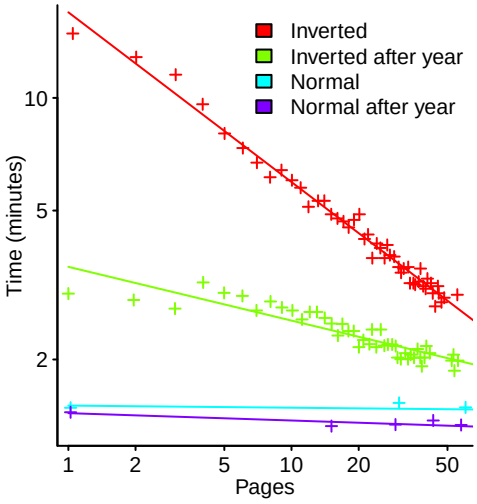


Figure 7.27: Time taken by subjects to read a page of text, printed with a particular orientation, as they read more pages (initial experiment and repeated after one year); with fitted regression lines. Results are for the same six subjects in two tests more than a year apart. Based on Kolers.[1033] Github–Local

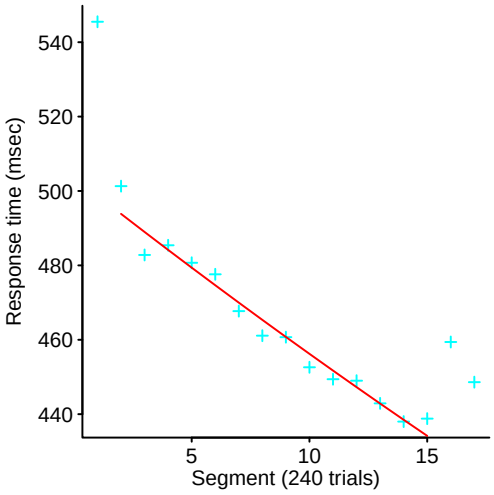


Figure 7.28: Mean response time for each of 17 segments; the regression line fitted to segments 2-15 has the form: $Response\_time \propto e^{-0.1Segment}$. Data extracted from Lewicki et al.[1120] Github–Local

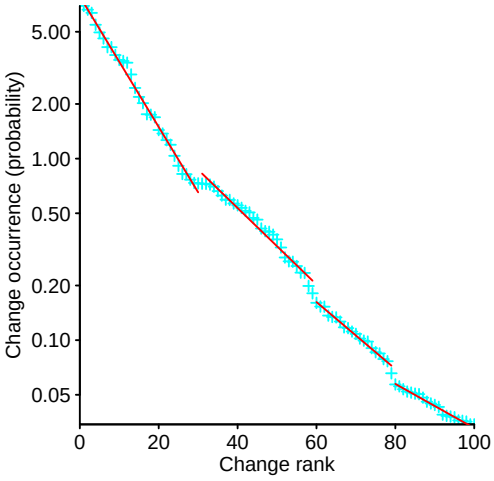


Figure 7.29: Percentage occurrence of kinds of source changes (in rank order), with fitted exponentials over a range of ranks (red lines). Data kindly provided by Martinez.[1213] Github–Local

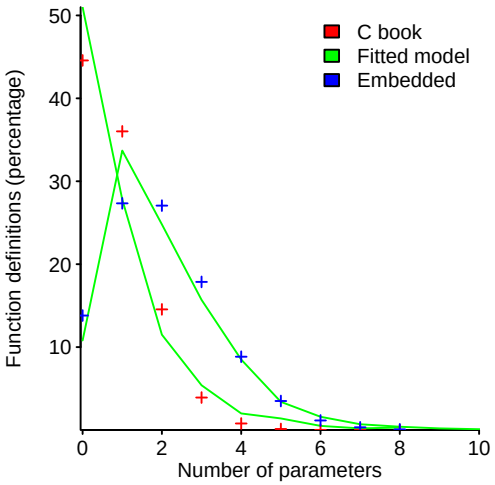


Figure 7.30: Percentage of function definitions declared to have a given number of parameters in: embedded applications, and the translated form of a sample of C source code. Data for embedded applications kindly supplied by Engblom,[545] C source code sample from Jones.[930] Github–Local

Today, companies are likely to be just as interested in profit, e.g., saving pennies. Compilers may have become better at reducing function parameter overheads for some processor, but it is beliefs that drives developer usage.

Embedded devices have become more mainstream, with companies selling IoT devices with USB interfaces. This availability provides an opportunity for aspects of desktop and mobile system development culture to invade the culture of embedded development. In some cases, where code size or/and performance is critical, developers looking for savings may learn about the overheads of parameter passing. Within existing embedded system communities, past folklore may no longer apply (because the hardware has changed).

Is there a range of values of λ, depending on developer experience (old habits die hard and parameter overhead will depend on processor characteristics, e.g., 4-bit, 8-bit and 16-bit processors)?

7.2.8 Identifier names

Identifier names provide a channel through which the person writing the code can communicate information to subsequent readers. The communications channel operates through the semantic associations triggered in the mind of a person as they read the source (tools might also attempt to gather and use this semantic information).

Semantic associations may be traceable to information contained in the source (e.g., the declared type of an identifier), or preexisting cultural information present in writers' or readers' memory, e.g., semantic interpretations associated with words in their native language within the culture it was learned and used.

Given that most functions are only ever modified by the original author (see fig 7.18), the primary beneficiary of any investment in naming of local identifiers is likely to be the developer who created them.

```
#      <     . >

#               13
#               0
#               1

              (          [],
                 *         )
{
   ,
        ;

*        =         ;
     =     (       );
  (       >          )
  {
  *         =          ;
  }

  {
     ( =0;   <        ;  ++)
    {
      ((       [ ] < '0') ||
       (       [ ] > '9'))
      {
      *         =          ;
      }
    }
  }
}
```

```
 include  string h

 define MAX_CNUM_LEN
 define VALID_CNUM
 define INVALID_CNUM

int chk_cnum_valid char cust_num
                 int  cnum_status

int i
   cnum_len

 cnum_status VALID_CNUM
cnum_len strlen cust_num
if  cnum_len   MAX_CNUM_LEN

   cnum_status INVALID_CNUM

else

  for  i    i   cnum_len  i

    if   cust_num i
         cust_num i

        cnum_status INVALID_CNUM
```

```
#include <string.h>

#define v1 13
#define v2  0
#define v3  1

int v4(char v5[],
      int *v6)
{
int v7,
   v8;

*v6=v2;
v8=strlen(v5);
if (v8 > v1)
  {
  *v6=v3;
  }
else
  {
  for (v7=0; v7 < v8; v7++)
    {
    if ((v5[v7] < '0') ||
        (v5[v7] > '9'))
      {
      *v6=v3;
      }
    }
  }
}
```

Figure 7.31: Three versions of the source of the same program, showing identifiers, non-identifiers and in an anonymous form; illustrating how a reader's existing knowledge of English word usage can reduce the cognitive effort needed to comprehend source code. Based on an example from Laitinen.[1069]

Identifiers are the most common token in source code (29% of the visible tokens in `.c` files,[930] with comma the second most common at 9.5%), and they represent approximately 40% of all non-white-space characters in the visible source (comments representing 31% of the characters in `.c` files).

Each identifier appearing in the visible source is competing for developer cognitive resources. Identifiers having similar spellings, pronunciations, or semantic associations may generate confusion, resulting in mistaken interpretations being made; identifiers with long names may consume cognitive resources that are out of proportion to the benefits of the information they communicate. Figure 7.32 shows the number of function definitions containing a given number of occurrences of identifiers (blue/green), and of distinct identifiers (red).

The meanings associated with a word, by a community, evolves,[1066] with different changes sometimes occurring in different geographic communities. One study[434] found people following a two-stage lifecycle: a linguistically innovative learning phase during which members align with the language of the community, followed by a conservative phase in which members don't respond to changes in community norms.

The same word may trigger different semantic associations in different people. For instance, the extent to which a word is thought to denote a concrete or abstract concept[1478] (*concrete* words, defined as things or actions in reality, experienced directly through the senses, whereas *abstract* words are not experienced through the senses, they are language-based with their meaning depending on other words). What is the probability that an identifier will trigger the same semantic associations in the mind of readers, when they encounter the identifier in code?

A study by Nelson, McEvoy and Schreiber[1363] investigated free association of words. Subjects were given a word, and asked to reply with the first word that came to mind. More than 6,000 subjects producing over 700,000 responses to 5,018 stimulus words.

What is the probability that the same response word will be given by more than one subject? Figure 7.33 shows the probability (averaged over all words) that a given percentage of subjects will give the same word in response to the same cue word (values were calculated for each word, for two to ten subjects, and normalised by the number of subjects responding to that word). The mean percentage of unique responses was 18% (sd 9).

In this study subjects were not asked to think about any field of study and were mostly students, i.e., were not domain experts. Domain experts may be more likely to agree on a response, for terms specific to their domain.

Table 7.5 shows the percentage of identifiers occurring in each pair of seven large software systems; top row is the total number of identifiers in the visible source of each system.

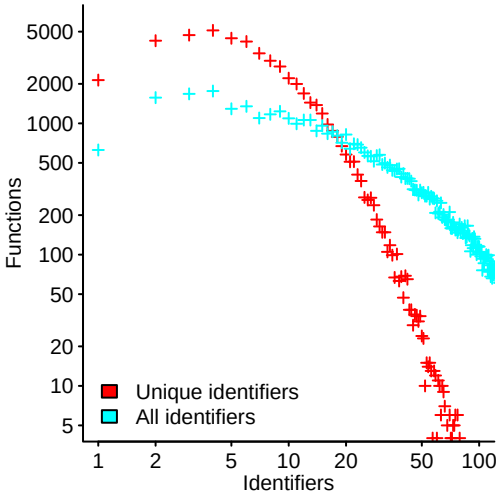


Figure 7.32: Number of C function definitions containing a given number of identifier uses (unique in red, all in blue/green). Data from Jones.[930] Github–Local

	gcc	idsoftware	linux	netscape	openafs	openMotif	postgresql
identifiers	46,549	27,467	275,566	52,326	35,868	35,465	18,131
gcc	-	2	9	6	5	3	3
idsoftware	5	-	8	6	5	4	3
linux	1	0	-	1	1	0	0
netscape	5	3	8	-	5	7	3
openafs	6	4	12	8	-	3	5
openMotif	4	3	6	11	3	-	3
postgresql	9	5	12	11	10	6	-

Table 7.5: Percentage of identifiers in one program having the same spelling as identifiers occurring in various other programs. First row is the total number of identifiers in the program, and the value used to divide the number of shared identifiers in that column. Data from Jones.[930]

Identifiers do not appear in isolation, in source, they appear within the context of other code. One study[932] found that identifier names can have a large impact on decisions made about the relative precedence of binary operators in an expression. Also, naming inconsistencies between the identifier passed as an argument, and the corresponding parameter has been used to find coding mistakes.[1578]

A study[1677] of word choice in a fill-in-the-blank task (e.g., "in tracking the . . . "), found that probability of word occurrence (derived from large samples of language use) was a good predictor of both the words chosen, and the order in which subjects produced them (subjects were asked to provide 20 responses per phrase).

English pronunciation can be context dependent, e.g., "You can lead a horse to water, but a pencil must be lead." and "Wind the clock when the wind blows."

Speakers of different natural languages will have trained on different inputs, and during school people study different subjects (each having its own technical terms). A study by Gardner, Rothkopf, Lapan, and Lafferty[652] asked subjects (10 engineering, 10 nursing, and 10 law students) to indicate whether a letter sequence was a word or a nonword. The words were drawn from a sample of high frequency words (more than 100 per million), medium-frequency (10–99 per million), low-frequency (less than 10 per million), and occupationally related engineering or medical words.

The results showed engineering subjects could more quickly and accurately identify the words related to engineering (but not medicine); the nursing subjects could more quickly and accurately identify the words related to medicine (but not engineering). The law

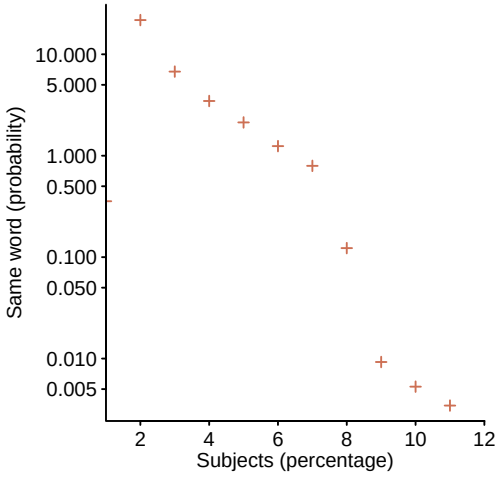


Figure 7.33: Probability (averaged over all cue words) that, for a given cue word, a given percentage of subjects will produce the same word. Data from Nelson et al.[1363] Github–Local

students showed no response differences for either group of occupationally related words. There were no response differences on identifying nonwords. The performance of the engineering and nursing students on their respective occupational words was almost as good as their performance on the medium-frequency words.

Object naming has been found to be influenced by recent experience,[1725] practical skills (e.g., typists selecting pairs of letters that they type using different fingers[1871]) and egotism, e.g., a preference for letters in one's own name or birthday related numbers.[1013, 1396]

Developers may select the same identifier for different reasons. A study[172] of the use of single letter identifiers in five languages found that `i` was by far the most common in source code written in four of the languages. This choice might be driven by abbreviating the words `integer` (the most common variable type) or `index`, or by seeing this usage in example code in books and on the web, or because related single letters were already used.

Desirable characteristics in an identifier name include: high probability of triggering the appropriate semantic associations in the readers' mind, a low probability of being mistaken for another identifier present in the associated source code, and consuming cognitive resources proportional to the useful information it is likely to communicate to readers.

Studies of the characteristics of words, in written form, found to have an effect on some aspect of subject performance include: *word length effect*,[1368] age of acquisition[405, 1648] (when the word was first learned), frequency of occurrence[98] (e.g., in spoken form and various kinds of written material), articulatory features of the initial phoneme (listener analysis of a spoken word is initiated when the first sounds are heard; differences at the beginning enable distinct words to be distinguished sooner).

Most studies of words have investigated English, but a growing number of large scale studies are investigating other languages.[597] Orthography (the spelling system for the written form of a language) can have an impact on reader performance, English has a deep orthography (i.e., a complex mapping between spelling and sound), while Malay has a shallow orthography (i.e., a one-to-one mapping between spelling and sound; also Spanish), and word length in Malay has been found to be a better predictor of word recognition than word frequency.[1991]

When creating a spelling for an identifier, a path of least cognitive effort is for developers to rely on their experience of using their own native language, e.g., lexical conventions, syntax,[302] word ordering conventions (adjectives). The mistakes made by developers, in the use of English, for whom English is not a native language are influenced by the characteristics of their native language.[1802]

What are the characteristics likely to increase the likelihood that an identifier will be mistaken for another one?

Dearest creature in creation,
Study English pronunciation.
I will teach you in my verse
Sounds like corpse, corps, horse, and worse.
I will keep you, Suzy, busy,
Make your head with heat grow dizzy.
Tear in eye, your dress will tear.
So shall I! Oh hear my prayer.
Pray, console your loving poet,
Make my coat look new, dear, sew it!

Just compare heart, beard, and heard,
Dies and diet, lord and word,
Sword and sward, retain and Britain.
(Mind the latter, how it's written.)
Now I surely will not plague you
With such words as plaque and ague.
But be careful how you speak:
Say break and steak, but bleak and streak;
Cloven, oven, how and low,
Script, receipt, show, poem, and toe.

THE CHAOS (first two verses)
by Dr. Gerard Nolst Trenité, 1870-1946

A study by Lambert, Chang, and Gupta[1072] investigated drug name confusion errors.[x] Subjects briefly saw the degraded image of a drug name. Both the frequency of occurrence of drug names, and their neighborhood density were found to be significant factors in subject error rate.

An analysis of the kinds of errors made found that 234 were omission errors and 4,128 were substitution errors. In the case of the substitution errors, 63.5% were names of other drugs (e.g., Indocin® instead of Indomed®), with the remaining substitution errors being spelling-related or other non-drug responses, e.g., Catapress instead of Catapres®.

Identifiers often contain character sequences that do not match words in the native language of the reader. Studies of prose have included the use non-words, often as a performance comparison against words, and nonwords are sometimes read as a word whose spelling it closely resembles.[1515]

Studies of letter similarity have a long history,[1836] and tables of visual[1322] (e.g., `1` (one) and `l` (ell)) and acoustic[1480] letter confusion have been published. When categorizing a stimulus, people are more likely to ignore a feature than they are to add a missing feature, e.g., **Q** is confused with **O** more often than **O** is confused with **Q**.

A study by Avidan[96] investigated how look it took subjects to work out what a Java method did, recording the time taken to reply. In the control condition subjects saw the original method, and in the experimental condition the method name was replaced by xxx,

[x] Errors involving medication kill one person every day in the U.S., and injure more than one million every year; confusion between drug names that look and sound alike account for 15% to 25% of reported medication errors

with local and/or parameter names replaced by single letter identifiers; in all experimental conditions the method name was replaced by xxx.

Subjects took longer to reply for the modified methods. When parameter names were left unmodified, subjects were 268 seconds slower (on average), and when locals were left unmodified 342 seconds slower (the standard deviation of the between subject differences was 187 and 253 seconds, respectively); see Github–sourcecode/Avidan-MSc.R.

A study[310] of the effectiveness of two code obfuscation techniques (renaming identifiers and complicating the control flow) found that renaming identifiers had the larger impact on the time taken by subjects to comprehend and change code; see Github–sourcecode/AssessEffectCodeOb.R.

One study[841] found that the time taken to find a mistake in a short snippet of code was slightly faster when the identifiers were words, rather than non-words; see Github–sourcecode/shorter-iden.R.

The names of existing identifiers are sometimes changed.[73] The constraints on identifier renaming include the cost of making all the necessary changes in the code and dependencies other software may have on existing names, e.g., identifier is in a published API.

One study[1520] created a tool that learned patterns of identifier usage in Javascript, which then flagged identifiers whose use in a given context seemed unlikely to be correct.

7.2.9 Programming languages

Thousands of programming languages have been created, and new languages continue to be created (see section 4.6.1); they can be classified according to various criteria.[1880]

Do some programming languages require more, or less, effort from developers, to write code having any of the desirable characteristics discussed in this chapter?

Every programming language has its fans, people who ascribe various positive qualities to programs written in this language, or in languages supporting particular characteristics. There is little or no experimental evidence for any language characteristics having an impact on developer performance, and even less evidence for specific language features having a performance impact.

The term *strongly typed* is applied as a marketing term to languages believed by the speaker to specify greater than some minimum amount of type checking. The available experimental evidence for the possible benefits of using strongly typed languages is discussed in section 7.3.6. Languages provide functionality and developers can choose to make use of it, or not. It would be more appropriate to apply the term strongly typed to source code that takes full advantage of the type checking functionality provided by a language.

There is often a mechanism for subverting a language's built-in type checks, e.g., the use of **unconstrained** in Ada, the `unsafe` package in Go,[407] and the **unsafe** keyword in Rust.[563]

Factors that might generate a measurable difference in developer performance, when using different programming languages, include: individual knowledge and skill using the language, and interaction between the programming language and problem to be solved, e.g., it may be easier to write a program to solve a particular kind of problem using language X than using language Y.

Studies[1352,1876,2031] that compare languages using small programs suffer from the possibility of a strong interaction between the problem being solved, the available language constructs (requiring a sample containing solutions to a wide variety of problems), and the developers' skill at mapping the problem constructs available in the language used. Some languages include support for programming in the large (e.g., sophisticated separate compilation mechanisms), and studies will need to use large programs to investigate such features.

A study by Back and Westman[106] investigated the characteristics of the 220,349 entries submitted to the Google code jam program competition for the years 2012-2016 (a total of 127 problems). Figure 7.34 shows the number of solutions containing a given number of lines for one of the problems (the one having the most submitted solution: 2,624), stratified by the five most commonly used languages.

A realistic comparison of two languages requires information from many implementations of large systems targeting the same application domain.

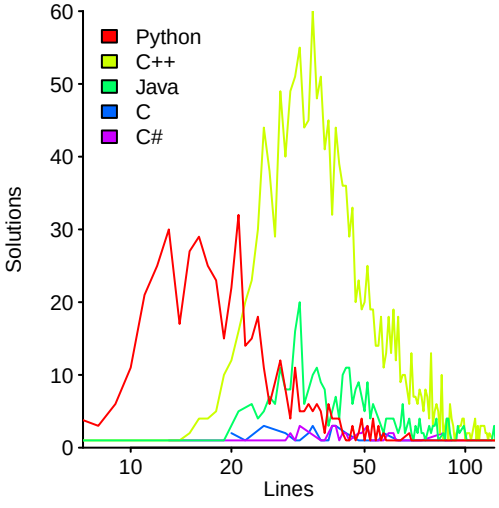


Figure 7.34: Number of solutions to one a problem posed in a Google code jam competition, containing a given number of lines, stratified by programming language. Data from Back et al.[106] Github–Local

A study by Waligora, Bailey and Stark[1917] compared various lines of code measurements (e.g., measurements of declarations, executable statements and code reuse) of 11 Ada and 19 Fortran implementations of NASA ground station software systems. While there were differences in patterns of behavior for various line counts, these differences could have been caused by factors outside the scope of the report (e.g., the extent to which individual companies were involved in multiple projects and in a good position to evaluate the potential for reuse of code from previous projects, or the extent to which project requirements specified that code should be written in a way likely to make it easier to reuse); see Github–projects/nasa-ada-fortran.R.

Differences in performance between subjects, and learning effects, can dominate studies based on small programs, or experiments run over short intervals. It is possible to structure an experiment such that subject performance improvement, on each reimplementation (driven by learning that occurred on previous implementations), is explicitly included as a variable; see section 11.6.

A study by Savić, Ivanović, Budimac and Radovanović[1636] investigated the impact of a change of teaching language on student performance in practical sessions (moving from Modula-2 to Java). Student performance, measured using marks assigned, was unchanged across the four practical sessions, as was mean score for each year; see Github–ecosystems/2016-sclit-uup.R.

A study by Prechelt and Tichy[1525] investigated the impact of parameter checking of function calls (when the experiment was performed, C compilers that did not perform any checking on the arguments passed to functions, so-called K&R style, were still in common use). All subjects wrote two programs: one program using a compiler that performed argument checking of function calls, and the second program using a compiler that did not perform this checking. Subjects were randomly assigned to the problem to solve first, and the compiler to use for each problem. The time to complete a correct program was measured.

Fitting a regression model to the results shows that the greatest variation in performance occurred between subjects (standard deviation of 74 minutes), the next largest effect was problem ordering (with the second problem being solved 38 minutes, on average, faster than the first). The performance improvement attributed to argument checking is 12 minutes, compared to no argument checking; see Github–experiment/tcheck98.R.

Studies[1286] investigating the use of a human language, to specify solutions to problems, have found that subjects make extensive use of the contextual referencing that is common in human communication. This use of context, and other issues, make it extremely difficult to automatically process the implementation instructions.

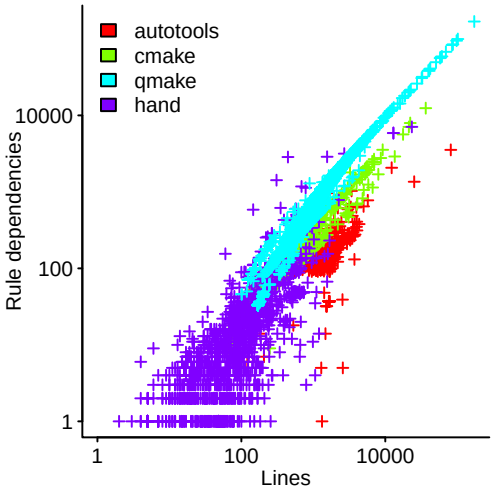


Figure 7.35: Number of lines against number of dependencies contained in rules, in 19,689 makefiles, stratified by method of creation. Data from Martin.[1210] Github–Local

Programming languages that support coding constructs at a level of abstraction higher than machine code have to make some implementation decisions about lower level behavior, e.g., deciding the address of variables defined by the developer, and the amount of storage allocated to hold them. These implementation decisions are implicit behavior.

Studies[1045] have investigated the use of particular kinds of implicit behavior, and fault repositories provide evidence that some coding mistakes are the result of developers not understanding the implicit behavior present in a particular sequence of code.

Your author is not aware of any evidence-based studies showing that requiring all behavior to be explicit, in the code, is more/less cost effective, i.e., supporting implicit behavior is less/more costly than the cost of [the assumed] more coding mistakes. Neither is your author aware of any evidence-based studies showing that alternative implicit behavior result in fewer developer mistakes.

7.2.10 Build bureaucracy

Software systems are built by selectively combining source code, contained in multiple files, libraries, and data files. Some form of bureaucracy is needed to keep track of the components required to perform a build, along with any dependencies they have on other components, and the tools (plus appropriate options) used to map the input files to the desired final software system. Build systems that have been created include: tools that process rules specifying operations to be performed (e.g., `make` operating on makefiles), and tools that create files containing target specific rules from higher level requirements, e.g., a `configure` script generates the makefiles appropriate to the build system; also see section 5.4.7.

A study by Martin[1210] investigated the features used in 19,689 makefiles. Figure 7.35 shows the number of lines contained in these makefiles, along with the number of dependencies contained in the rules of the respective file. Most of the larger files have been generated by various tools that process higher level specifications, with smaller files being mostly handwritten.

Program source code may be written in a way that supports optional selection of features at build time. One technique for selecting the code to process during compilation is conditional compilation, e.g., `#ifdef/#endif` in C and C++ checks whether an identifier is defining, or not (the identifier is sometimes known as a *feature test macro*, *feature constant*, or *build flag*).

A study by Liebig, Apel, Lengauer, Kästner and Schulze[1138] measured various attributes associated with the use of conditional compilation directives in 40 programs written in C (header file contents were ignored). Figure 7.36 shows the number of unique *feature constants* appearing in programs containing a given number of lines of code.

How extensive is the impact of build flags on source code? A study by Ziegler, Rothberg and Lohmann[2027] investigated the number of source files in the Linux kernel affected by configuration options. Figure 7.37 shows the number of files affected by the cumulative percentage of configuration options. The impact of 37.5% of options is restricted to a single file, and some options have an impact over tens of thousands of files.

Applications may be shipped with new features that are not fully tested, or that may sometimes have undesirable side effects. User accessible command line (or configuration file) options may be used to switch features on/off. A study by Rahman, Shihab and Rigby[1556] investigated the feature toggles supported by Google Chrome. The code supporting a given feature may be scattered over multiple files and provides an insight into the organization of program source. Figure 7.38 shows a density plot of the number of files involved in each feature of Google Chrome; the number of feature toggles grew from 6 to 34 over these four releases.

As source code evolves the functionality provided by a package or library may cease to be used, removing the dependency on this package or library. A missing dependency is likely to be flagged at build time, but unnecessary dependencies are silently accepted. The result is that over time the number of unnecessary, or redundant, dependencies grows.[1739]

One study[2012] of C/C++ systems found that between 83% and 97% recompilations, specified in makefiles, were unnecessary.

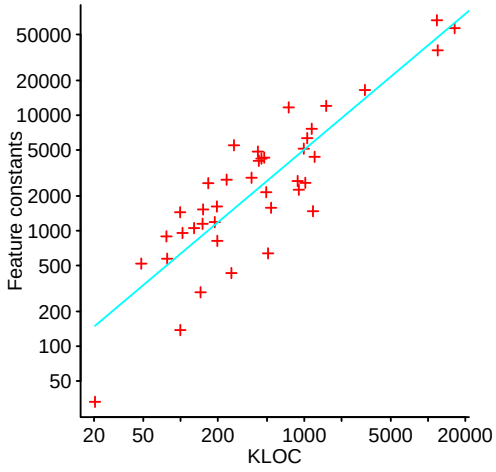


Figure 7.36: Number of feature constants against LOC for 40 C programs; fitted regression line has the form: $Feature\_constants \propto KLOC^{0.9}$. Data from Liebig et al.[1138] Github–Local

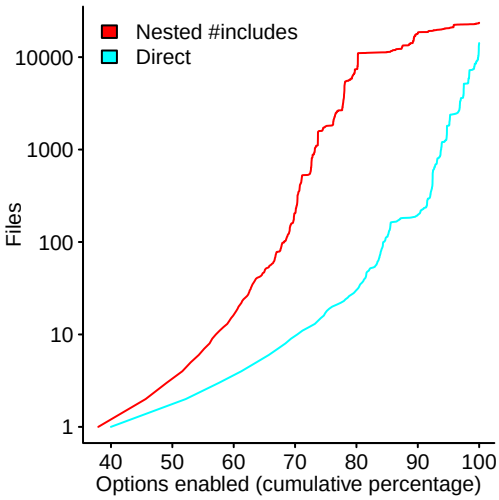


Figure 7.37: Cumulative percentage of configuration options impacting a given number of source files in the Linux kernel. Data kindly provided by Ziegler.[2027] Github–Local

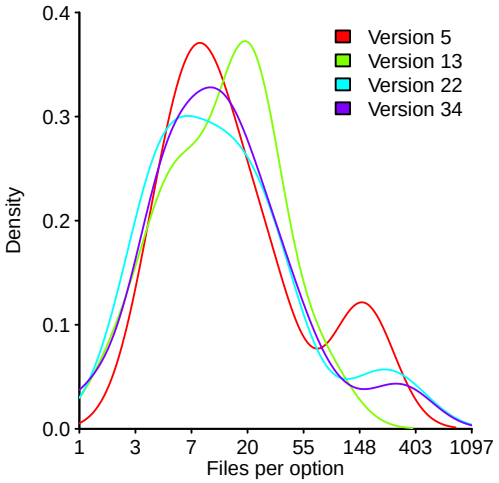


Figure 7.38: Density plot of the number of files containing code involved in supporting distinct options in four versions of Google Chrome. Data from Rahman et al.[1556] Github–Local

7.3 Patterns of use

Patterns of source code use are of general interest to the extent they provide information that aids understanding of the software development process. There are special interest groups interested in common usage patterns, such as compiler writers wanting to maximise their investment in code optimizations by focusing on commonly occurring constructs, by static analysis tools focusing on the mistakes commonly made by developers, and by teachers looking to minimise what students have to know to be able to handle most situations (and common novice mistakes[923]); these groups are a source code data.

Patterns that occur during program execution[1580] can be used to help tune the performance of both the measured program and any associated runtime system; researchers sometimes focus on individual language constructs.[1579]

The spoken form of human languages have common patterns of word usage[1106] and phrase usage,[192] the written form of languages also have common patterns of letter usage.[942] Common usage patterns are also present in the use of mathematical equations.[1732]

A theory has no practical use unless it can be used to make predictions that can be verified (or not), and any theory of developer coding behavior needs to make predictions about common patterns that appear in human written code. For instance, given the hypothesis that developers are more likely to create a separate function for heavily nested code, then the probability of encountering an `if-statement` should decrease with increasing nesting depth.[xi]

Common usage patterns in human written source code are driven by developer habits (perhaps carried over from natural language usage, or training material), recurring patterns of

[xi] Which does not occur in practice, at least in C code; see figure 7.39.

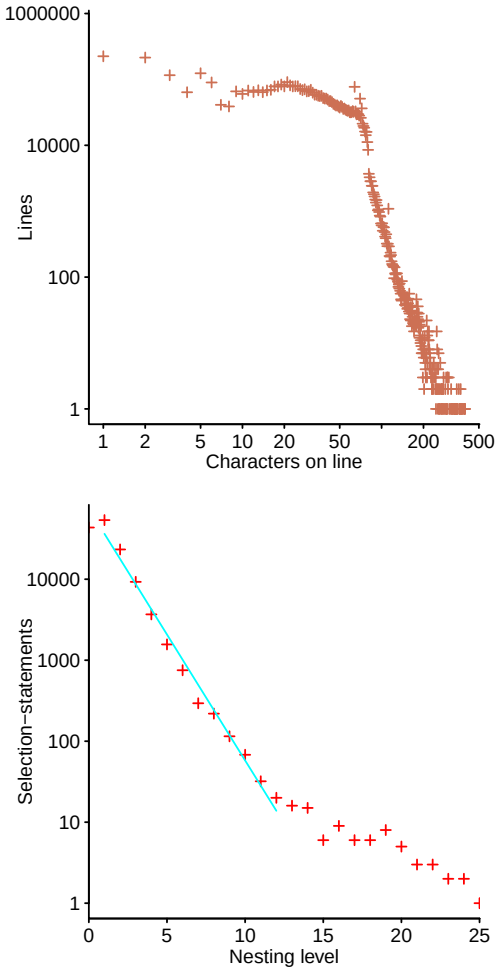


Figure 7.39: Number of `selection-statements` having a given maximum nesting level; fitted regression line has the form: $num\_selection \propto e^{-0.7 nesting}$. Data from Jones.[930] Github–Local

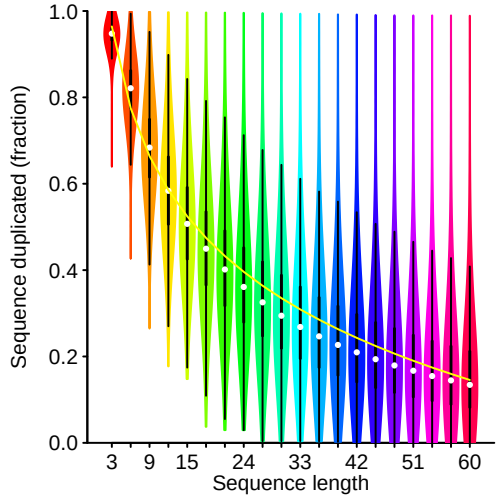


Figure 7.40: Fraction of a project's token sequences, containing a given number of tokens, that appear more than once in the projects' Java source (for 2,637 projects); the yellow line has the form: $fraction \propto a - b * \log(seq\_len)$, where a and b are fitted constants. Data from Lin et al.[1145] Github–Local

behavior in the application domain, hardware characteristics, the need to interface to code written by others, or influenced by the characteristics of the devices used to write code. For instance, the width of computer screens limits the number of characters visible on a line within a window. Figure 7.39, upper plot, shows the number of lines, in C source files, containing a given number of characters. The sharp decline in number of lines occurs around a characters-per-line values supported by traditional character based terminals.[xii]

Some code may be automatically generated. Figure 7.39, lower plot, shows the number of C `selection-statements`[xiii] occurring at a given maximum nesting depth. One interpretation of the decreasing trend, at around a nesting level of 13, is that automatically generated code becomes more common at this depth, than human written code.

Common patterns may exist because a lot of code is built from sequences of simple building blocks (e.g., assigning one variable to another), and there are a limited number of such sequences (particularly if the uniqueness of identifiers is ignored).

A study by Lin, Ponzanelli, Mocci, Bavota and Lanza[1145] investigated the extent to which the same sequence of tokens (as specified by the Java language, with identifiers having different names treated as distinct tokens) occurred more than once in the source of a project. Figure 7.40 shows violin plots of the fraction of a project's token sequences of a given length (for sequence lengths between 3 and 60 tokens) that appeared more than once in the projects Java source (for each of 2,637 projects).

A study by Baudry, Allier and Monperrus[146] investigated the possibility of generating slightly modified programs (e.g., add, replace and delete a statement) having the same behavior (i.e., passing the original program's test suite; the nine large Java programs used had an average statement coverage of 85%). On average, 16% of modified programs produced by the best performing generated add-statement algorithm passed the test suite; 9% for best performing replace and delete; see Github–sourcecode/Synth-Diverse-Programs.csv.xz.

Individual developers have personal patterns of coding usage.[292] These coding accents are derived from influences such as developer experience with using other languages, ideas picked up from coding techniques appearing in books, course notes, etc.

There may be common patterns specific to application domains, programming language or large organizations. The few existing studies have involved specific languages in broad domains (e.g., desktop computer applications written in C[930] and Java usage in open source repositories[734]), and without more studies it is not possible to separate out the influences driving the patterns found.

A variety of development practices can introduce bias into source code measurements, including:

- when making use of source written by third-parties, it may be more cost effective to maintain a local copy of the source files, than link to the original. A consequence of this behavior is the presence of duplicate source files in public repositories (skewing population measurements), and individual project measurements may be unduly influenced by the behavior of developers working on other projects.

  A study by Lopes, Maj, Martins, Saini, Yang, Zitny, Sajnani and Vitek[1157] investigated duplicate code in 4.5 million non-forked Github hosted projects, i.e., known forks of a project were not included in the analysis. Of the 428+ millions source files written in Java, C++, Python or Javascript, 85 million were unique. Table 7.6 shows the percentage of unique files by language, and percentage of projects containing at least a given percentage of duplicates files.

  Figure 7.41 shows the number of Python files containing a given number of SLOC, for a 10% sample of all 31,602,780 files, and the 9,157,622 unique files,

- the decision about which code matches a particular pattern may not be a simple yes/no, but involve a range, e.g., the number of tokens needing to match before a code sequence is considered to be a clone. Tools used to extract patterns from source code often provide options for controlling their behavior,[1554]

- when modifying source, some developers commit every change they make to the project-wide repository, while other developers only make project-wide commits of code they have tested (i.e., they commit changes of behavior, not changes of code).[1362]

[xii] Historically, typewriters supported around 70 characters per line, depending on paper width, and punched cards supported 80 characters.

[xiii] `if-statements` and `switch-statements`.

	Java	C++	Python	JavaScript
Unique files	60%	27%	31%	7%
Projects	1,481,468	364,155	893,197	1,755,618
duplicates > 50%	14%	25%	18%	48%
duplicates > 80%	9%	16%	11%	31%
duplicates 100%	6%	7%	6%	15%

Table 7.6: Percentage of unique files for a given language, number of projects, and average percentage of duplicated files in projects for Github hosted projects written in various languages. Data from Lopes et al.[1157]

When every change is committed, there will be more undoing of previous changes, than when commits are only made after code has been tested.

A study by Kamiya[967] investigated how many deleted lines of code were added back to the source in a later revision, in a FreeBSD repository of 190,000 revisions of C source. Figure 7.42, upper: shows the number of reintroduced line sequences having a given difference in revision number between deletion and reintroduction, and lower: number of reintroductions of line sequences containing a given number of lines, with fitted power laws.

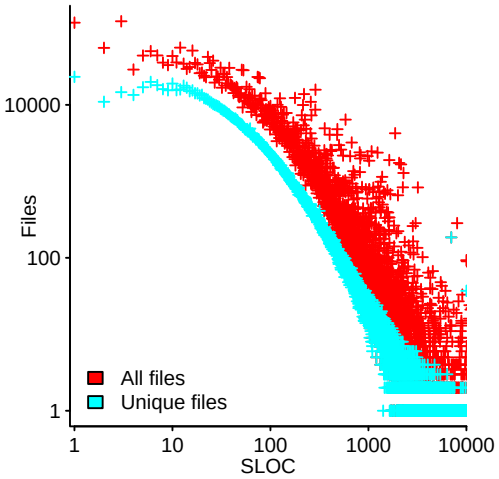


Figure 7.41: Number of Python source files containing a given number of SLOC; all files, and with duplicates removed. Data from Lopes et al.[1157] Github–Local

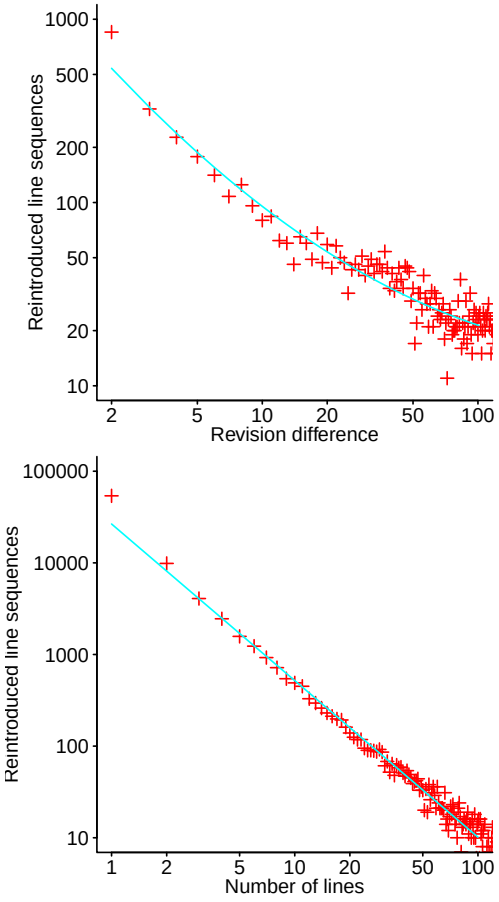


Figure 7.42: Number of reintroduced line sequences having a given difference in revision number between deletion and reintroduction (upper), and number of reintroduced line sequences containing a given number of lines (lower); the fitted regression lines have the form: $Occurrence \propto NumLines^{-1.4} e^{0.1 \log(NumLines)^2}$ and $Occurrences \propto NumLines^{-1.7}$. Data kindly provided by Kamiya.[967] Github–Local

7.3.1 Language characteristics

The idea that the language we use influences our thinking is known as the *Sapir-Whorf* or *Whorfian* hypothesis.[667] The *strong language-based* view is that the language used influences its speakers' conceptualization process; the so-called *weak language-based* view is that linguistic influences occur in some cases, such as the following:

- *language-as-strategy*: language affects speakers performance by constraining what can be said succinctly with the set of available words; a speed/accuracy trade-off, approximating what needs to be communicated in a brief sentence rather than using a longer sentence to be more accurate,[877]
- *thinking-for-speaking*: for instance, English uses count nouns, which need to modified to account for the number of items, which requires speakers to pay attention to whether one item, or more than one item, is being discussed; Japanese nouns make use of classifiers, e.g., shape classifiers such as hon (long thin things) and mai (flat things), and measuring classifiers such as yama (a heap of) and hako (a box of). Some languages assign a gender to object names, e.g., the Sun is feminine in German, masculine in Spanish and neuter in Russian.

  thinking for coding occurs when creating names for identifiers, where plurals may sometimes be used (e.g., the `rowsum` and `rowSums` functions in R),
- languages vary in the way they articulate numbers containing more than one digit, e.g., the components contained in 24 might be ordered as 20+4 (English) or 4+20 (French). Linguistic influences on numerical cognition have been studied.[270]

While different languages make use of different ways of describing the world, common cross-language usage patterns can be found. A study by Berlin and Kay[183] isolated what they called the *basic color terms* of 98 languages. They found that the number and kind of these terms followed a consistent pattern, see figure 7.43; while the boundaries between color terms varied, the visual appearance of the basic color terms was very similar across languages. Simulations of the evolution of color terms[134] suggest that it takes time for the speakers of a language to reach consensus on the naming of colors, and over time languages accumulate more color terms.

Languages vary in their complexity, i.e., there is no mechanism that ensures all languages are equally complex.[1251]

Many programming languages in common use are still evolving, i.e., the semantics of some existing constructs are changing and support for new constructs is being added. Changing the semantics of language existing constructs involves a trade-off between alienating the developers and companies currently using the language (by failing to continue to process existing code), and fully integrating new constructs into the language.

At the end of 2008 the Python Software Foundation released Python 3, a new version of the language that was not compatible with Python 2. Over time features only available in Python 3 have been back-ported to Python 2. How have Python developers responded to the availability of two similar, but incompatible languages?

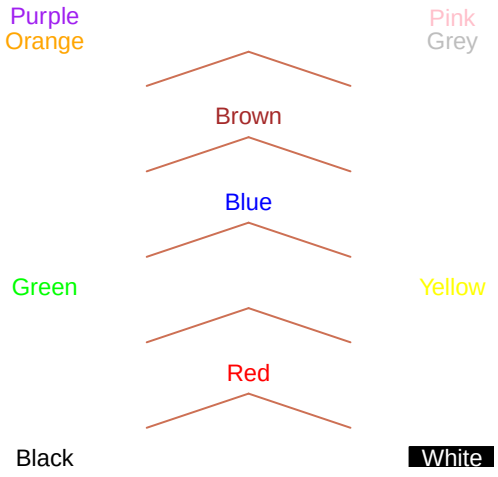


Figure 7.43: The Berlin and Kay[183] language color hierarchy. The presence of any color term in a language implies the existence, in that language, of all terms below it. Papuan Dani has two terms (black and white), while Russian has eleven (Russian may also be an exception in that it has two terms for blue.) Github–Local

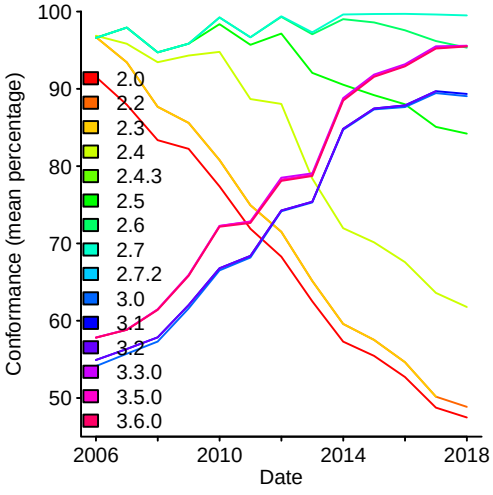


Figure 7.44: Mean compatibility of 50 applications to 11 versions of Python, over time. Data from Malloy et al.[1196] Github–Local

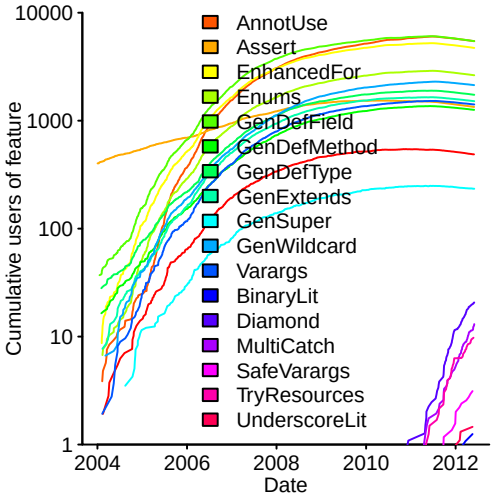


Figure 7.45: Cumulative number of developers who have committed Java source making use of particular new feature added to the language. Data from Dyer et al.[524] Github–Local

A study by Malloy and Power[1196] investigated the extent to which 50 large Python applications were compatible with various releases of the Python language. Figure 7.44 shows changes in the estimated mean compatibility of the 50 applications to 11 versions of Python, over time.

While new features are often added to languages, it is rare for a feature to be removed (at least without giving many years notice). The ISO Standards for both the Fortran and C have designated some constructed as deprecated, i.e., to be removed in the next release, but in practice they are rarely removed.[xiv]

Whatever the reason for the additions, removals, or modifications to a language, such changes will influence the characteristics of some of the code written by some developers. A new construct may add new functionality (e.g., atomics in C and C++), displace use of an existing construct, e.g., lambda expressions replacing anonymous classes.[1224]

How quickly are new languages constructs adopted, and regularly used by developers? Use of new language constructs depends on:

- compiler support. While vendors are quick to claim compliance with the latest standard, independent evidence is rarely available (e.g., compiler validation by an accredited test lab),[xv]
- existing developer knowledge and practices. What incentives do existing users of a language have to invest in learning new language features, and to then spend time updating existing habits? Is new language feature usage primarily driven by developers new to the language, i.e., learned about the new feature as a by-product of learning the language?.

A handful of compilers now dominate the market for many widely used languages. The availability of good enough open source compilers has led to nearly all independent compiler vendors exiting the market.

For extensive compiler driven language usage to exist, widely used diverse compilers are required (something that was once common for some languages). With the small number of distinct compilers now in widespread use, any diversity of language construct use is likely to be driven by the evolution of compiler support for new language constructs, and the extent to which source has been updated to adapt to modified or new features.

A study by Dyer, Rajan, Nguyen and Nguyen[524] investigated the use of newly introduced Java language features, based on the source of commits made to projects on SourceForge. Figure 7.45 shows the cumulative growth (adjusted for the growth of registered SourceForge users[1996]) in the number of developers who had checked in a file containing a use of a given new feature, for the first time. Possible reasons for the almost complete halt in the growth of developers using a new Java language construct for the first time include: the emptying of the pool of developers willing to learn and experiment with new language features, and developers switching to other software hosting sites, e.g., Github became available in 2008.

A unit of source code may contain multiple languages, e.g., SQL,[56] assembler,[1586] or C preprocessor directives (also used in Fortran source to support conditional compilation), or database schema contained within string literals of the glue language used (such as PHP or Python[1147]).

7.3.2 Runtime characteristics

The runtime characteristics of interest to users of software, and hence of interest to developers, are reliability and performance. Reliability is discussed in chapter 6, and issues around the measurement of performance are discussed in chapter 13.

When execution time needs to be minimised, the relative performance of semantically equivalent coding constructs are of interest. A study by Flater and Guthrie[614] investigated

[xiv] ANSI X3.9-1978,[65] known as Fortran 77, listed 24 constructs which it called conflicts with ANSI X3.9-1966. Some of these conflicts were removal of existing features (e.g., Hollerith constants), while others were interpretations of ambiguous wording. Subsequent versions of the Standard have not removed any language constructs.

[xv] When the British Standards Institute first offered C compiler validation, in 1991, three small companies new to the C compiler market paid for this service; all for marketing reasons. Zortech once claimed their C compiler was 100% Standard compliant (it was not illegal to claim compliance to a yet to be published standard still under development), and when the C Standard was published their adverts switched to claiming 99% compliance, i.e., a meaningless claim.

the time taken to assign a value to an array element in C and C++ , using various language constructs. A fitted regression model contains interactions between almost every characteristic measured; see Github–benchmark/bnds_chk.R.

The interaction between algorithm used, size of data structures and hardware characteristics can have a large impact on performance, see fig 13.21.

Patterns of behavior that frequently occur during the execution of particular sequences of source code are of great interest to some specialists, and include (fig 7.6 illustrates that the relationship between static and dynamic behavior may have its own patterns).

- a symbiosis between cpu design and existing code; developers interested in efficiency attempt to write code that makes efficient use of cpu functionality, and cpu designers attempt to optimise hardware characteristics for commonly occurring instruction usage[28] and patterns of behavior (e.g., locality of reference[1297] can make it worthwhile caching previously used values, and the high degree of predictability of conditional branches, statically[126] and dynamically,[1298] can make it worthwhile for the cpu to support branch prediction),
- implementers of runtime libraries. Common patterns in the dynamic allocation of storage include: relatively small objects, with program-specific sizes make up most of the requests,[1206] once allocated the storage usually has a short lifetime,[1206] and objects declared using the same type name tend to have similar lifetimes.[1714]

  A study by Suresh, Swamy, Rohou and Seznec[1798] investigated the value of arguments passed to transcendental functions. Figure 7.47 shows the autocorrelation function of the argument values passed to the Bessel function `j0`.

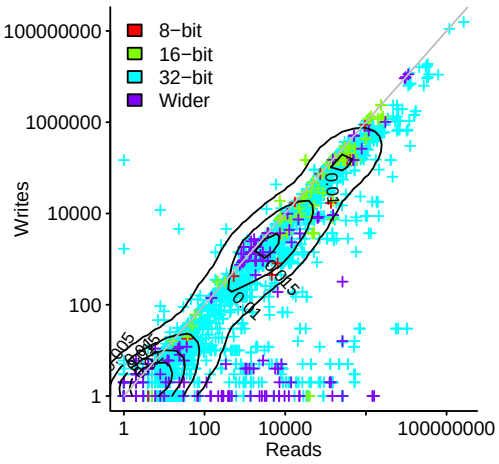


Figure 7.46: Number of reads and writes to the same variable, for 3,315 variables occupying various amounts of storage, made during the execution of the Mediabench suite; grey line shows where number of writes equals number of reads. Data kindly provided by Caspi.[305] Github–Local

7.3.3 Statements

Statements have been the focus of the majority of studies of source code; see fig 9.12 for percentage occurrence of various kinds of statements in C, C++ and Java.

Some languages support the creation of executable code at runtime, e.g., concatenating characters to build a sequence corresponding to an executable statement and then calling a function that interprets the string just as-if it had originally appeared in the source file.

A study by Rodrigues and Terra[1597] investigated the use of dynamic features in 28 Ruby programs. On average 2.6% of the language features appearing in the source were dynamic features; it was thought possible to replace 50% of the dynamic statements with static statements.

Figure 7.48 shows the number of dynamic statements, LOC, and methods appearing in Ruby programs containing a given number of dynamic constructs. Lines are a power law regression fit, with the exponents varying between 0.8 and 0.9.

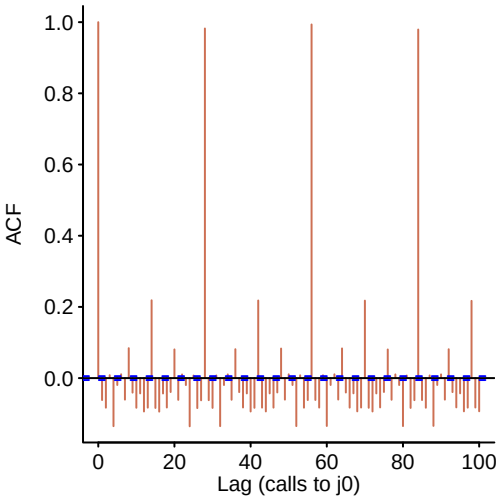


Figure 7.47: Autocorrelation function of the argument values passed to the Bessel function `j0`. Data kindly provided by Suresh.[1798] Github–Local

7.3.4 Control flow

An `if-statement` is a decision point, its use is motivated by either an application requirement or an internal house-keeping issue, e.g., an algorithmic requirement, or checking an error condition.

Developers often using indentation to visually delimit constructs contained within particular control flows; see fig 11.64.

The ordering of `if-statements` may be driven by developer beliefs about the most efficient order to perform the tests, as the code evolves adding new tests last, or other reasons. One study[1987] used profile information to reorder `if-statements`, by frequency of their conditional expression evaluating to true; the average performance improvement, on a variety of Unix tools, was 4%.

The control flow supported by many early programming languages closely mimicked the support provided by machine code.

The `goto-statement` was the work-horse of program flow control, and the term *spaghetti code* was coined to describe source code using `goto-statements` in a way that required excessive effort to untangle control flows through a program. The sometimes heated debates around the use of the `goto-statement`, from the late 1960s and 1970s,[499,1029] have become embedded in software folklore, and continue to inform discussion, e.g., guidelines recommending against the use of `goto-statement`.[1291]

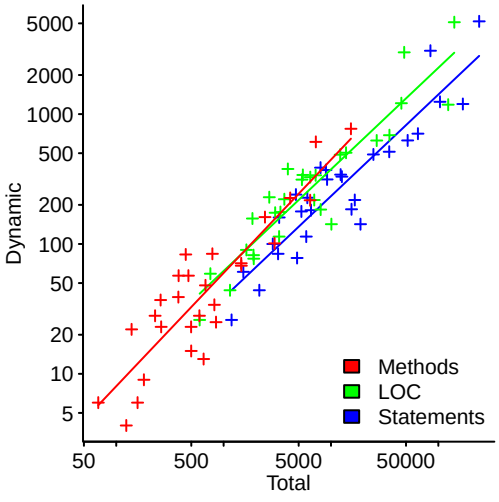


Figure 7.48: The number of dynamic statements, LOC and methods against total number of those constructs appearing in 28 Ruby programs; lines are power law regression fits. Data from Rodrigues et al.[1597] Github–Local

```
if (a != 1)
   goto 100;
b=1;
c=2;
100:;
d=3;
```

```
if (a == 1)
   {
   b=1;
   c=2;
   }
d=3;
```

```
if ((c = *sp++) == 0)
   goto cerror;
if (c == '<') { ... }
if (c == '>') { ... }
if (c == '[') { ... }
if (c == ']') { ... }
if (c >= '1' && c <= '9') { ... }
```

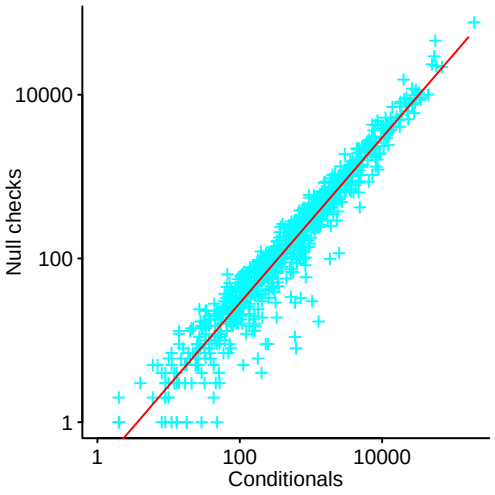


Figure 7.49: Total `if-statements` against `if-statements` whose condition involves a null check, in each of 800 Java projects; regression line fitted has the form: $null\_checks \propto Conditionals$. Data kindly provided by Osman.[1425] Github–Local

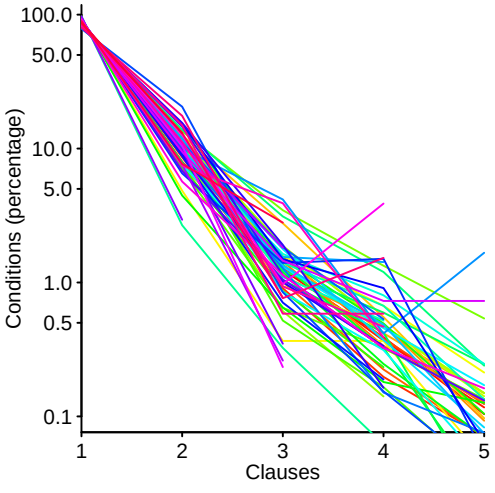


Figure 7.50: Percentage of conditional expressions, in 63 Java programs, containing a given number of clauses; one fitted regression model has the form: $Num\_conditions \propto e^{Num\_predicates \times (\log(SLOC) - 0.6\log(Files) - 11)}$, where each variable is the total for a program's source. Data from Durelli et al.[520] Github–Local

Over time higher level control flow abstractions have been introduced, e.g., *structured programming*; code in the margin shows an example of unstructured and structured code.

A study by Osman, Leuenberger, Lungu and Nierstrasz[1425] investigated the use of checks against the null value in the `if-statements` within 800 Java systems. Figure 7.49 shows the number of `if-statements` against the number of these `if-statements` whose condition checks a variable against null.

Studies of the use of **goto** in Ada[665] and C[930] have found that it is mostly used to jump out of nested constructs, to statements appearing earlier or later; one study[1343] found that 80% of usage in C was related to error handling.

The lower plot in figure 7.39 shows the number of C `selection-statements`, occurring at a given maximum nesting depth. The probability of encountering a `selection-statement` remains relatively constant with nesting depth, implying that developers are not more likely to create a function to contain more deeply nested code, than any other code.

Many languages support a statement to handle the need to select one control flow path from multiple possibilities, based on the value of an expression, e.g., a `switch-statement`. Even when such a statement is available, developers may choose to use a sequence of `if-statements`. For instance, ordering the sequence to reflect the expected likelihood of the condition being true (in the belief that this improves performance); the code in the margin is from the source of a version of `grep`.

A study by Jones[933] investigated developer choice of control flow construct, when the problem allowed either `if-statement` or `switch-statement`, to be used. The questions involved writing a function that used the value of a parameter to select the value to assign to a specific variable; each question specified whether the parameter took 3, 4 or 5 values. The following shows two possible solutions to one question:

```
if (company == 1)
   X = "Intel";
else if (company == 20)
   y = "Motorola";
else if (company == 33)
   W = "IBM";
else if (company == 41)
   p = "Sun";
```

```
switch(company)
   {
   case 1: X = "Intel";
           break;
   case 20: y = "Motorola";
            break;
   case 33: W = "IBM";
            break;
   case 41: p = "Sun";
            break
   }
```

A total of 199 questions were answered by 12 professional developers. One subject used the `if-else-if` form when the parameter contained three values, and a `switch-statement` when more than three values. Two subjects tended to always use the if-else-if form, and nine subjects always used an `switch-statement` (a few subjects answered one question using an `if-statement`).

A study by Durelli, Offutt, Li, Delamaro, Guo, Shi and Ai[520] investigated clauses[xvi] within the conditional expressions contained in 63 Java programs. Figure 7.50 shows the percentage occurrence of conditional expressions containing a given number of clauses; see Github–sourcecode/1-s2.R and Github–sourcecode/sast_2017.R.

Some languages include statements that provide a restricted form of `goto-statement`, e.g., **break** for jumping just past the end of the associated loop; see fig 11.31. Use of this form removes the need for those reading the code to deduce that the purpose of a **goto**) is to exit a loop (and use of these forms do not have the perceived negative connotations associated with the word **goto**).

Some languages include support for a more powerful form of `goto-statement`, originally based on functionality provided by the hardware, e.g., *signal handling* (known as *exception handling* in some languages). The non-local nature of signal handling (it may cause control flow to exit one or more functions in the call tree) can create a lot of need to know.

A study by de Pádua and Shang[456] investigated exception handling in seven C♯ projects (1,502 `try` blocks) and nine Java projects (7,116 `try` blocks). Both C♯ and Java support what are known as `try-catch` blocks; if the execution of code within a `try` block raises

[xvi] A clause is a basic subexpression returning a boolean value, which may be combined with `AND` and `OR` operators to form a more complicated expression.

an exception, it can be caught by the `catch` block (provided the particular exception raised is specified in the list of exceptions handled).

How many exceptions might a `try` block raise? Figure 7.51 shows the number of `try` blocks whose code is capable of raising a given number of exceptions, along with lines showing fitted regression models. Possible reasons for the difference in fitted regression models (i.e., exponential vs. bi-exponential) include: different language characteristics affecting which runtime behaviors are capable of generating an exception, and a consequence of the relatively small number of projects sampled.

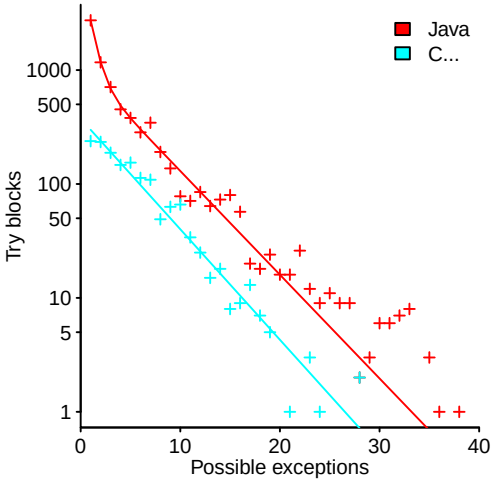


Figure 7.51: Number of `try` blocks whose code might raise a given number of exceptions; fitted regression models have the form: (lower) $Num\_tryBlocks \propto Possible\_exceptions^{-0.22}$ and (upper) $Num\_tryBlocks \propto 7300e^{-1.4 Possible\_exceptions} + 1100e^{-0.21 Possible\_exceptions}$. Data from de Pádua et al.[456] Github–Local

7.3.5 Loops

Loop statements have traditionally been of interest because programs often spend most of their time executing within a few loops (the characteristics of code within loops is intensively studied by compiler writers, optimizing code that commonly occurs in loops is likely to return the greatest ROI for new optimizations).

Compilers attempt to figure out the characteristics of code within loops, such as dependencies between variables in successive iterations, to detect code optimizations[34] that improve the efficiency of the generated code. Calculating worst case program execution time (WCET) requires accurate estimates of the number of iterations, along with the execution time of a single iteration.[1094]

Loops might be classified based on difficulty of automated analysis.[1444]

7.3.6 Expressions

What do developers need to know about the semantics of expression evaluation?

Many languages may perform implicit type conversions on one or more of the operands in an expression, e.g., casting one operand of a binary operator so that both operands have the same type. For instance, many languages consider the operands in the expression `1+1.0` to be some integer type and some floating-point type, respectively, and in languages with C-like implicit conversion rules the behavior is as-if `(double)1+1.0` had been written.

The implicit conversions that might be performed vary between languages. For instance, the expression `1+"1"` may return the result 2 (e.g., PHP and Lua), or `"11"` (e.g., Javascript), or generate a compile-time error (e.g., many languages), or perhaps something else.

Implicit conversions remove the developer effort needed to write an explicit conversion (and the effort involved in processing its visual form, if the code is later read), but creates a need to know about the implicit conversions specified by the language.

The original need for operands to be converted to a common type, before being operated on, was driven by the behavior of the underlying hardware instructions; the concept of same type was synonymous with same underlying data representation. Some languages have moved away from the concept of types being solely dependent on the underlying representation, and provide a means for developers to specify new type compatibility relationships.

The purpose of developer-defined type constraints is to detect coding mistakes, and their ability to catch mistakes is dependent on the extent to which developers make use of the available functionality. Some languages were explicitly designed to support developer-defined type constraints (e.g., Ada; see margin code), while other language support such functionality through the use of constructs designed for more general uses, e.g., C++ .[214]

```
type
     celsius is new real;
     fahrenheit is new real;
var
     L_temp  :celsius;
     NY_temp :fahrenheit;
...
 L_temp:=NY_temp; - types not compatible
```

When the functionality is available, the extent to which developers make use of user defined type constraints appears to be cultural. For instance, while both Ada and C++ provide mechanisms offering the same level of support for user defined type constraints, there is a culture of developer-defined type constraint use in the Ada community, but not in the C++ community.

The following studies have experimental investigated differences in developer performance when using languages the researchers claim differ in support for strong typing:

- Gannon:[650] used two simple languages, which by today's standards were weakly typed, with one less so than the other (think BCPL and BCPL plus a string type and simple structures). A single problem was solved by subjects, which had been designed to require the use of features available in both languages, e.g., a string oriented problem

(final programs were between 50-300 lines). The result data included number of errors during development and number of runs needed to create a working program (this happened in 1977, before the era of personal computers, when batch processing was common; see Github–experiment/Gan77.R).

There was a small language difference in number of errors/batch submissions; the difference was about half the size of the effect of experimental order of language used by subjects, both of which were small in comparison to the variation due to subject performance differences. While the language effect was small, it was present. It is not possible to separate out performance differences due to stronger typing, rather than built in support for a string type only being available in one language.

- Mayer, Kleinschmager and Hanenberg:[1023,1223] Two experiments using different languages (Java and Groovy) and multiple problems; the performance metric was time to complete the task. There was no significant difference due to just language, but large differences due to language/problem interaction, with some problems solved more quickly in Java and others more quickly in Groovy, and learning took place, i.e., the second task was completed in less time than the first. As often occurs, there were large variations in performance between subjects; see Github–experiment/mayerA1-oopsla2012.R and Github–experiment/kleinschmagerA1.R.
- Hoppe and Hanenberg:[853] one language (Java) was used, and multiple problems; the problems involved making use of either Java's generic types or non-generic types. Again, the only significant language difference effects occurred through interaction with other variables in the experiment (e.g., the problem or the language ordering), and there were large variations in subject performance.

To summarise: when a language typing/feature effect has been found, its contribution to overall developer performance has been small. Possible reasons for the small or non-existent effect, include: [xvii] the use of subjects with little programming experience (i.e., students; experienced developers are more likely to make full use of the consistency checking provided by a type system), and the small size of the programs (type checking comes into its own when used to organize, and control, large amounts of code).

Many languages contain more than twenty different kinds of operators which can appear in expressions (supporting the wide variety of different kinds of operations that have been created to combine values). By specifying operator precedence for the relative binding strength of operators (commonly used languages have 10 to 15 precedence levels), to their operands, languages remove the need for developers to explicitly specify the intended binding of operands to operators (by using parenthesis). Expressions that do not use parenthesis create a developer need to know for operator precedence.

One study[931] found that the likelihood of developers knowing the correct relative precedence of two binary operators increased with frequency of occurrence of the respective pair of operators in existing C source; see fig 2.38.

Section 2.3.1 discusses studies investigating the processes involved in reading expressions.

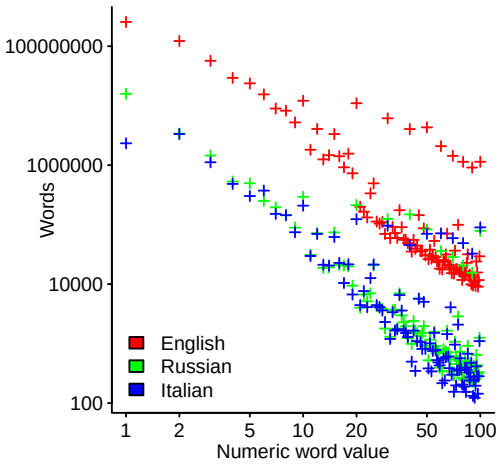


Figure 7.52: Yearly occurrence of number words (e.g., "one", "twenty-two"), averaged over each year since 1960, in Google's book data for three languages. Data kindly provided by Piantadosi.[1484] Github–Local

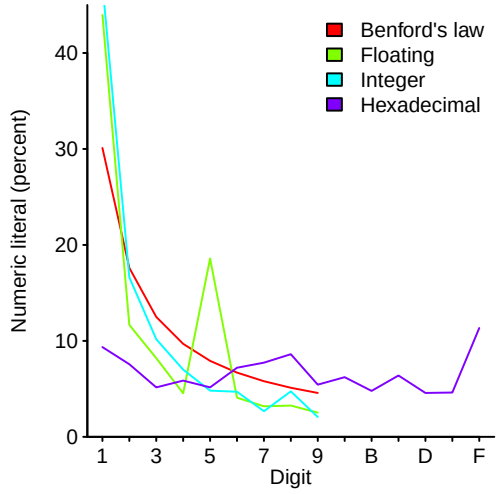


Figure 7.53: Percentage occurrence of the most significant digit of floating-point, integer and hexadecimal literals in C source code. Data from Jones.[930] Github–Local

7.3.6.1 Literal values

Literal values appear in source for a variety of reasons, including: specific value required by an algorithm,[74] size or number of elements in the definition of an array,[930] implementation specific values (e.g., urls, dates and developer credentials[2025]), application domain values, personal preferences of developers (see section 2.7.1), and a representation of no-value, i.e., a null value.

The distribution of numeric values in application domains will have been influenced by real-world usage. Figure 7.52 shows the yearly occurrence of number words (averaged over each year since 1960) in Google's book data. The English counts are larger because most of the books processed were written in English. Decade values (e.g., ten, twenty) follow their own trend, and these are much more common than adjacent values.

Some languages support multiple ways of representing numeric literals, e.g., decimal, binary, hexadecimal. Figure 13.5 suggests that the distribution of the value of numeric literals depends on the representation used. Figure 7.53 shows that Benford's law is a very crude approximation for decimal integer and floating-point numeric literal usage in source code.

[xvii] Your author declares his belief that when integrated into the design process, strong typing has cost/benefit advantages.

The use of the value zero during the execution of many kinds of program is sufficiently common that many RISC processors hard-code one register to contain zero; the use of the whitespace character is very common in Cobol applications.[xviii]

7.3.6.2 Use of variables

What are the patterns of use of variables in source code?

Section 8.3.1 discusses models that relate frequency of local variable use and number of variable declarations, within in a function.

A study by Sajaniemi and Prieto[1629] investigated the roles of variables in source code. They were able to categorise variable use into one of approximately 10 roles, which included: *stepper* which systematically takes predictable successive values, *follower* which obtains its new value from the old value of another variable, and *temporary* which holds some value for a short time.

Variable use may be driven by the constructs supported by the language. For instance, in C, the loop header often contains three appearances of the loop control variable, e.g., `for (i=0;i<10;i++)`; in languages that support constructs of the form `for (i in v_list)`, only one appearance is required. In languages that support vector operations, an explicit loop may not be needed to perform some operations on variables, e.g., in R two vectors can be added together using the binary plus operator.

An analysis[381] of integer use in C found that around 20% of accesses to variables, having an integer type, were made in a context having a signedness that was different from the declared type; also the declarations of variables having an integer type were not usually modified.

How often are variables read and written by functions? One study[930] measured C source, with variables local to the function, source file or externally visible. Figure 7.54 shows: upper the number of functions containing a given number of references to the same variable (the same function may be counted more than once), and lower: the number of functions containing a given number of references to all variables; solid lines are reads, dashed lines are writes. Most functions reference a few variables, which is consistent with most functions containing a few lines; see fig 7.16.

A study by Gonzaga[702] investigated the use of global variables and parameters in the functions defined in 40 C programs. Comparing the number of function parameters, functions that did not access global variables had 0.4 more parameters (on average). For 30 programs the larger number of parameters, for functions accessing/not accessing global variables, was statistically significant; see Github–sourcecode/Gonzaga.R, and fig 7.30.

Figure 7.55 shows the number of functions defined to have a given number of parameters; solid lines are functions that did not access global variables, dashed lines are functions that accessed global variables.

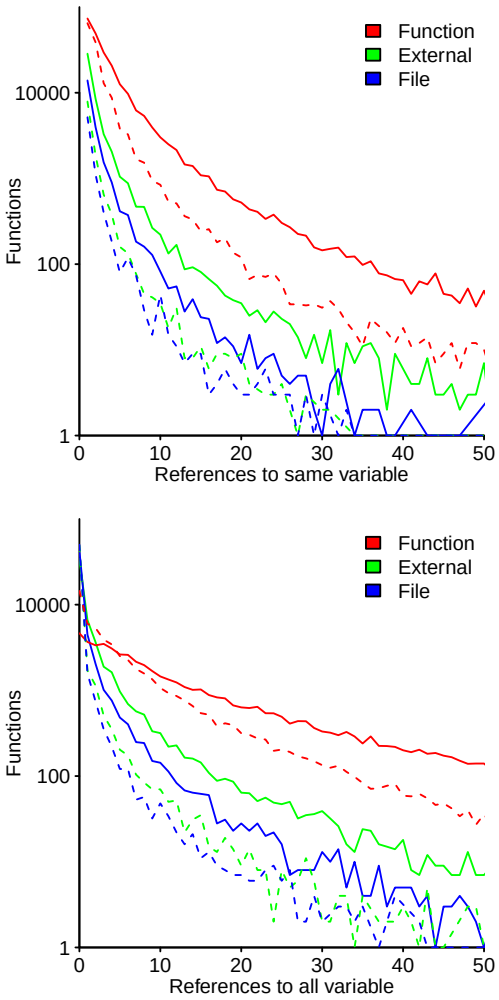


Figure 7.54: Number of C functions contains a given number of references to the same variable (upper), and a given number of references to all variables (lower); reads are full lines, writes dashed lines, colors indicate variable's visibility. Data from Jones.[930] Github–Local

7.3.6.3 Calls

Roughly 1-in-5 statements contains an explicit function/method call (compilers sometimes introduce additional calls to implement language constructs; see fig 7.6).

Some call sequences are part of a common narrative, e.g., open a file, write to it and close it. Detecting narrative sequences can be straightforward in code written in object-oriented language, because the variable name associated with an object is included in calls to methods associated with that object, e.g., `var.strLength()`.

A study by Mendez, Baudry and Monperrus[1262] investigated method call sequences associated with the same variable, based on an analysis of 4,888 classes in 3,418 Jar files (i.e., Java bytecode; some method calls may not explicitly appear in the original source, e.g., the compiler maps string concatenation using binary + to a call to the method `StringBuilder.append`). Figure 7.56 shows the 10 most frequent sequences of `java.lang.StringBuilder` methods called on the same variable (lines connect methods called in sequence; method call argument types are ignored).

Figure 7.57 shows the number of sequences having a given length (i.e., measured in methods; in blue), and number of sequences that appear in the code (i.e., are used; in red) a

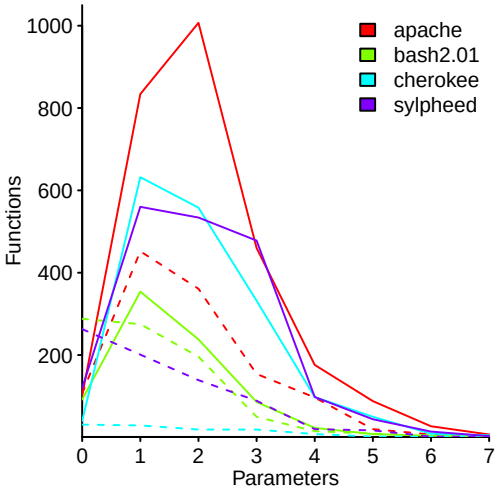


Figure 7.55: Number of functions defined with a given number of parameters in the C source of four projects; solid lines function body did not access global variables, dashed lines function body accessed global variables. Data from Gonzaga.[702] Github–Local

[xviii] The MicroFocus Cobol code generator for the SPARC processor, designed by your author, dedicated one 32-bit register to always hold the value 0x20202020, i.e., four whitespace characters.

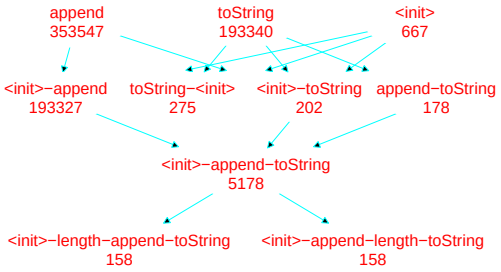


Figure 7.56: Sequences of methods, from `java.lang.StringBuilder`, called on the same object; based on 3,418 Jar files. Data from Mendez et al.[1262] Github–Local

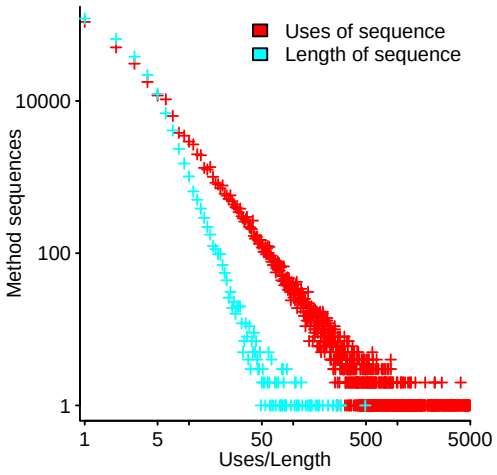


Figure 7.57: For each Java class, in 3,418 jar files, the number of method sequences containing a given number of calls (red), and the number of uses of each sequence (blue). Data from Mendez et al.[1262] Github–Local

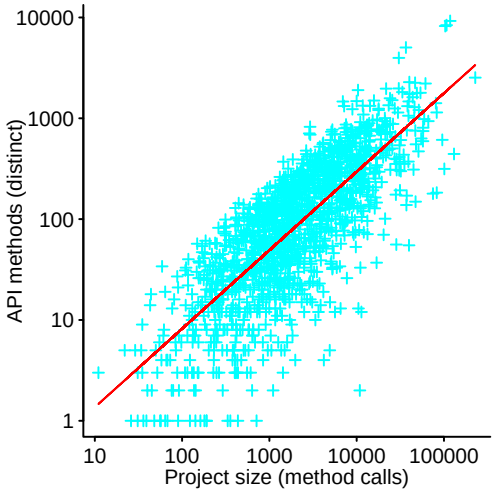


Figure 7.58: Number of distinct API methods called in 1,435 Java projects containing a given number of method calls; the line is a fitted regression model of the form: $unique \propto calls^{0.78}$. Data from Lämmel et al.[1073] Github–Local

given number of times; only classes containing method sequences used at least 100 times are included.

How does the number of calls to distinct methods grow with project size?

A study by Lämmel, Pek and Starek[1073] investigated calls to methods from third-party APIs, and those defined within 1,435 projects. Figure 7.58 shows the number of distinct API methods called against project size (measured in method calls).

The function/method called may be passed as an argument, e,g., a *callback*. A study by Gallaba, Mesbah and Beschastnikh[645] investigated the use of callbacks in 130 Javascript programs. Figure 7.59 shows the total number of calls in each program, against the number of calls containing callbacks, and just anonymous callbacks.

7.3.7 Declarations

Declarations[xix] are code bureaucracy that provides two basic services: a means of introducing a sequence of characters that is to be treated as a valid identifier (sometimes known as a *name*), and specifying operations associated with uses of the identifier in source code (these operations are derived from information appearing in the declaration of the name, e.g., the type of an object).

Large programs may define tens of thousands of identifiers.[930]

Some languages do not require an identifier to be defined in a declaration, before (or after) it appears in the source code. In such languages the context in which the identifier appears is used to derived attributes associated with subsequent uses, e.g., variables have the type of the value last assigned to them. A study[558] of four large PHP applications found that less than 1% of variables were assigned values having different types, e.g., assigning an array and later assigning an integer.

Desirable characteristics of declarations are those that minimise the need to know about the identifiers defined.

Some degree of visibility is one characteristic identifiers acquire during the definition process, i.e., they can be referred to over some region of the source code. Reducing the visibility of identifiers reduces the amount of information developers need to know, when dealing with code where the identifiers are not required to be visible.

Many languages provide mechanisms for restricting the visibility of identifiers, e.g., the `private` in Java and `static` in C; R is an example of a language that provides limited functionality. While studies[1900] have found that identifiers are sometimes declared with greater visibility than necessary (given their existing use in code), there has not been any analysis of the cost/benefit of supporting potential future unintended/intended uses; see Github–sourcecode/TR_DCC-overExposure/TR_DCC-overExposure.R.

To what extent do declarations change over time?

A study by Neamtiu, Foster and Hicks[1360] investigated the release history of three C programs over 3-4 years, and a total of 48 releases. They found that one or more fields were added to one or more existing structure or union types in 79% of releases, while structure or union types had one or more fields deleted in 51% of releases; a later study[1361] found one or more existing fields had their types changed in 35% of releases. Figure 7.60 shows the relationship between the number of global variables and lines of code, in three C programs, over multiple releases.

A study by Robbes, Róthlisberger and Tanter[1590] investigated data extensions (i.e., the visitor pattern) and operation extensions to Smalltalk classes; the 2,505 projects analyzed contained 95,662 classes, forming 48,595 class hierarchies (with 41% containing more than one class). Figure 7.61 shows the number of data and operation extensions made to 1,560 class hierarchies containing both kinds of extension.

One study[1353] of eight Java systems found that the number of methods and classes having a given inheritance depth decreased by a factor of 0.25 per inheritance level; see Github–sourcecode/JavaInherit.R

[xix] Some languages use the term *definition*, and some use both, e.g., in C a definition is a declaration that causes storage to be allocated for an object.

7.3.8 Unused identifiers

Some identifiers are defined, and never referenced again, i.e., they are unused. Reasons for the lack of use include: a mistake has been made (e.g., the variable should have been referenced), the identifier was once referenced (i.e., the declaration is now redundant), and there is an expectation of future need for the entity that has been defined.

Unused identifiers consume cognitive resources for no benefit.

It is not always cost effective to remove the definition of an unused identifier. For instance, removing an unused function parameter may require a greater investment than is likely to be worthwhile (because changing the number of function parameters requires corresponding changes to the arguments of all calls).

Figure 7.62 shows the total number of functions having a given number of parameters, a given number of unused parameters, and various fitted regression models. Unused function parameters, which at around 11% of all parameters are slightly more common than unused local variables.

The fitted regression model, for the number of functions containing a given number of unused parameters has the form: $functions \propto e^{-0.5unused}$ (in practice, functions are likely to acquire unused parameters one at a time, as the code evolves). An alternative formula for estimating the number of functions containing u unused parameters is: $\sum_{p=u}^{8} \frac{F_p}{7p}$, where: F_p is the total number of function definitions containing p parameters; figure 7.62 shows how well this wet-finger model fits.

7.3.9 Ordering of definitions within aggregate types

Consistent patterns appear in the ordering of member declarations within many Java `class` and C `struct` types; for instance, members sharing an attribute are often sequentially grouped together, or have a preferred relative ordering.

These usage patterns may be the result of many developers making individual choices (either explicitly or implicitly), or because externally specified ordering rules are being followed, e.g., the Java coding conventions (JCC)[1795] specifies a recommended ordering for the declaration of: class variables (or fields), instance variables (or static initializers), constructors and methods; see fig 12.8.

Statistical analysis techniques for items believed to have a preferred order are discussed in section 12.4.

Patterns in the ordering of fields in C `struct` types are discussed in section 9.6.1. One interpretation of the pattern seen, is developers wanting to reduce unused storage by grouping together objects whose types have the same alignment requirements.

A study by Geffen and Maoz[664] investigated various patterns of method ordering within Java classes; for instance, the rules specified by the `StyleCop` tool (e.g., group by access modifiers), the commonly seen pattern of called methods appearing after the method that involved them, and the concept of clustering, i.e., related methods appearing in the same class or file.

A study by Biegel, Beck, Hornig and Diehl[196] investigated the impact of the kind of activity performed by a method on its relative ordering. The method activity attributes considered were: 1) all static methods (declared using the `static` keyword), 2) initializers (method name begins with `init`), 3) getters and setters (non-void return type and method name begins with `get`, `is` or `set` followed by a capital letter) and 4) all other non-static methods.

Figure 7.63 shows that methods performing two of the activities are very likely to be ordered before methods performing the two remaining other activities. Method declaration sequences containing more than two kinds of method activity occurred in 62% of contexts analysed, with 54% of these passing the threshold needed for analysis, leaving 33% of all declarations sequences.

The original author of the code may not be responsible for maintenance, and it is possible that declarations added during maintenance were not inserted into an existing structure/-class declaration according to the ordering pattern used during initial development, e.g., some developers may prefer to add new declarations at the end of an existing structure/-class.

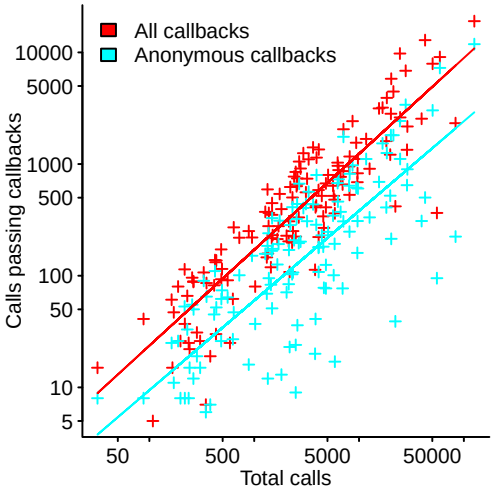


Figure 7.59: Number of function calls, against corresponding number of calls containing callbacks and anonymous callbacks, in 130 Javascript programs; lines are fitted regression models of the form: $allCallbacks \propto allCalls^{0.86}$ and $anonCallbacks \propto allCalls^{0.8}$, respectively. Data from Gallaba et al.[645] Github–Local

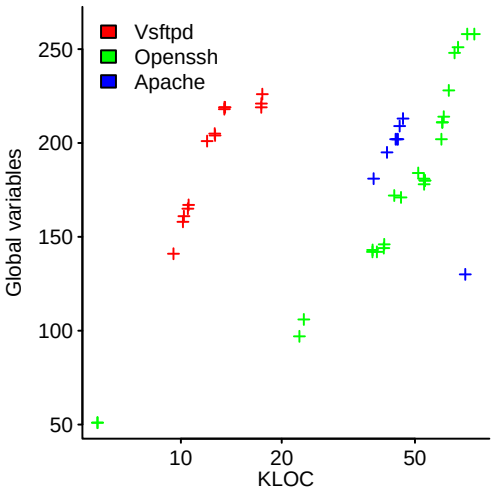


Figure 7.60: Number of global variables against lines of code over 48 releases of three systems written in C. Data kindly provided by Neamtiu.[1360] Github–Local

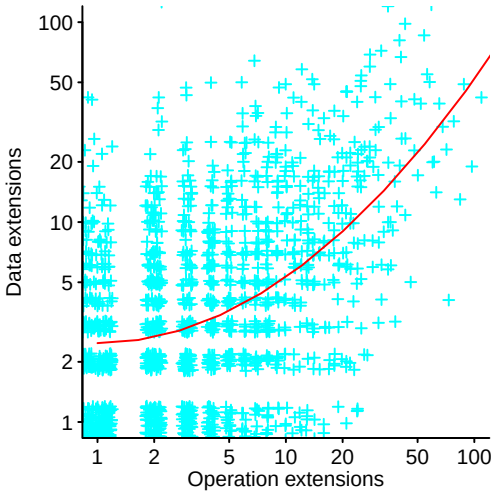


Figure 7.61: Jittered number of data and operation extensions to 1,560 Smalltalk class hierarchies containing both kinds of extension; regression line has the form: $\log(Data\_extensions) \propto \log(Operation\_extensions)^2$. Data from Robbes et al.[1590] Github–Local

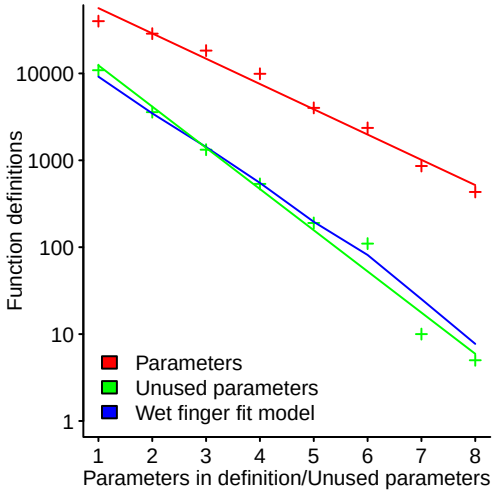


Figure 7.62: Number of C function definitions having a given number of parameters (red) and unused parameters (green); parameter fitted regression line has the form: $functions \propto e^{-0.67 parameters}$. Data from Jones.[930] Github–Local

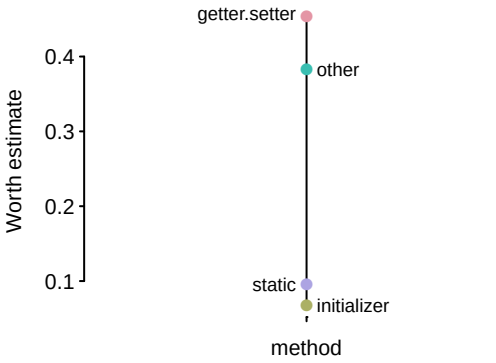


Figure 7.63: "Worth estimate" for the kind of method activity attribute; see section 12.4. Data from Biegel et al.[196] Github–Local

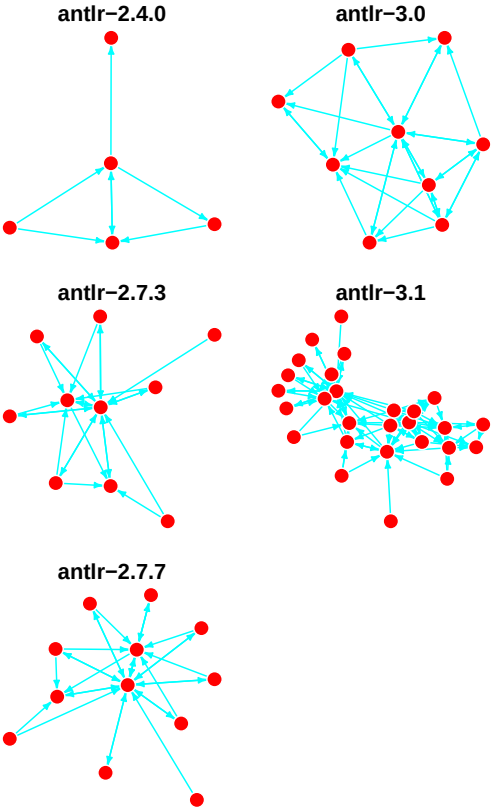


Figure 7.64: Dependencies between the Java packages in various versions of ANTLR. Data from Al-Mutawa.[29] Github–Local

7.4 Evolution of source code

Software only changes when developers have an incentive to spend time making the changes, incentives include: being paid, and a desire to change the code to satisfy a personal need, e.g., refactoring code to maintain a personal self-image, as a developer, because of a belief, for instance, that other developers reading the unrefactored code would form a low opinion of the author.

If payment is involved, there is a customer, and the changes are supposed to address customer needs (it can be very difficult to work out what the customer needs actually are, and there may be as many opinions about these needs as there are people trying to keep the customer happy).

The uncertainty of future customer demand creates uncertainty in the cost/benefit analysis of investment decisions.

Software systems growth, in lines of code, over time is a commonly quoted metric; reasons for growth include improvements to existing functionality and the addition of new functionality. Some systems grow at a consistent rate over many years (e.g., for FreeBSD see fig 11.2, and for the Linux kernel see fig 11.7), while others appear to have stopped adding lines (e.g., the glibc library, see fig 11.52), or grow sporadically, e.g., the Groovy compiler, see fig 11.9.

Growth can increase interdependencies between components; figure 7.64 shows the relationship between the separate components of ANTLR over various releases.

Factors influencing the rate of evolution of source code characteristics include:

- customer limited: insufficient customer demand (as measured by willingness to pay) for it to be economically worthwhile updating existing functionality (to support changes in the world), or adding new functionality, e.g., new hardware requiring device drivers,
- developer limited: bottlenecks in the development process that restrict the quantity of change per unit time. For instance, a limited number of people with the necessary skills, change requests requiring sign-off by a handful of senior managers, or increasing developer resources required to support a growing system leading to diminishing returns from adding more developers,
- competition from other applications: source code may cease to evolve because its host, the application, is out-competed, e.g., customers stop using the application and/or it looses developer mindshare,
- hardware characteristics: there may be benefits to adapting software to the characteristics of the hardware on which it is used.

  In Fortran, **common** blocks provide a means of specifying how different variables are overlaid in memory; for several decades **common** blocks were widely used. As the amount of memory available on computers grew, and compilers became more sophisticated at optimizing memory allocation, the need to use **common** decreased.[xx]

A consistent rate of code growth suggests some degree of consistency in demand for new updates, and developer resources available to do the work; see fig 11.2.

During the evolution of source code some of the contents of units of code (e.g., files or functions) may be moved to other units.[690] Studies of code evolution that do not take code migration into account will overestimate the amount of code added and deleted, over time.

Updating existing functionality may result in source code being deleted.

Figure 7.65 shows the percentage of code in 130 releases of Linux that originated in earlier releases, and fig 4.18 shows code shared between different releases of related BSD operating systems; fig 11.70 shows the correlation between lines added/deleted for glibc, fig 9.21 shows a Markov chain for the creation/modification/deletion of files in the Linux kernel.

[xx] A common Fortran coding mistake was to assign to a variable sharing the same memory location as another variable, and later to access the other variable believing it contained what was earlier assigned to it, i.e., the lifetimes of variables stored in the same memory location overlapped.

7.4.1 Function/method modification

The likelihood of modifying existing code is an essential input to the cost/benefit analysis carried out prior to making any investment intended to reduce the cost of future modifications. The expected lifespan of the system containing the code is a higher level consideration discussed in section 4.2.2.

A new function/method definition is about to be written, and it is believed that at some future time it may need to be modified. If an investment, I, in extra work is made today to receive the benefit, B, for each of the M_t future modifications: like all investments, the expected benefit is required to be greater than the investment, e.g., $I < M_t B$.

Let s be the likelihood that a function is modified in the future, and that once modified the likelihood of it being modified again remains unchanged; the expected number of modifications of a given function is then: $M_t = s + 2s^2 + 3s^3 + \cdots + ns^n$, where: n is the maximum number of modifications of a function; this series sums to:

$$M_t = \frac{s-(n+1)s^{n+1}+ns^{n+2}}{(1-s)^2}$$

substituting and rearranging the cost/benefit equation, and assuming $(n+1)s^{n+1}$ is very small, gives:

$$\frac{(1-s)^2}{s} < \frac{B}{I}$$

What range of values might s have in practice? A study by Robles, Herraiz, German and Izquierdo-Cortázar[1595] analysed the change history of functions in Evolution (114,485 changes to functions over 10 years), and Apache (14,072 changes over 12 years).

Figure 7.66 shows the number of functions (in Evolution) that have been modified a given number of times (upper), and the number of functions modified by a given number of different authors (lower). A bi-exponential model provides a reasonable fit to both sets of data. One interpretation of this bi-exponential model is that many functions are modified by the same developer (or core team members) during initial implementation (see figure 7.68), with fewer functions modified after initial development (with non-core developers more likely to be involved).

The previous analysis assumes s is constant (i.e., the data is fitted by one exponential), but figure 7.66 is fitted using a bi-exponential (which has a non-constant s). The mean half-life of the bi-exponential: $ae^{-\lambda_1 x} + be^{-\lambda_2 x}$, is: $\tau_{mean} = \frac{a\tau_1^2 + b\tau_2^2}{a\tau_1 + b\tau_2}$, where: $\tau_1 = \frac{1}{\lambda_1}$ and $\tau_2 = \frac{1}{\lambda_2}$.

Using τ_{mean} gives, for Evolution: $s = 0.64$, and $0.56 < \frac{B}{I}$. While using the post initial development exponential, gives: $s = 0.85$, and $47 \times 0.025 = 1.2 < \frac{B}{I}$ (the original investment was made in 47 times as many functions/methods as fitted by this exponential); see Github–evolution/author-mod-func.R.

For Apache, the mean τ_{mean} gives: $s = 0.81$, and $0.046 < \frac{B}{I}$, and post initial development is: $s = 0.95$, and $96 \times 0.0032 = 0.3 < \frac{B}{I}$.

This model does not take into account any benefits received if developers read the code without modifying it.

Figure 7.67 shows the number of modifications of a function, stratified by number of authors. The form of the following equation was found by trial and error, it fits the data reasonably well: $\log(num\_authors)^{0.2}(\alpha + \beta num\_mods^{0.3}) + \gamma num\_mods^{0.3}$, where: α, β and γ are fitted constants.

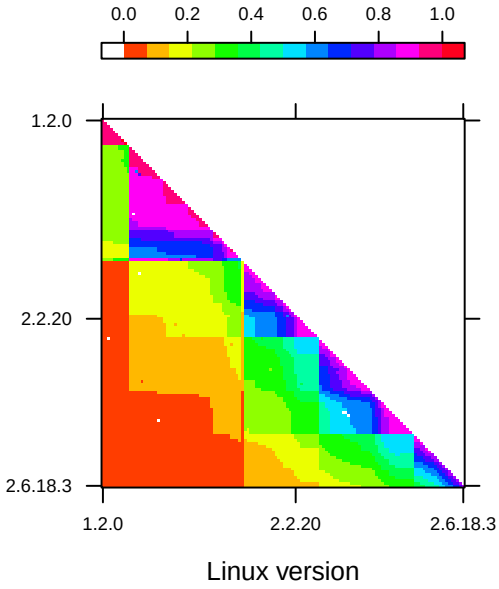


Figure 7.65: Fraction of source in 130 releases of Linux (x-axis) that originates in an earlier release (y-axis). Data extracted from png file kindly supplied by Matsushita.[1155] Github–Local

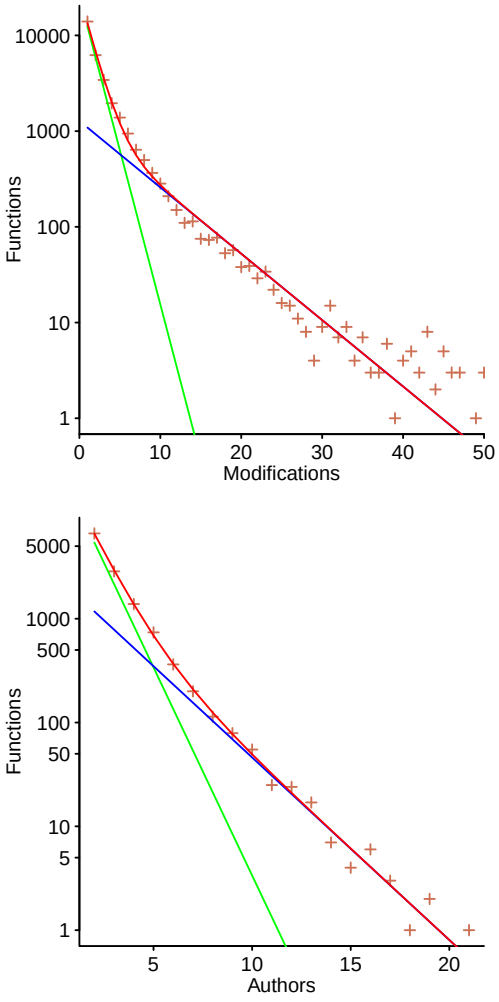


Figure 7.66: Number of functions in Evolution modified a given number of times (upper), and modified by a given number of different people (lower); red line is a fitted bi-exponential, green/blue lines are the individual exponentials. Data from Robles et al.[1595] Github–Local

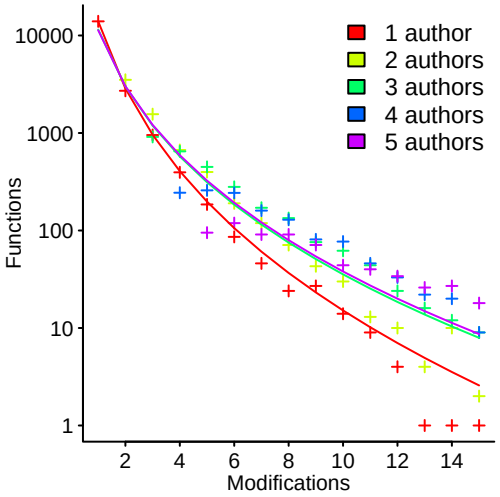


Figure 7.67: Number of functions (in Evolution) modified a given number of times, broken down by number of authors; lines are a fitted regression model. Data from Robles et al.[1595] Github–Local

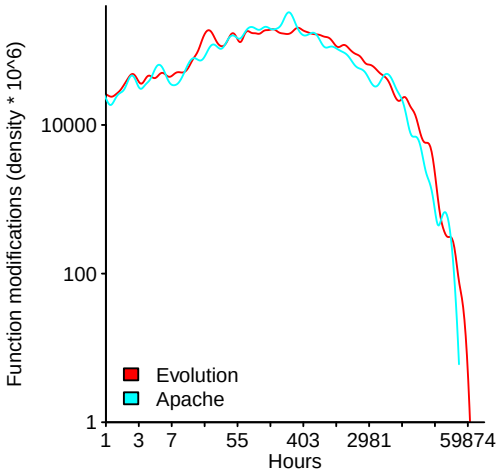


Figure 7.68: Density plot of the time interval, in hours, between each modification of the functions in Evolution and Apache. Data from Robles et al.[1595] Github–Local

Chapter 8

Stories told by data

8.1 Introduction

Data analysis is the process of finding patterns in data and weaving a story around these patterns.

Finding patterns in data is easy, weaving a believable narrative around them can be very difficult. Figure 8.1 may be interpreted as evidence for a causal connection between UFO activity and computer virus infections. Domain knowledge (e.g., personal experience of reporting problems and events) might lead us to believe that these reports were made by people, and an alternative interpretation is that U.S. counties with larger populations experienced and reported more virus infections and UFO sightings, compared to counties having smaller populations.

An understanding of common patterns found in data is the starting point for an appreciation of the kinds of stories that these patterns might be used to substantiate. This chapter starts with an overview of techniques that may be used to uncover patterns in data, before moving on to discussing the communication of these patterns to others. Those performing the analysis are responsible for weaving a story around the patterns found; the figures, and numeric values provide props that may be used to conjure a convincing narrative.

The patterns sought have the form of a relationship between two or more measured quantities. Managers want to control software development, and to do this they need understanding of the processes that are driving it. Regression modeling is this book's default technique for modeling the relationships between the quantities that have been measured; see chapter 11.

Ideally you, the data analyst, have:

- sufficient domain knowledge to be able to distinguish between spurious correlations that may be present in the data, and correlations connected to the processes that generated the data,
- practical ideas relating to the questions for which answers are sought, in practice there may be a lot of uncertainty about what the questions are.

  Questions have to have answers that can be used to make predictions about expected patterns of behavior in the data (which can be searched for).

  If a question does not have an associated answer that has a predictable, detectable, pattern of behavior, then 42 is as good an answer as any other,
- the time and resources needed to obtain data likely to contain answers to the questions asked; obtaining data is often time-consuming and/or expensive and it is often necessary to make do with whatever data is cheaply and quickly available (even if it only indirectly relate to the questions being asked). This book generally assumes that a dataset has been obtained, some of the issues around obtaining data are discussed in chapter 13.

  The data should contain as little noise, in practice the available data may be very noisey and cleaning may be very time-consuming,
- the ability to deal effectively with uncertainty, and an awareness of personal cognitive biases,[823]

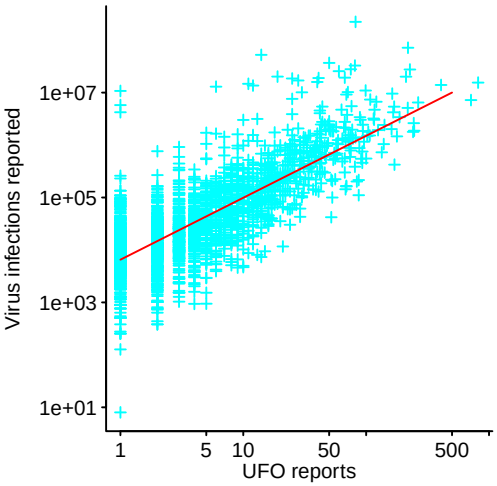


Figure 8.1: Number of virus infections and UFO sighting, reported in 3,072 U.S. counties during 2010; the line is a fitted regression model of the form: $virus\_reports \propto UFO\_reports^{1.2}$. Data from Jacobs et al.[908] Github–Local

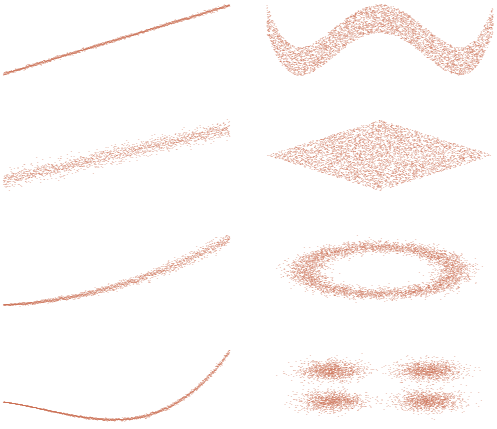


Figure 8.2: Data having values following various visual patterns, when plotted. Github–Local

- statistical analysis techniques capable of providing answers to the desired level of certainty; in practice it may not be possible to draw any meaningful conclusions from the data or more questions will be uncovered.

Data analysis is like programming, in that people get better with practice; there are a few basic techniques that can be used to solve many problems and doing what you did on a previous successful project can save lots of time.

This, and subsequent chapters explicitly discuss the R code that was used (previous chapters discuss the results of data analysis, not how the analysis was done).

There is no guarantee that the available data contains any information that might be used to answer any of the questions being asked of it.

Considerations used to evaluate possible interpretations of patterns found in data include: model simplicity, consistency with existing models of how things are believed to work, and how well a model fits the available data. If the data does not contain population information (and it cannot be easily obtained), the extent to which this alternative interpretation is consistent with the report data can be checked. Without appropriate data, alternative interpretation is based on the analysts model of the world from which the data was obtained.

At a bare minimum, the story told by an analysis of data needs to meet the guidelines for truthfulness in advertising that is specified by the national advertising standards' authority. If manufacturers of soap powder have to meet these requirements, when communicating with the public, then so should you.

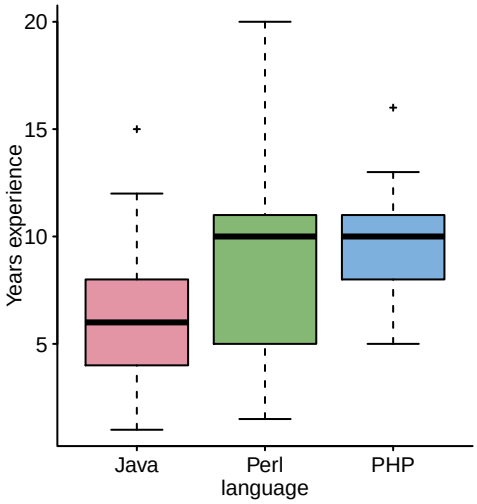


Figure 8.3: Years of professional experience in a given language for experimental subjects. Data from Prechelt.[1523] Github–Local

Check assumptions derived from visualizations Assumptions suggested by a visualization of data need to be checked statistically. For instance, Figure 8.3 shows professional software development experience, in years, of subjects taking part in an experiment using a particular language. The visual appearance suggests that as a group, the PHP subjects are more experienced than the Java subjects. However, a permutation test, comparing years of experience for the PHP and Java developers, shows that the difference in mean values is not significant (there are only nine subjects in each group, and the variation in experience is within the bounds of chance; see Github–communicating/postmortem-answers.R).

8.2 Finding patterns in data

When a specific pattern is expected, the data can be checked to see whether it contains this pattern. Otherwise, the search for patterns is essentially a fishing expedition.

Figure 8.2 shows some common and less common patterns seen in data. The left column shows data forming lines of various shapes; a straight line is perhaps the most commonly encountered pattern in data and points may all be close to the line or form a band of varying width. The right column shows data clustering together in various regular shapes. Uncovering a pattern is the next step along the path to understanding the processes that generated the sample measurements.

Vision is the primary pattern detection pathway used in this book. Animals have developed sophisticated visual pattern detection and recognition systems (see section 2.3); number processing is a very new ability, and as such is relatively slow and unsophisticated.

Compelling numbers. For small quantities of numeric data, the pattern present in the printed form of the values may be the most compelling visual representation. For instance, relative spacing is sometimes used within the visible form of expressions to highlight the relative precedence of binary operators (e.g., more whitespace around the addition operator when it appears adjacent to a multiplication, as in: `5 + 2*3`). Table 8.1 shows that when relative spacing is used, it nearly always occurs in a form that where the operator with higher precedence has closer proximity to its operands (relative to the operator having a lower precedence). The number of cases where the reverse occurs is small, suggesting that either the developer who wrote the code did not know the correct relative precedence or there is a fault in the code.

A study by Landy and Goldstone[1080] found that subjects were more likely to give the correct answer (and answer more quickly) to simple arithmetic expressions, containing two binary operators, when there was greater visual proximity between the operands that were separated by the binary operator having the higher precedence.

	Total	High-Low	Same	Low-High
no-space	34,866	2,923	29,579	2,364
space no-space	4,132	90	393	3,649
space space	31,375	11,480	11,162	8,733
no-space space	2,659	2,136	405	118
total	73,032	16,629	41,539	14,864

Table 8.1: Number of expressions containing two binary operators having the specified spacing in the visible source (i.e., no spacing, no-space, or one or more whitespace characters {excluding newline}, space) between a binary operator and both of its operands. The High-Low column lists counts for expressions where the first operator of the pair has the higher precedence (some are expressions where the both operators of the pair have the same precedence), the Low-High column lists counts for expressions where the first operator of the pair has the lower precedence. For instance, `x + y*z` is space no-space because there are one or more space characters either side of the addition operator and no-space either side of the multiplication operator, the precedence order is Low-High. Data from Jones.[930]

8.2.1 Initial data exploration

Initial data exploration starts with the messy issue of how the data is formatted (lines containing a fixed number of delimited values is the ideal form, because many tools accept this as input; if a database is provided it may be worth extracting the required data into this form).

A programmer's text editor is as good a tool as any for an initial look at data, unless the filename suggests it is a known binary format, e.g., spreadsheet or database. For data held in spreadsheets exporting the required values to a csv file is often the simplest solution.

This initial look at the data will reveal some basic characteristics, such as: number of measurement points (often the number of lines) and number of attributes measured (often the number of columns), along with the kind of attributes recorded, e.g., date, time, lines of code, language, cost estimated, email addresses, etc.

The most important reason for viewing the file with an editor, first, is to identify the character used to delimit columns.

A call to `read.csv` reads the entire contents of a text file into a data frame (what R calls a structure or record type). The file is assumed to contain rows of delimited values (there is an option to change the default delimiter); spurious characters or missing column entries can cause subsequent values to appear in the incorrect column (chapter 14 provides some suggestions for finding and correcting problems such as this). The `foreign` package contains functions for reading data stored in a variety of proprietary binary forms.

Having read the file into a variable, the following functions are useful for forming an initial opinion of the characteristics of the data that has been read (unless the dataset is small enough to be displayed on a screen in its entirety):

- `str` returns information about its argument, e.g., the number of rows and columns, along with the names, types and first few values of each column in a data frame,
- `head` and `tail` print six rows from the start/end of their argument respectively,
- `table` prints a count of the number of occurrences of each value in its argument, e.g., a particular column of a `data.frame` (by default `NA`s are not included). The `cut` function can be used to divide the range of its argument into intervals, and return the bounds of the intervals and the corresponding counts in each interval.

If `str` reports a column having an unexpected type (e.g., `chr` rather than `int`), the likely causes are missing values and spurious characters in the data.

When one or two columns are of specific interest, `plot` can be used to quickly visualize the specific values of interest. Chapter 14 discusses techniques for cleaning data.

Figure 8.4, upper plot, shows a very noticeable change in the number of occurrences, in C source files, of lines having a given length, i.e., number of characters on a line. What might cause this pattern to occur?

The change occurs at around the maximum line length commonly supported by non-GUI, non-flat screen, terminals (these measurements are of C source that is over 10 years old, i.e., before flat screen monitors became available). One hypothesis is that a system limit has a significant impact on the usage characteristics. A prediction derived from this hypothesis is that code written by developers using terminals that supported more characters per line would contain a greater number of longer lines, i.e., the downturn in the plot would move to the right.

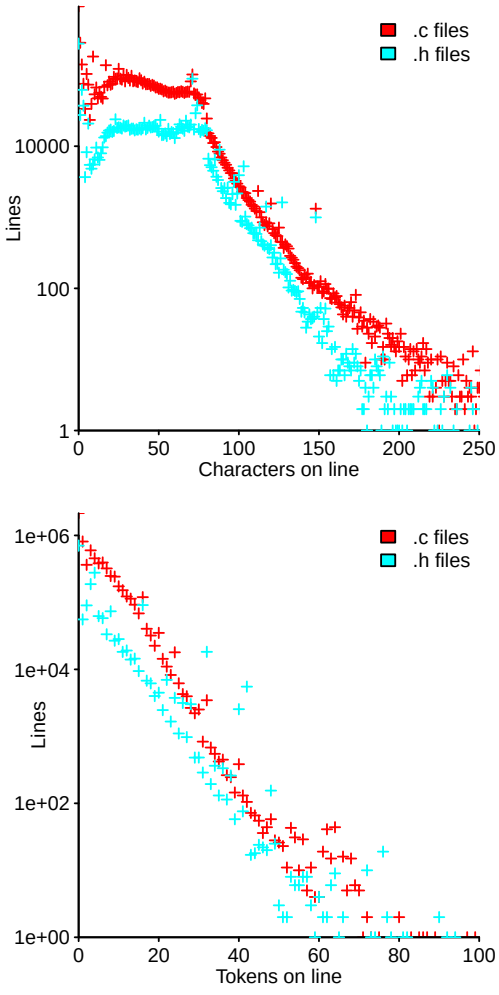


Figure 8.4: Total number of lines of C source, in .c and .h files, having a given length, i.e., containing a given number of characters (upper) and tokens (lower). Data from Jones.[930] Github–Local

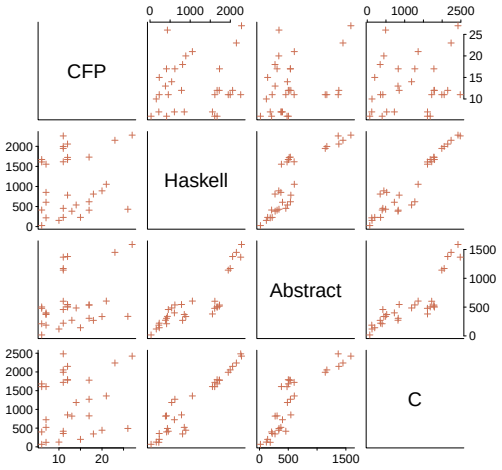


Figure 8.5: Various measurements of work performed implementing the same functionality, number of lines of Haskell and C implementing functionality, CFP (COSMIC function points; based on user manual) and length of formal specification. Data kindly provided by Staples.[1762] Github–Local

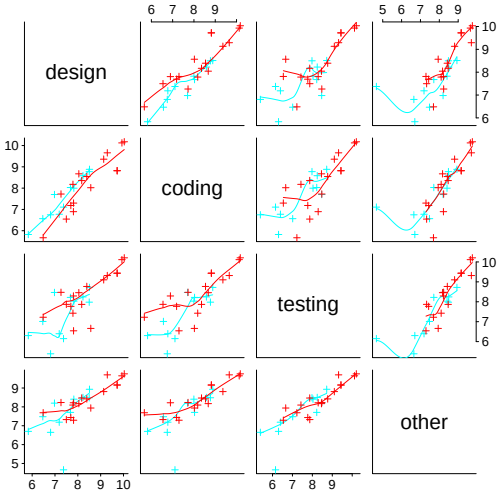


Figure 8.6: Effort, in hours (log scale), spent in various development phases of projects written in Ada (blue) and Fortran (red). Data from Waligora et al.[1917] Github–Local

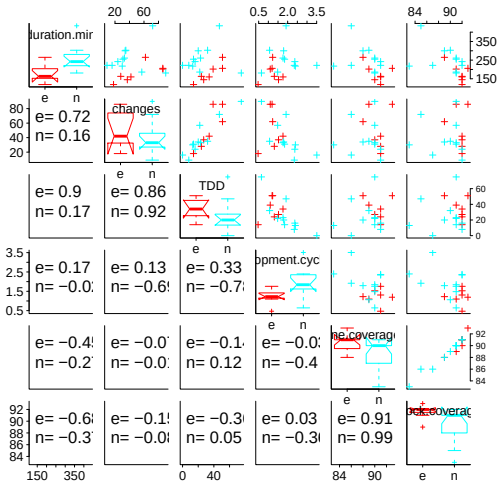


Figure 8.7: Performance of experts (e) and novices (n) in a test driven development experiment. Data from Muller et al.[1326] Github–Local

Figure 8.4, lower plot, illustrates that a different representation of the same information may not have any immediately obvious visual pattern. This plot is a count of the number of tokens per line. Knowing that average token length is around 3-4 characters, suggests that the slight change in the downward slope of the data points just visible at around 25 tokens corresponds to the more dramatic dip seen in the characters-per-line plot.

When a data set contains many variables, plotting one pair of variables at a time is an inefficient use of time. The `plot`, when given a data frame containing three or more columns, creates nested plots of every pair of columns. Figure 8.5 shows four sets of measurements relating to the same task; some measurement pairs are in a roughly linear relationship, while no obvious visual pattern is apparent for other pairs.

```
work=read.csv(paste0(ESEUR_dir, "communicating/pub-fs-fp.csv.xz"), as.is=TRUE)
# -1 removes the first column
plot(work[, -1], col=point_col, cex.labels=2.0)
```

A list of columns can be specified using the formula notation; the following code has the same effect as the previous example:

```
plot( ~ CFP+Haskell+Abstract+C, data=work[, -1], col=point_col, cex.labels=2.0)
```

If a more tailored visualization of pairs of columns is required, the `pairs` function supports a variety of options. For instance, separating out and highlighting subsets of a sample (known as *stratifying*) can be used to highlight differences and similarities. Figure 8.6 separates out measurements of Ada and Fortran projects. The lines are from fitting the points using loess, a regression modeling technique; see below and section 11.2.5.

```
panel.language=function(x, y, language)
{
   fit_language=function(lang_index, col_str)
   {
   points(x[lang_index], y[lang_index], col=pal_col[col_str])
   lines(loess.smooth(x[lang_index], y[lang_index], span=0.7), col=pal_col[col_str])
   }

fit_language(language == "Ada", 2)
fit_language(language != "Ada", 1)
}

# rows 28 and 30 are zero, and we only want columns 16:19
pairs(log(nasa[-c(28, 30) , 16:19]), cex.labels=2.0,
              panel=panel.language, language=nasa$language)
```

The default behavior of `pairs` produces a plot containing redundant information; it is possible to display different information in the upper and lower halves of the plot, and along the diagonal. Figure 8.7 shows expert and novice performance (time taken to complete various tasks and final test coverage) in a test driven development task, with a boxplot along the diagonal and correlation between each pair of attributes, for the two kinds of subjects, in the lower half of the plot. This plot, which primarily uses the default values for its visual appearance, needs more work before being presented to customers.

```
panel_user=function(x, y, user)
{
expert=(user == "e")
points(x[expert], y[expert], col=pal_col[1])
points(x[!expert], y[!expert], col=pal_col[2])
}

panel_correlation=function(x, y, user)
{
expert=(user == "e")
r_ex=cor(x[expert], y[expert])
r_nov=cor(x[!expert], y[!expert])
txt = paste0("e= ", round(r_ex, 2), "\n", "n= ", round(r_nov, 2))
text(0.0, 0.5, txt, pos=4, cex=1.6)
}

panel_boxplot=function(x, user)
{
```

```
t=data.frame(x, user)
boxplot(x ~ user, data=t, notch=TRUE, border=pal_col, add=TRUE)
}

pairs( ~ duration.min+changes+TDD+
          log(development.cycle.length)+line.coverage+block.coverage,
        data=tdd, cex.labels=1.3,
        upper.panel=panel_user, lower.panel=panel_correlation,
        diag.panel=panel_boxplot, user=tdd$user)
```

The `splom` function in the `lattice` package supports creating more complex pair-wise plots.

As the number of columns increases, the amount of detail visible in a `pairs` plot decreases. The correlation between pairs of columns can be compactly displayed, and provides the minimally useful information, i.e., a linear relationship exists.

The `corrgram` package implements various techniques for displaying correlation information. Figure 8.8 shows the correlation between every pair of 27 columns, with correlation used to control the color of each entry. In the upper triangle blue/clockwise denotes a positive correlation, and red/anti-clockwise a negative one; in the lower triangle the numeric values are also blue/red colored; looking at the numbers, more reader effort is needed to locate pairs having a high correlation (coloring reduces the effort).

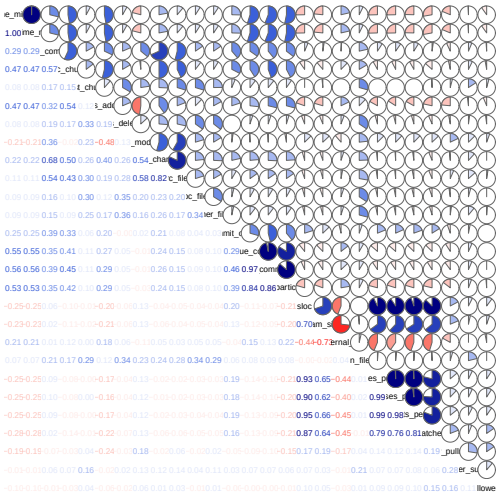


Figure 8.8: Correlations between pairs of attributes of 12,799 Github pull requests to the Homebrew repo, represented using numeric values and pie charts. Data from Gousios et al.[720] Github–Local

Having column names appear along the diagonal creates a compact plot; when many columns are involved this form of display is better suited to situations where the names follow a regular pattern. The `plotcorr` function in the `ellipse` package places column names around the outside; see Github–communicating/pull-req-cor.R.

```
library("corrgram")

corrgram(ctab, upper.panel=panel.pie, lower.panel=panel.shade)
```

Hierarchical clustering is another technique for finding columns that share some degree of similarity, based on a user supplied distance metric. The `hclust` function requires the user to handle the details; the `varclus` function in the `Hmisc` package provides a higher level interface. The following code uses `as.dist` to map the cross-correlation matrix returned by `cor` to a distance, to produce figure 8.9:

```
library("dendextend") # for coloring effects

# Cross correlation
ctab = cor(used, method = "spearman", use="complete.obs")

pull_dist=as.dist((1-ctab)^2)
t=as.dendrogram(hclust(pull_dist), hang=0.2)

col_pull=color_labels(t, k=5)
col_pull=color_branches(col_pull, k=2)
plot(col_pull, main="", sub="", col=point_col, xlab="", ylab="Height\n")
mtext("Pull related variables", side=1, padj=14, cex=0.7)
```

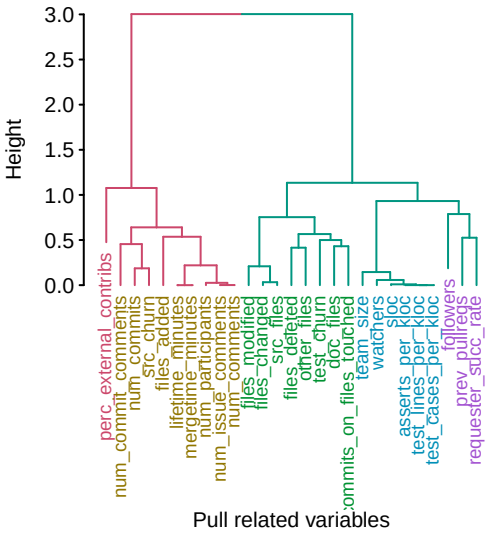


Figure 8.9: Hierarchical cluster of correlation between pairs of attributes of 12,799 Github pull requests to the Homebrew repo. Data from Gousios et al.[720] Github–Local

8.2.2 Guiding the eye through data

It may be difficult to reliably estimate the path of a central line through a collection of points, in a plot (according to some other goodness of fit criteria; section 11.2.5 contains a more detailed discussion of fitting a trend line to data). A way to quickly add such a line to an existing plot is to use the `loess.smooth` function, with the following code producing figure 8.10:

```
plot(res_tab$Var1, res_tab$Freq, col=pal_col[2],
        xlab="SPECint result", ylab="Number of computers\n")

lines(loess.smooth(res_tab$Var1, res_tab$Freq, span=0.3), col=pal_col[1])

scatter.smooth(res_tab$Var1, res_tab$Freq, span=0.3, col=point_col,
        xlab="SPECint Result", ylab="Number of computers\n")
```

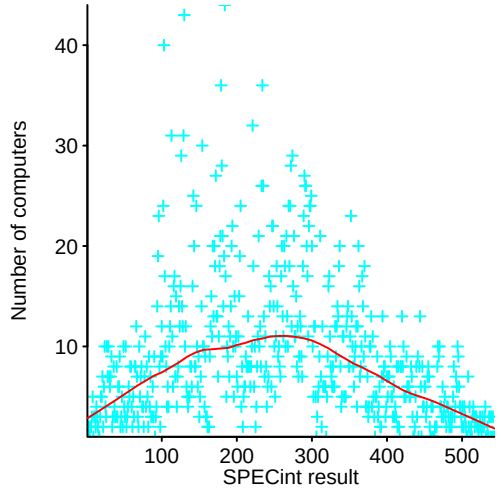


Figure 8.10: Number of computers having a given SPECint result; line is a loess fit. Data from SPEC.[1742] Github–Local

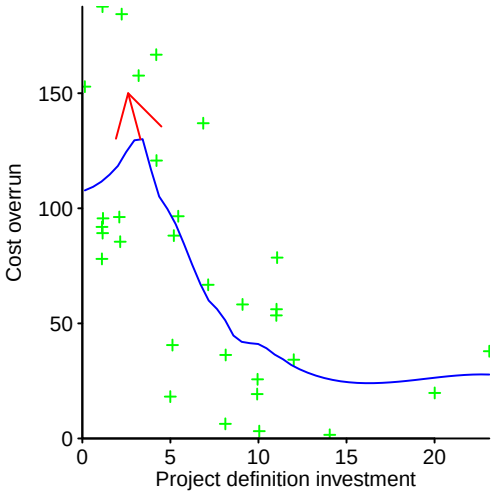


Figure 8.11: Effort invested in project definition (as percentage of original estimate) against cost overrun (as percentage of original estimate). Data extracted from Gruhl.[750] Github–Local

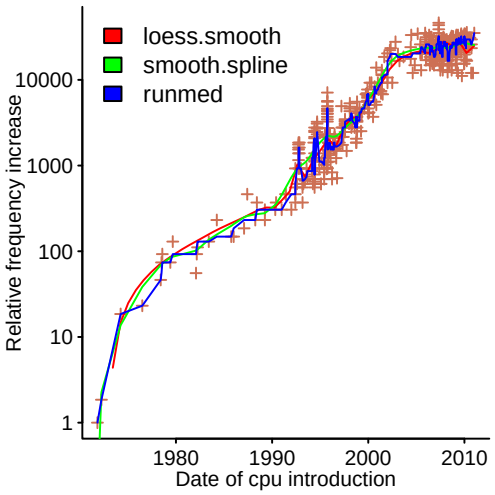


Figure 8.12: Relative clock frequency of cpus when first launched (1970 == 1). Data from Danowitz et al.[436] Github–Local

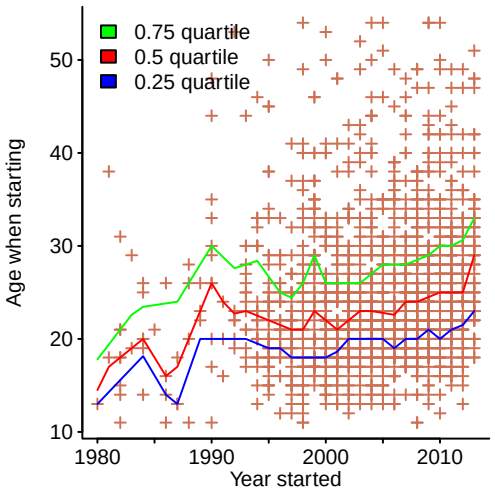


Figure 8.13: Year and age at which survey respondents started contributing to FLOSS, i.e., made their first FLOSS contribution. Data from Robles et al.[1596] Github–Local

The `scatter.smooth` function both plots and draws the loess line (no options are available to control the color of the line).

Plotting a fitted line is a way of visually showing that expectations of a pattern of behavior is being followed (or not). Figure 8.11 shows a loess fit (green) to NASA data[750] on cost overruns for various space probes, against effort invested in upfront project definition; the upward arrow shows the continuing direction of the line seen in the original plot created by one user of this data (who was promoting a message that less investment is always bad).

There are a variety of techniques for calculating a smooth line that is visually less noisy than drawing a line through all the points. Splines are invariably suggested in any discussion of fitting a smooth curve to an arbitrary set of points; the `smooth.spline` function will fit splines to a series of points and return the x/y coordinates of the fitted curve.

Splines originated as a method for connecting a sequence of points by a visually attractive smooth curve, not as a method of fitting a curve that minimises the error in some measurement. LOESS is a regression modeling technique for fitting a smooth curve that minimises the error between the points and the fitted curve; the `loess.smooth` function fits a loess model to the points and return the x/y coordinates of the fitted curve.

Both splines and loess can be badly behaved when asked to fit points that include extreme outliers, or have regions that are sparsely populated with data. The running median (e.g., `median(x[1:k])`, `median(x[(1+1):(k+1)])`, `median(x[(1+2):(k+2)])` and so on for some `k`) is a smoothing function that is robust to outliers; the `runmed` function calculates the running median of the points and returns these values (the points need to be in increasing, or decreasing, order).

Figure 8.12 shows the relative clock frequency of cpus introduced between 1971 and 2010; the various lines were produced using the values returned by the `smooth.spline`, `loess.smooth` and `runmed` functions; see fig 14.4. Don't be lulled into a false sense of security by the lines looking very similar, the *smoothing parameter* provided by each function was manually selected to produce a visually pleasing fit in each case; the mathematics behind the functions can produce curves that look very different, and the choice of function will depend on the kind of curve required and perhaps be driven by the characteristics of the data.

```
plot(x_vals, y_vals, log="y", col=point_col,
       xlab="Date of cpu introduction", ylab="Relative frequency increase\n")
lines(loess.smooth(x_vals, y_vals, span=0.05), col=pal_col[1])

# smooth.spline and runmed don't handle NAs
t=!is.na(x_vals) ; x_vals=x_vals[t] ; y_vals=y_vals[t]
t=!is.na(y_vals) ; x_vals=x_vals[t] ; y_vals=y_vals[t]

lines(smooth.spline(x_vals, y_vals, spar=0.7), col=pal_col[2])

t=order(x_vals)
lines(x_vals[t], runmed(y_vals[t], k=9), col=pal_col[3])
```

Lines drawn through a sample of measurements values often follow the path specified by a central location metric, e.g., the mean value. In more cases it may be more informative to fit a line such that 25% of measurements are below/above it, or some other percentage; *quantile regression* is a popular technique used for fitting such lines. Figure 8.13 is based on a study[1596] of 2,183 replies from a survey of FLOSS developers; two questions being the year and age at which those responding first contributed to FLOSS.

If you find yourself writing lots of algorithmic R code during initial data exploration, you are either investing too much effort in one area, or you have found what you are looking for and have moved past initial exploration. Why are you writing lots of R, there is probably a package that does most of what you want to do and perhaps even more.

8.2.3 Smoothing data

Measured values sometimes fluctuate widely around a general trend (the data is said to be *noisey*). Smoothing the data can make it easier to see any pattern that might be present in the clutter of measured values. The traditional approach is to divide the range of measurement values into a sequence of fixed width bins and count the number of data points in each bin; the plotted form of this binning process is known as a *histogram*.

Histograms have the advantage of being easy to explain to people who do not have a mathematical background and existing widespread usage means that readers are likely to have encountered them before. Until the general availability of computers, histograms also had the advantage of keeping the human effort needed to smooth data within reasonable limits.

Figure 8.14, upper plot, shows a count of computers having the same SPECint result, aggregated into 13 fixed width bins (the number of bins selected by the `hist` function for this data).

```
hist(cint$Result, main="", col=point_col,
        xlab="SPECint result", ylab="Number of computers\n")
```

The `histogram` package supports a wider range of functionality and more options than is available in the base system functions.

The advantage of the binning approach to smoothing and aggregating data is ease of manual implementation, and for this reason it has a long history. The disadvantages of histograms are: 1) changing the starting value of the first bin can dramatically alter the visual outline of the created histogram, and 2) they do not have helpful mathematical properties.

A technique that removes the arbitrariness of histogram bins' starting position is averaging over all starting positions, for a given bin width (known as a *average shifted histogram*); this is exactly the effect achieved using kernel density with a rectangular kernel function.

It often makes sense for the contribution made by each value to be distributed across adjacent measurement points, with closer points getting a larger contribution than those further away. This kind of smoothing calculation is too compute-intensive to be suited to manual implementation, but are easily calculated when a computer is available.

The distribution of values across close measurement points is known as *kernel density*; histograms are the manual labourer's poor approximation to Kernel density, if a computer is available use the better technique.

The `density` function returns a kernel density estimate (which can be passed to `plot` or `lines`); the following code produced the lower plot in figure 8.14:

```
plot(density(cint$Result))
```

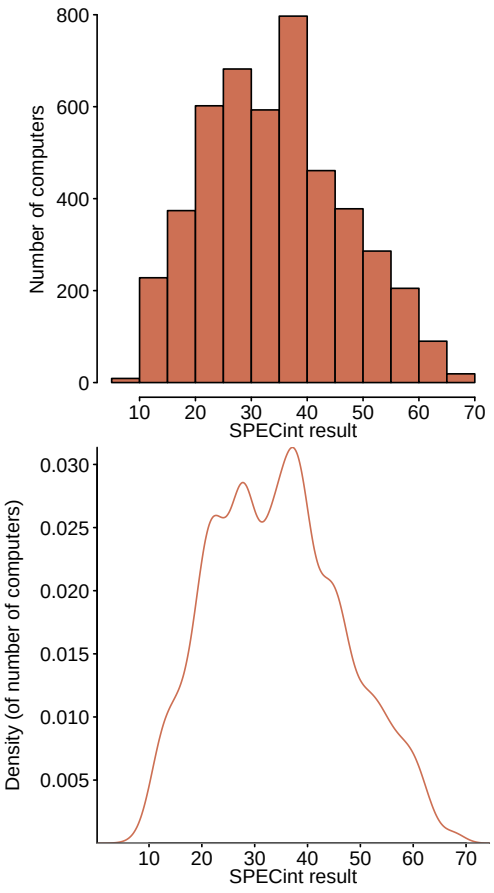


Figure 8.14: Number of computers with a given SPECint result, summed within 13 equal width bins (upper) and kernel density plot (lower). Data from SPEC.[1742] Github–Local

Density plots also perform well when comparing rapidly fluctuating measurements of related items. Figure 8.15, upper plot, shows the number of commits of different lengths (in lines of code) to the Linux filesystem code, for various categories of changes; the lower plot is a density plot of the same data.

The kernel density approach generalizes to more than one dimension; see the `KernSmooth` and `ks` packages.

When dealing with measurements that span several orders of magnitude, a log scale is often used. Creating a histogram using a log scale requires the use of bin widths that grow geometrically (coding is needed to get the `hist` function to use variable width bins; the `histogram` package contains built-in support for this functionality), and bin contents has to be expressed as a density (rather than a count). A histogram based on counts, rather than density, can produce misleading results; figure 8.16 was produced by the following code, where `y` is assigned decreasing values (the histogram should be continuously decreasing and not show a second peak, which is an artifact generated by inappropriate analysis):

```
x=1:1e6
y=trunc(1e6/x^1.5)
log_y=log10(y)

hist(log_y, n=40, xlim=c(0, 3),
        main="", xlab="log(quantity)", ylab="Count\n")
```

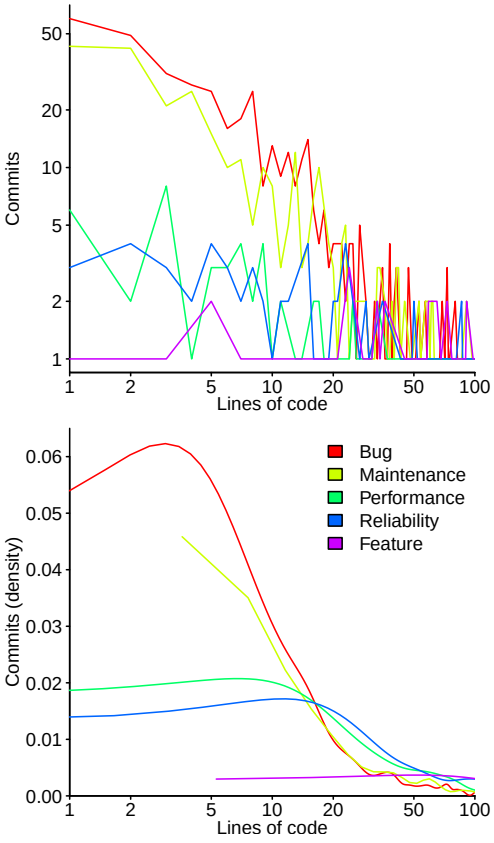


Figure 8.15: Number of commits containing a given number of lines of code made when making various categories of changes to the Linux filesystem code (upper), and a density plot of the same data (lower). Data from Lu et al.[1164] Github–Local

8.2.4 Densely populated measurement points

Some samples contain data whose characteristics produce result in plots containing lots of ink and little visual information; some common characteristics of the density of values include:

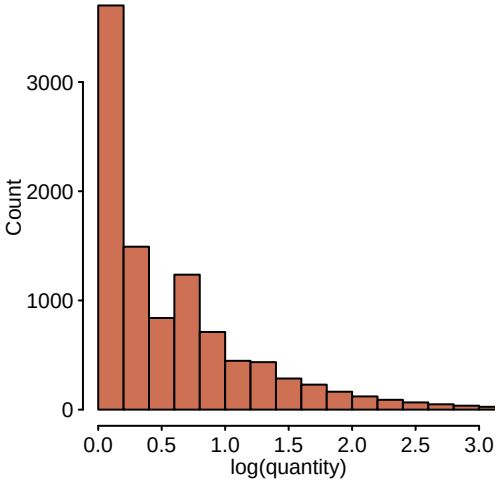


Figure 8.16: Histogram of the log of some measured quantity. Github–Local

- adjacent values on the x-axis having widely different values on the y-values, e.g., figure 8.10,
- multiple points having the same x/y value, all combined visually as a single point in a plot, e.g., figure 8.17,
- many very similar values that merge into a formless mass, when plotted, e.g., figure 8.18.

A plot of values gives a misleading impression when multiple measurements have the same value, i.e., a single point represents many measurements (the problem is more likely to occur for measurements that can only take a small set of values, e.g., discrete values); see figure 8.17, upper plot. The `jitter` function returns its argument with a small amount of added random noise; the middle plot of figure 8.17 shows the effect of jittering the values used in the upper plot. Another possibility is for the size of the plotted symbol to vary with the number of measurements at a given point (see figure 8.17, lower plot); as discussed elsewhere, people are poor at estimating the relative area and so size should not be treated as anything more than a rough indicator.

```
plot(maint$est_time, maint$act_time, col=point_col, xlab="",
                                        ylab="Actual hours\n")

plot(jitter(maint$est_time), jitter(maint$act_time), col=point_col,
                xlab="Estimated hours", ylab="Actual hours\n")

library("plyr")
t=ddply(maint, .(est_time, act_time), nrow)
plot(t$est_time, t$act_time, cex=log(1+t$V1), pch=1, col=point_col,
                                xlab="", ylab="Actual hours\n")
```

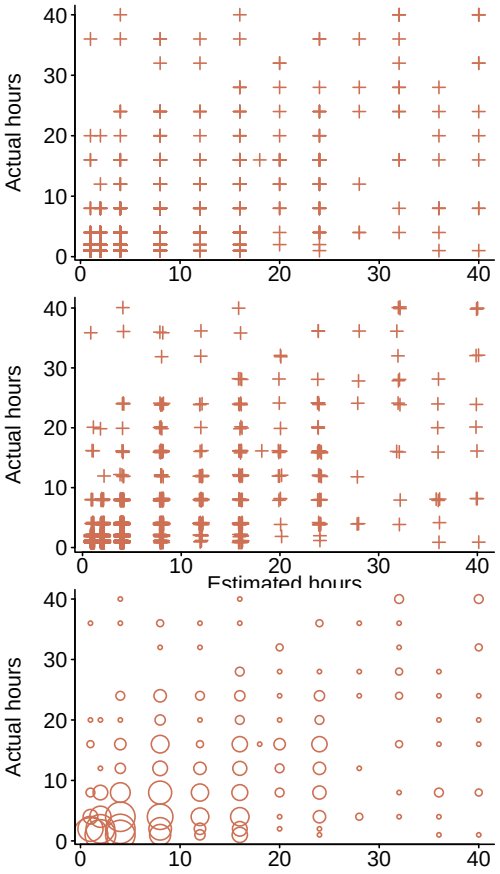


Figure 8.17: Developer estimated effort against actual effort (in hours), for various maintenance tasks, e.g., adaptive, corrective and perfective; upper as-is, middle jittered values and lower size proportional to the log of the number measurements. Data from Hatton.[786] Github–Local

A different kind of communications problem occurs when data points are so densely packed together, that any patterns that might be present are hidden by the visual uniformity (figure 8.18, upper plot; also see fig 11.23). One technique for uncovering patterns in what appears to be a uniform surface is to display the density of points. The `smoothScatter` function calculates a kernel density over the points to produce a color representation (middle plot); contour lines can be drawn with `contour` using the 2-D kernel density returned by `kde2d` (lower plot).

```
plot(udd$age, udd$insts, log="y", col=point_col,
                xlab="Age (days)", ylab="Installations\n")
# Bug in support for log argument :-(
smoothScatter(udd$age, log(udd$insts),
                xlab="Age (days)", ylab="log(Installations)\n")

library("MASS")

plot(udd$age, udd$insts, log="y", col=point_col,
       xlab="Age (days)", ylab="Installations\n")

# There is no log option, so we have to compress/expand ourselves.
d2_den=kde2d(udd$age, log(udd$insts+1e-5), n=50)
contour(d2_den$x, exp(d2_den$y), d2_den$z, nlevels=5, add=TRUE)
```

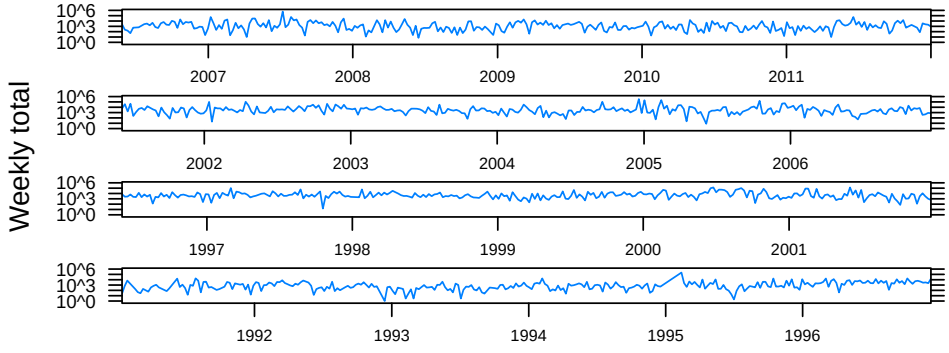


Figure 8.19: Number of lines added to glibc each week. Data from González-Barahona et al.[704] Github–Local

The `hexbin` package is available for those who insist on putting values into bins, in this case using hexagonal binning to support two dimensions.

One solution to a high density of points in a plot, is to stretch the plot over multiple lines; the `xyplot` function, in the `lattice` package, can produce a strip-plot such as the one in figure 8.19, produced by the following code:

```
library("lattice")
library("plyr")

cfl_week=ddply(cfl, .(week),
               function(df) data.frame(num_commits=length(unique(df$commit)),
                                       lines_added=sum(df$added),
                                       lines_deleted=sum(df$removed)))

# Placement of vertical strips is sensitive to the range of values
# on the y-axis, which may have to be compressed, e.g., sqrt(...).
t=xyplot(lines_added ~ week | equal.count(week, 4, overlap=0.1),
               cfl_week, type="l", aspect="xy", strip=FALSE,
               xlab="", ylab="Weekly total",
               scales=list(x=list(relation="sliced", axs="i"),
                           y=list(alternating=FALSE, log=TRUE)))
plot(t)
```

8.2.5 Visualizing a single column of values

The available data may contain a single column of values, or only one column of interest, i.e., there is no related column that can be used to create a 2-D plot. A *box-and-whiskers* plot (or *boxplot* as it is more generally known) is a traditional visualization technique that is practical to perform manually. Figure 8.20 highlights the following characteristics:

- median, i.e., the point that divides the number of values in half,
- first/third or lower/upper quartile, the 25th/75th percentiles respectively,
- lower/upper hinges, the points at a distance $\pm 1.5 \cdot IQR$ where IQR is the interquartile range (the difference between the lower quartile and the upper quartile). The dotted line joining the hinges to the quartile box are the whiskers,
- outliers, all points outside the range of the lower/upper hinge.

The `boxplot` function produces a boxplot; the argument `notch=TRUE` can be used to create a plot that includes a *notch* indicating the 95% confidence interval of the median; right boxplot in figure 8.20.

```
box_inf=boxplot(eclipse_rep$min.response.time, log="y",
                boxwex=0.25, col="yellow", yaxt="n",
                notch=TRUE, xlim=c(0.9, 1.3), ylab="")
```

When a computer is available to do the calculation, more visually informative techniques can be used. What is known as a *violin plot* uses a kernel density of the values, as the outline of the container image; see figure 8.21 (a mirror image is usually included in the plot, hence the name). The `vioplot` function in the `vioplot` package is used for the violin plots in this book.

```
library("vioplot")

vioplot(log(eclipse_rep$min.response.time), col="yellow", colMed="red",
        ylim=range(log(eclipse_rep$min.response.time)),
        xlab="", ylab="log(Seconds)")
```

Formula notation can be used to display multiple violin plots in the same plot, with the following code producing figure 8.22:

```
vioplot(time ~ group+task, data=gs, horizontal=TRUE, col=pal_col,
               xlab="Time (minutes)", ylab="")
```

A bar chart with error bars is regularly used to visually summarise values (sometimes known as *dynamite plots*). A study[403] investigating the effectiveness of various ways of visually summarizing data (including boxplots, violin plots and others) found that when extracting information from bar charts (with or without error bars) subjects did not perform as well as they did when using the other techniques.

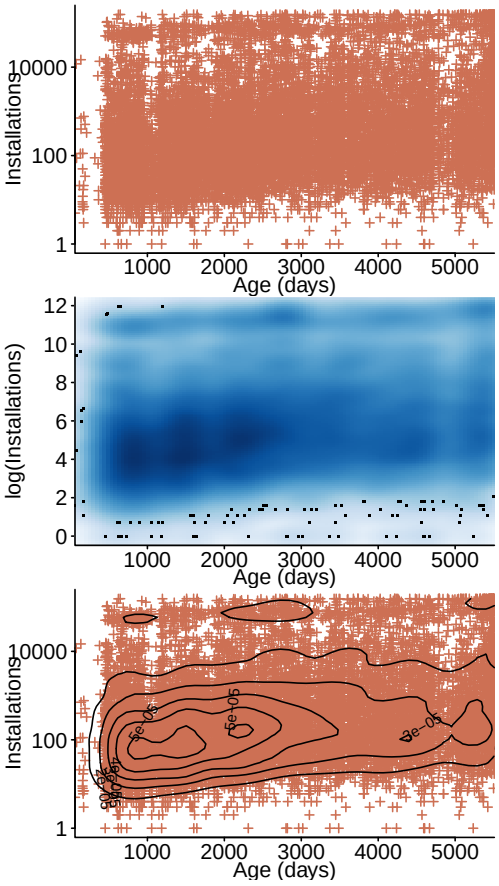


Figure 8.18: Number of installations of Debian packages against the age of the package; middle plot was created by `smoothScatter` and lower plot by `contour`. Data from the "wheezy" version of the Ultimate Debian Database project.[1862] Github–Local

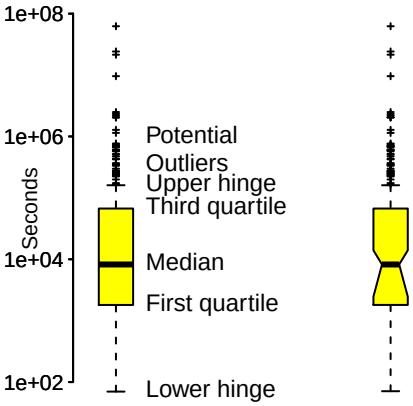


Figure 8.20: Boxplot of time between a potential mistake in Eclipse being reported and the first response to the report; right plot is notched. Data from Breu et al.[252] Github–Local

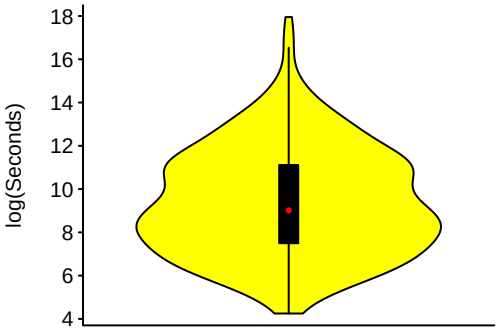


Figure 8.21: Violin plot of time between bug being reported in Eclipse and first response to the report. Data from Breu et al.[252] Github–Local

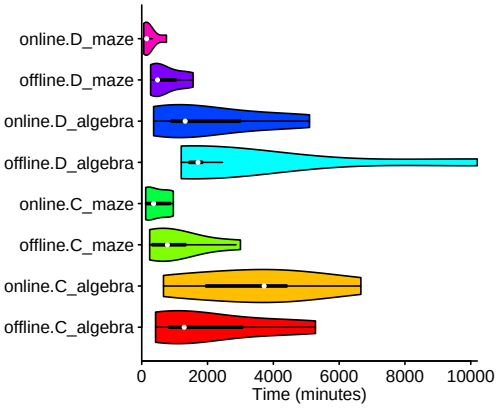


Figure 8.22: Time taken for developers to debug various programs using batch processing or online (i.e., time-sharing) systems. Data kindly provided by Prechelt.[1522] Github–Local

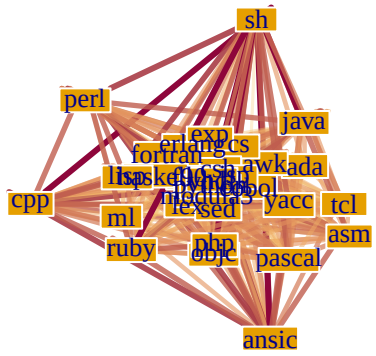


Figure 8.23: Pairs of languages used together in the same GitHub project with connecting line width, color and transparency related to number of occurrences. Data kindly supplied by Bissyande.[203] Github–Local

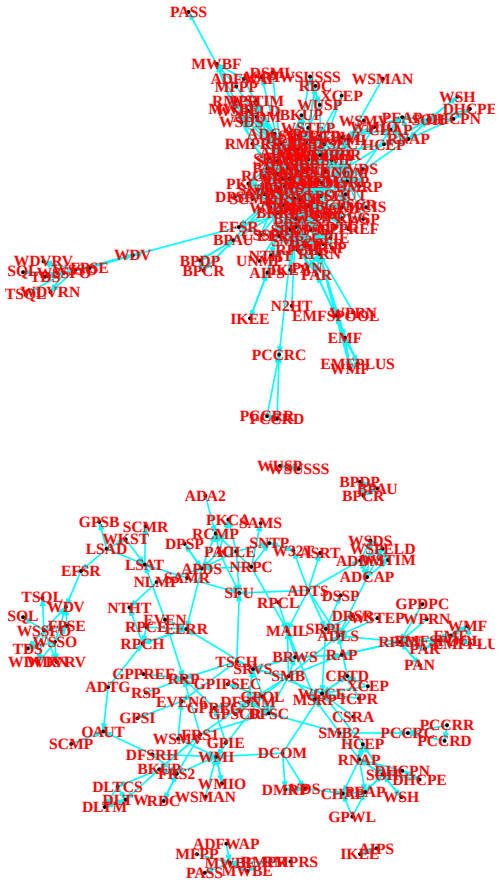


Figure 8.24: References from one document to another in the Microsoft Server Protocol specifications. Data extracted by your author from the 2009 document release.[1277] Github–Local

8.2.6 Relationships between items

The relationship between two entities may be the attribute of interest. Graphs are the data structure commonly associated with relationships, and the `igraph` package contains numerous functions for processing graphs.

When displaying graphs containing large numbers of nodes, potentially useful information in the visual presentation may be swamped by many nodes having relatively few connections. Figure 8.23 is an attempt to show which languages commonly occur, within the same project, with another language, in a sample of 100,000 GitHub projects. The number of projects making use of a given pair of languages is represented using line width and to stop the plot being an amorphous blob the color and transparency of lines also changes with number of occurrences.

Perhaps items having relatively few connections are the ones of interest. The Microsoft Server protocol specifications[1277] contain over 16 thousand pages, across 130 documents (the client specification documents are also numerous). Figure 8.24, upper plot, shows dependencies between the documents (based on cross-document references in the 2009 release[1277]); the lower plot shows the dependencies after excluding the 18 most referenced documents (plot based on the following code):

```
library("igraph")
library("sna")

interest_gr=graph.adjacency(interest, mode="directed")

# V(interest_gr)[names(in_deg)]$size=3+in_deg^0.7
V(interest_gr)$size=1
V(interest_gr)$label.color="red"; V(interest_gr)$label.cex=0.75
E(interest_gr)$arrow.size=0.2

plot(interest_gr)
```

It is possible to use R to draw presentable graphs, however, if your primary interest is drawing visually attractive graphs containing lots of information, then there other systems that may be easier to use, e.g., GraphViz.[728] Yes, an R interface to these systems may be available, but if statistical analysis is not the primary purpose, why is R being used?

Alluvial plots are a method for visualizing the flow between connected entities. Figure 8.25 shows factors used to prioritize the application of Github pull requests, and the relative orders in which they appear in a dataset of pull requests;[721] the `alluvial` package was used.

8.2.7 3-dimensions

We live in a world of three spatial dimensions, which is only one more than the two dimensions available on flat screens and paper; various techniques for enhancing a flat surface to display information in one more dimension are available.

Heatmaps use color to display information about a third quantity within a 2-D plot. Figure 8.26 shows the L3 cache bandwidth (color+number) of an Intel Sandy Bridge processor running at various clock frequencies and using various combinations of cores.

Both the `heatmap` function in the base system and the `heatmap.2` function in the `gplots` package, clusters the rows/columns and then plots a dendrogram; various arguments have to be set to switch off this default behavior, with `heatmap` doing its best to make life difficult including not coexisting with other plots in the same image; `heatmap.2` is more reasonable.

The `levelplot` function in the `lattice` package provides straightforward functionality for producing heat maps, and it is used to produce all the heatmaps in this book.

```
library("lattice")

t=levelplot(L3_band,
            col.regions=rainbow(100, end=0.9),
            xlab="Clock frequency (Mhz)", ylab="Cores used",
            scales=list(x=list(cex=0.70, rot=35),
                        y=list(cex=0.65)),
```

```
                panel=function(...)
                        {
                        panel.levelplot(...)
                        panel.text(1:11, rep(1:8, each=11),
                                        L3_band, cex=0.55)
                        })

plot(t, panel.height=list(3.8, "cm"), panel.width=list(6.2, "cm"))
```

A contour plot can be used for visualizing the relationship between a response variable and two explanatory variables; the `contour` function is part of the base system. A study by Thereska, Doebel, Zheng and Nobel[1828] measured the performance of various applications running on a variety of desktop computers; the cpu speed and memory capacity of the computer hosting each of the 4,924,467 user sessions was recorded. The contours in figure 8.27 are based on the number of user sessions measured on computers having a given processor speed and memory capacity.

```
library("plyr")

Um=unique(memcpu$MemorySize)
M_map=mapvalues(memcpu$MemorySize, from=Um, to=rank(Um))

Us=unique(memcpu$ProcSpeed)
S_map=mapvalues(memcpu$ProcSpeed, from=Us, to=rank(Us))

cnt_mat=matrix(data=0, nrow=length(Us), ncol=length(Um))

cnt_mat[cbind(S_map, M_map)]=log(memcpu$Session_Count)

contour(x=seq(min(Us)/max(Us), 1, length.out=length(Us)),
        y=seq(min(Um)/max(Um), 1, length.out=length(Um)),
        z=cnt_mat, col=pal_col, nlevels=10, axes=FALSE,
        xlim=c(min(Us)/max(Us), 1), ylim=c(min(Um)/max(Um), 1),
        xlab="Processor speed (GHz)",
        ylab="Memory size (Mbyte)\n")

axis(1, at=sort(Us)/max(Us), labels=sort(Us))
axis(2, at=sort(Um)/max(Um), labels=sort(Um))
```

A variety of functions are available for representing a 3-D plot as on 2-D surface, including the `scatterplot3d` function in the `car`, and the `plot3d` function in the `rgl` package.

Histograms in 3-dimensions provide more opportunities, than histograms in 2-dimensions, for looking impressive with little data and misleading viewers. A study by Hamill and Goseva-Popstojanov[773] investigated the origin of 1,257 faults in 21 large safety critical applications, recording where the fixes were made, e.g., requirements, design, code or supporting files. Figure 8.28 shows a 3-D histogram of root cause/fix location on the x-y axis and a count of occurrences on the z-axis. Color has the effect of enhancing the visual appearance of the plot, and makes it easier to locate stacks having similar values, but it is very difficult to obtain detailed information from this plot. Adding numeric values would provide detail, but the real issue is what information is the plot intended to communicate?

```
library("lattice")
library("latticeExtra")

# log transform pulls out small differences in majority of counts
transform_breaks= exp(do.breaks(range(log(1e-4+STVR_col$occurrences)), 20))
t=cloud(occurrences ~ fix+fault, STVR_col,
                panel.3d.cloud=panel.3dbars,
                xlab="Fixes involved", ylab="Fault found", zlab="Count",
                xbase=0.5, ybase=0.5, aspect=c(1, 1),
                col.facet = level.colors(STVR_col$occurrences,
                                at = transform_breaks,
                                col.regions = rainbow),
                scales=list(arrows=FALSE, distance=c(2, 1.1, 1),
                            x=list(rot=-20) # Rotate tick labels
                        ))
plot(t)
```

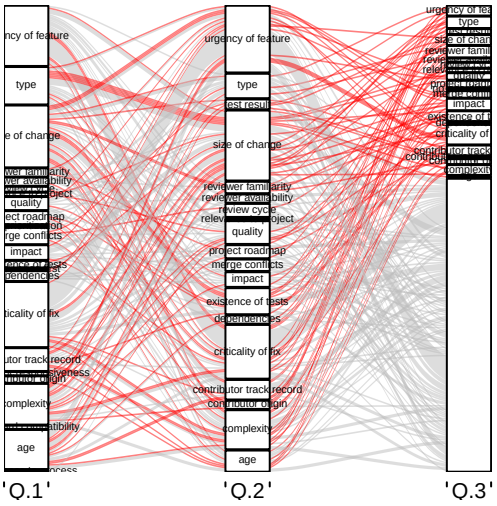


Figure 8.25: Alluvial plot of relative prioritization order of selection and application of Github pull requests. Data from Gousios et al.[721] Github–Local

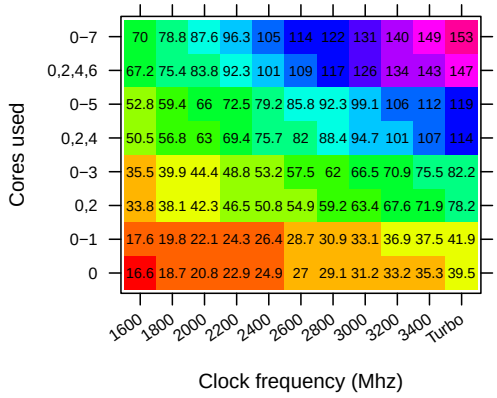


Figure 8.26: Intel Sandy Bridge L3 cache bandwidth in GB/s at various clock frequencies and using combinations of cores (0-3 denotes cores zero-through-three, 0,2,4 denotes the three cores: zero, two and four). Data from Schone et al.[1650] Github–Local

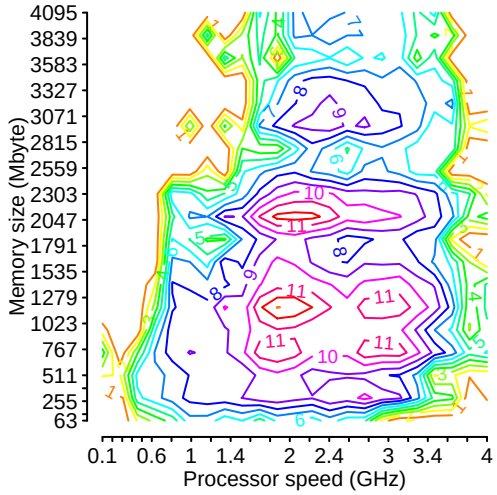


Figure 8.27: Contour plot of number of sessions executed on a computer having a given processor speed and memory capacity. Data kindly provided by Thereska.[1828] Github–Local

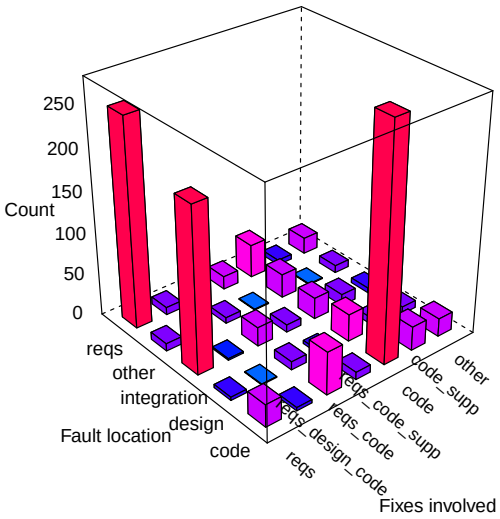
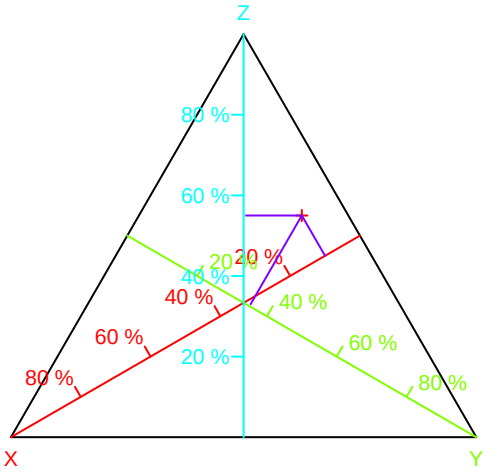


Figure 8.28: Root source of 1,257 faults and where fixes were applied for 21 large safety critical applications. Data from Hamill et al.[773] Github–Local



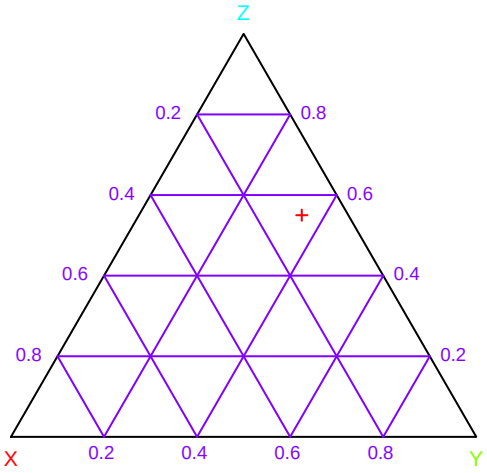


Figure 8.29: Ternary plots drawn with two possible visual aids for estimating the position of a point (red plus at x=0.1, y=0.35, z=0.55); axis names appear on the vertex opposite the axis they denote. Github–Local

A ternary, or triangle, plot has three axes. The axes are inclined at an angle of 60°to each other, and practice is needed to become proficient at estimating the coordinates of any point. Figure 8.29 shows two ways of labelling a ternary plot (with the three coordinates summing to 100%), with labels appearing at the vertex rather than along the axis and axis scales drawn either perpendicular to the axis or labeled along the axis and within the triangle as a grid. The upper plot shows how lines perpendicular to the appropriate axis are used to find the location of a point (at 10, 35, 55 in this case).

The closer points are to a vertex the larger the value of the corresponding variable, the closer points are to an axis the smaller the value of the corresponding variable.

Ternary plots are used to visualize compositional data (see fig 5.32); the `compositions` and `vcd` packages include support for creating ternary plots.

In the following code `rcomp` normalises its argument (so that rows sum to 100) using an interval scale and returns an object having class `rcomp` (the `compositions` package has overloaded functions for handing objects of this type):

```
library("compositions")

xyz=c(10, 35, 55)
plot(rcomp(xyz), labels="", col="red", mp=NULL)
ternaryAxis(side=-1:-3, labels=paste(seq(20, 80, by=20), "%"),
                pos=c(0.5,0.5,0.5), col.axis=hcl_col, col.lab=pal_col,
                small=TRUE, aspanel=TRUE,
                Xlab="X", Ylab="Y", Zlab="Z")

lines(rcomp(rbind(xyz, c(10, 45, 45))), col=hcl_col[4])
lines(rcomp(rbind(xyz, c(32, 35, 33))), col=hcl_col[4])
lines(rcomp(rbind(xyz, c(22, 23, 55))), col=hcl_col[4])

plot(rcomp(xyz), labels="", col="red", mp=NULL)

isoPortionLines(col=hcl_col[4])
ternaryAxis(side=0, col.axis=hcl_col, small=TRUE, aspanel=TRUE,
                Xlab="X", Ylab="Y", Zlab="Z")
```

8.3 Communicating a story

Results from data analysis are of no value unless they are reliably communicated to the target audience.[362] Reliably communicating information to other people is difficult; the intended message may be misunderstood or important parts may simply be overlooked by readers. The data analyst has to provide a narrative that tells the intended story. How people process visual information is discussed in section 2.3.

No known algorithm is available that selects a method that ensures communication is made in a way that will be correctly interpreted by the audience.[i] The main techniques available for presenting numeric information, and how they might be implemented using R, are covered in the rest of this chapter.

There are a wide variety of ways of presenting information, e.g., tables, pie charts, bar charts and scatter plots. Which of these is best, at communicating information to readers? The answer from a wide range of studies is that it depends on what information readers are trying to obtain, and the following is a brief summary of some research findings:

- graph or table? Studies have found that except for reading-off specific values (and recall of these values later), subjects perform better with line graphs, than tables. However, while graphs have better performance when presenting a given perspective (e.g., by selection of the axis), tables may be preferable[241] when wanting to present data in a way that does not favour any one perspective on the data; it boils down to selecting the best cognitive fit,[1893]
- the ability of pie charts to communicate information has been questioned over the years.[418] A study[1745] comparing subject performance using pie charts, a horizontal

[i] Studies have found that people are much better at extracting certain kinds of information when it is presented in the form of frequency of occurrence rather than as a percentage.[678]

divided bar chart, a vertical bar charts and a table, found that except when direct magnitude estimation was required pie charts were comparable to bar charts, but for combinations of proportions pie charts were superior; a study[1718] comparing the three visual clues present in a pie chart (i.e., angle, area and circumference length) found that angle and area were poor methods of communicating information, and that circumference length was the best of three,

- adding a third dimension to a graph has been found to slow down reader performance,[824] i.e., subjects take longer to extract information and may be less accurate. The conclusion would appear to be, don't use three dimensions when two will do. While subjects have expressed a preference for using 3-D graphs to impress others, no studies have investigated whether they have this effect,
- a study by Jansen and Hornbæk[915] investigated the perceived relative size of bars and spheres. Subjects saw an image containing either two bars of different length, or two spheres of different size, and were asked to estimate the size ratio of the two objects. Figure 8.30 shows the actual and estimated bar and sphere ratios for each of the ten subjects, with fitted regression lines;[ii] grey line shows where estimate equals actual. The lower plot shows subjects consistently underestimating the ratio of sphere sizes.
- studies by Cleveland are often cited in R related publications: one study[373] asked subjects to make judgements about graphical information encoded in various ways[iii]; the results showed that accuracy of subjects' answers varied slightly between encoding methods, the ordering from most accurate to least accurate was: position along a common scale, positions along nonaligned scales, length, direction, angle, area, volume, curvature and shading, color saturation. Later studies[1745] suggest that the factors are not so well-defined, with some effects found to be influenced by the structure of the experiments, or performance with a particular encoding depending on the task subjects' performed.

  The thinking behind some layout details used by R's `plot` function are based on experimental work by Cleveland.[372] Although not explicitly stated the aim appears to have been to present data in a workman-like way that avoids the possibility of plotted data values being obscured by plot markings, e.g., tick marks.

The `plot` function is a workhorse for handing the graphical display of data in R; it does a good job of producing a reasonable looking plot from whatever it is passed. Based on this book's implementation goal of using one implementation technique, where-ever possible, the plots in this book were generated using the `plot` function.

The `lattice`[1635] and `ggplot`[325] packages provide alternative world views on the plotting of data; `lattice` is based on the Trellis graphics system[160] from Bell Labs and has an emphasis on multivariate data, while the design of `ggplot` is derived from the work of Wilkinson.[1963][iv] Both `lattice` and `ggplot` provide a great deal of control over the created plot through the use of user supplied functions. While `ggplot` is widely used by experienced R developers, its inability to sensibly handle whatever nonsense data is thrown at it (e.g., nonsense in that there are mistakes in the R code that produced it) prevents this package being recommended for casual use. A detailed technical overview R's graphics subsystems is available in "R Graphics" by Murrell.[1336]

Combining data with visual information familiar to readers helps them to extract patterns that mean something to them. Figure 8.31 shows single event upsets (i.e., radiation induced memory faults) experienced by NASA's Orbview-2 spacecraft during one day in 2000. Overlaying the satellite location at the time of the upset on a map of the Earth (using the `map` package) provides context to help readers understand where most upsets occur.

In some cases the intent of a plot may be to communicate that life is complicated, or that there are a few big fish and many small ones. Figure 8.32 shows an estimate of the market share of Android devices in use in 2015, by brand/company and product name (based on the 682,000 unique devices that downloaded an App from OpenSignal[1421]). Treemaps encode information using area, a quantity that many readers have problems accurately interpreting.

```
library("treemap")
```

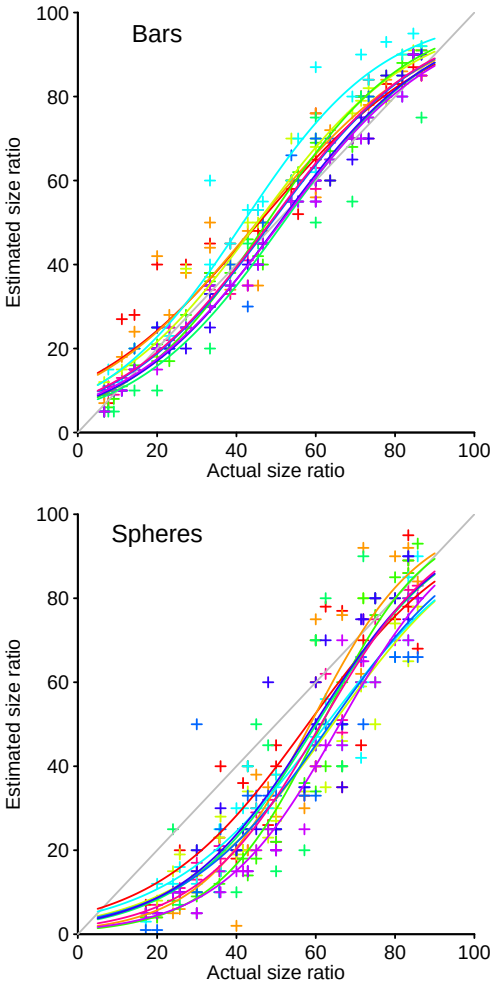


Figure 8.30: Actual and estimated size ratio for bars and spheres, for each of the ten subjects (in different colors, with line from fitted regression model), with grey line showing where estimate equals actual. Data from Jansen et al.[915] Github–Local

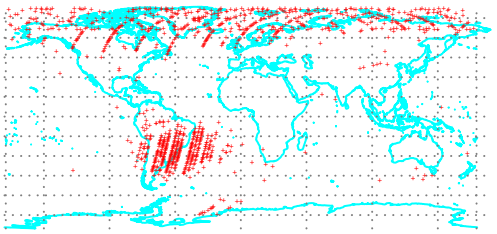


Figure 8.31: Earth relative positions of NASA's Orbview-2 spacecraft when it experienced a single event upset (in blue) on 12 July 2000. Data kindly provided by LaBel.[1500] Github–Local

[ii] Beta regression is used, because it provides a much better fit to the data than the model (based on Stevens' power law) fitted[1744] by Jansen and Hornbæk.

[iii] One study[799] has replicated some of these findings.

[iv] The title of Wilkinson's book "The Grammar of Graphics" refers to the structure of software written to display graphics rather than the structure of the displayed information.

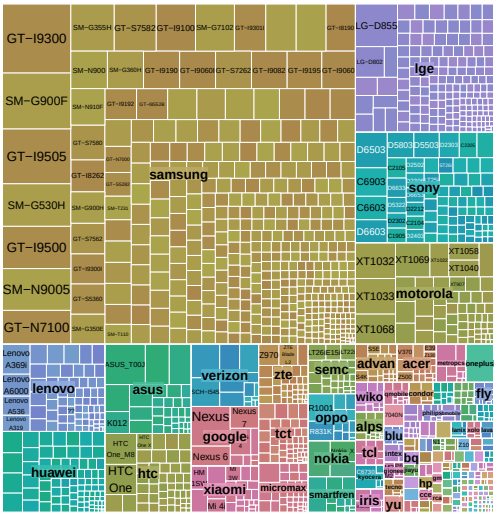


Figure 8.32: Estimated market share of Android devices by brand and product, based on downloads from 682,000 unique devices in 2015. Data from OpenSignal.[1421] Github–Local

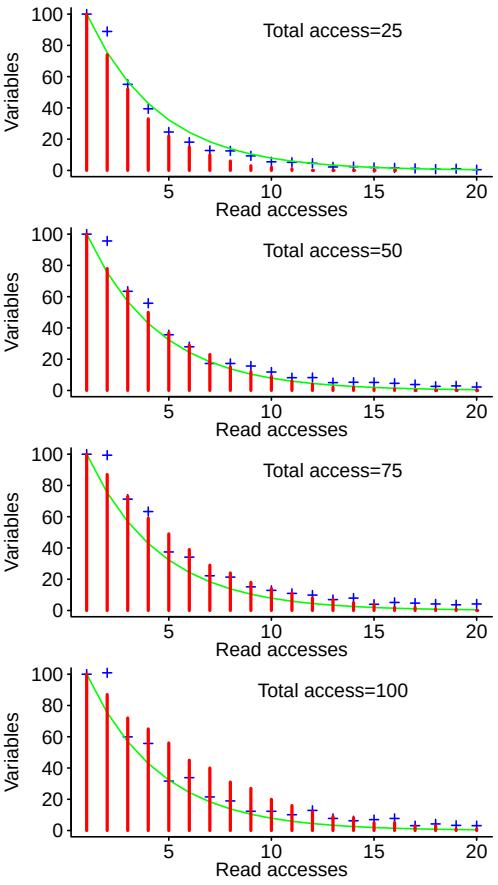


Figure 8.33: Variables having a given number of read accesses, given 25, 50, 75 and 100 total accesses, calculated from running the weighted preferential attachment algorithm (red), the smoothed data (blue), and a fitted exponential (green). Github–Local

```
and_tree=treemap(android, c("brand", "model"), "august2015",
                 title="", palette=pal_col,
                 border.col="white", border.lwds=c(0.5, 0.25))
```

8.3.1 What kind of story?

The kinds of output from statistical data analysis include the following:

- a description of the data, e.g., its mean and variance, how measurements cluster, an equation summarizing the data. A descriptive model, built from the data, can be used to help gain insights into the system that was measured, for comparing different descriptions (e.g., benchmark results) and for building similar systems (e.g., automatically creating file system contents[17] for benchmarking purposes)
- a model built to mimic the behavior of a system (as expressed in the measurements made), e.g., a simulator,
- a predictive model capable of making appropriately accurate predictions for values not in the set of measurements used to build the model. The possible range of prediction values may be within the range of values used to build the model or outside the range of these values, e.g., making predictions about a future time,

A standard reply to any complaints about the adequacy of a model built using data is the adage "All models are wrong, but some are useful."

An example of the different kinds of model that can be built, and how their usefulness depends on the problem they are intended to solve, is provided by a question involving the use of local variables in the source code of a function definition.

If the source code contains a total of N read accesses to variables defined locally within the function, what percentage of variables will be read from once, twice and so on (based on a static count of the visible source code, not a dynamic count obtained by executing the function)?

Data from an analysis of C source[930] provides a description of "what is". Plotting the data shows that a few variables account for most accesses, i.e., read from. After some experimentation the following equation was found to be a good fit to the data (see figure 8.33):

$pv = 34.2 \times e^{-0.26acc-0.0027N}$, where: pv is the percentage of variables, acc the number of read accesses to a given variable, and N is the total number of accesses to all local variables within a function. For example, when a function contains a total of 30 read accesses of its local variables, the expected percentage of variables accessed twice is: $34.2 \times e^{-0.26\times 2-0.0027\times 20}$.

What other kind of model can be built to answer this question?

This problem has a form that has parallels with the growth of new pages and links to existing pages on the World Wide Web. Each access of a local variable could be thought of as a link to the information contained in that variable. One idea that has been found to be integral to modeling the number of links between web pages is *Preferential attachment*.

With some experimentation an iterative algorithm based on preferential attachment, was created, that produced a pattern of behavior close to that seen in the data. The algorithm is as follows:

Assume we are automatically generating code for a function, and from the start of the function, to the current point in the code L distinct local variables exist (and have been accessed), with each accessed R_i times ($i = 1, \ldots, L$). The following weighted preferential attachment algorithm is used to select the next local variable to access (global variables are ignored in this analysis):

- With probability $\frac{1}{1+0.5L}$ create a new variable to access,[v]
- with probability $1 - \frac{1}{1+0.5L}$ select a variable that has previously been accessed in the function, choose an existing variable with probability proportional to $R + 0.5L$ (where R is the number of times the variable has previously been read from;); e.g., if the total accesses up to this point in the code is 12, a variable that has had four previous read accesses is $\frac{4+0.5\cdot 12}{2+0.5\cdot 12} = \frac{10}{8}$ times as likely to be chosen as one that has had two previous accesses.

[v] The unweighted preferential attachment algorithm uses a fixed probability to decide whether to access a new variable.

The red points in figure 8.33 were calculated using the above algorithm.

This preferential attachment model provides insights into local variable usage that are very different from those provided by the fitted exponential equation. Neither of them could not be said to be realistic descriptions of the process used by developers, when writing code. Both models are descriptions of the end result of the emergent process of writing a function definition; each model has its own advantages and disadvantages, including the following:

- the fitted equation is fast and simple to calculate, while the output from the iterative model is slow (an average over 1,000 runs in the example code) and requires more work to implement,
- the iterative model automatically generates a possible sequence of accesses (for machine generated source), while a fitted equation does not provide any obvious method of generating a sequence of accesses,
- multiple executions of an iterative model can be used to obtain an estimate of standard deviation, while the equation does not provide a method for estimating this quantity (it may be possible to fit another regression model that provides this information),
- the equation provides an end-result way of thinking, while the iterative model provides a choice-based way of thinking about variable usage.

A common technique for devising a model for a new problem is to find a very similar problem that has a proven model, and to adapt this existing model to the new problem. A model based on existing practice is often easier to sell to an audience, than a completely new model.

Some multiprocessor system have a "shared nothing" architecture, which minimises the sharing of hardware resources. Performance measurements of such systems, under various loads, shows that even when tasks can be evenly distributed across all X processors in the system, performance is rarely X times faster. Which model provides a good explanation of the performance seen?

Amdahl's law predicts changes in multiprocessor performance as the number of processors used changes, where the multiprocessor system has a shared hardware architecture. Gunther[757] extended this "law" to cover multiprocessors having "shared nothing" architecture; the adapted model, plus a further adaption, are not good fits to the data.

Gunther[758] created a model based on queuing theory and simulated model performance (with each job waiting in a queue for time t_1 and executing for time t_2). The argument for using queuing theory is that data sharing between different programs can create resource contention that the "shared nothing" hardware architecture cannot unblock. Figure 8.34 shows that the queuing model more accurately follows the pattern measured. Given the small amount of data available it would be unwise to attempt further model tuning.

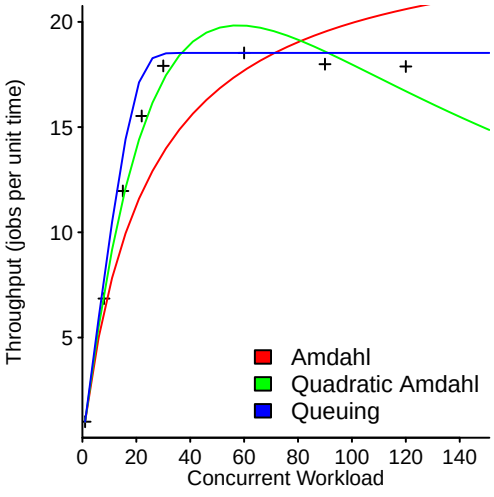


Figure 8.34: Throughput when running the SPEC SDM91 benchmark on a Sun SPARCcenter 2000 containing 8 CPUs, with the predictions from three fitted queuing models. Data from Gunther.[758] Github–Local

The R language does not contain features designed with simulation in mind[vi], but like most languages it can be adapted to solve problems outside its core domain; see the `simFrame` and `simmer` packages.

Finding a workable model, based on the available data, can involve many iterations over a long time. For instance, modeling the growth of the size and number of files/directories in a filesystem has a long history, with current models[1300] either involving a mixture of two distributions for the equation fitting approach, or a generative approach based on simulating the way new files are created from existing files.

Perhaps the most important question to answer when proposing any model is the purpose to which it will be put. A model intended to gain insight might not be of any use in making practical recommendations, and a model used to make predictions might not provide any useful insight. For instance, modeling the connection between modifications to files and the introduction of mistakes, which cause faults to be experienced may be used to predict coding mistake rates based on modification history, but such a model has limited scope for directly deriving methods for reducing faults, e.g., reduce faults by reducing file modifications, is of no use when customers want new or modified behavior in the applications they use.

In most cases, a great deal of domain knowledge is required to build a model having the desired level of performance. There is no guarantee that any created model will be sufficiently accurate to be useful for the problem at hand; this is a risk that occurs in all

[vi] Interfacing to Netlogo[1527]

model building exercises. Ideally model building is driven by a theory describing the behavior of the system being modeled. When a theory is complete there is no need for new models, the fact that the creation of a new model is being considered implies that existing models are lacking in some respect.

8.3.2 Technicalities should go unnoticed

The machinery of information presentation should not get in the way of reader's access to that information.

There are many books offering tips, suggests and recommendations for how best to present visual information; the only book recommended by your author (it is based on a wide range of empirical research) is "Graph Design for the Eye and Mind" by Stephen Kosslyn.[1039] Sometimes multiple plots are used to tell an evolving story, McCloud[1231] is a great introduction to this art form.

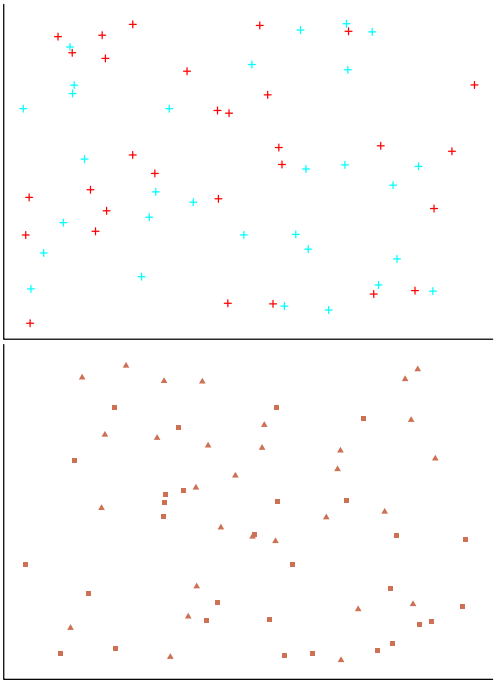


Figure 8.35: Illustration of the difference in cognitive effort needed to locate points differing by shape or color (one is a serial search, while the other operates in parallel). Github–Local

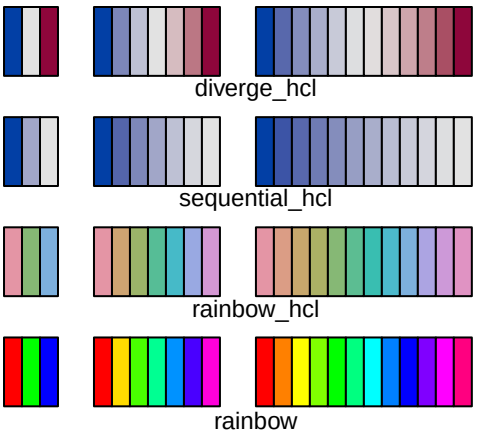


Figure 8.36: The three, seven and twelve color palettes returned by calls to the `diverge_hcl`, `sequential_hcl`, `rainbow_hcl` and `rainbow` functions. Github–Local

8.3.2.1 People have color vision

Until the mid 1980s most people used computer terminals that were only capable of displaying black and white (or green and black). Forty years later, the look-and-feel of mid-1980s computer usage still predominates in serious works of data visualization.

This book treats color as an essential component of numeric story telling. Color provides an extra dimension that enables more information to be present within the same area, and make it easier for viewers to extract information from a plot.[vii]

Selecting the most appropriate colors to use requires skill and experience. The `colorspace` and `RColorBrewer` packages both include functions that automatically select a color palette based on the arguments passed;[viii] the `colorspace` package provides a wider range of functionality than `RColorBrewer`, and is used to select the colors for the plots appearing in this book.

The Hue-Chroma-Luminence (HCL) color space is claimed[2002] to provide a better mapping to the human color perceptual system (hue: dominant wavelength; chroma: colorfulness, intensity compared to gray; and luminence: brightness, amount of gray), than alternative spaces.[ix] The color palettes generated by the `rainbow_hcl` function are considered to be qualitative palettes, that are suitable for depicting different categories; those generated by the `sequential_hcl` function to be suitable for coding numerical information that ranges over a given interval, with the `diverge_hcl` function also encoding numerical information, but including a neutral value.

The `choose_palette` function provides an interactive, slider based, method for developers to define their own color palettes.

Approximately 10% of men and 1% of women have some form of color blindness. The `dichromat` package provides a way of showing how a plot containing color would appear to a viewer having some form of color blindness. The package makes use of experimental data[1901] to simulate the effects of different kinds of color blindness, modifying the requested colors to appear, to normal sighted viewers, like they would to viewers having the selected kind of color blindness.

8.3.2.2 Color palette selection

Figure 8.37 shows how time varying data involving related items (in this case market share of successive versions of Android) can be displayed in a way that preferentially highlights one aspect of the data; the upper plots highlighting individual versions while the lower plots show each version's contribution to overall market share. Bold colors are effective at drawing attention to individual lines, but can be overpowering when a large area of color appears in the plot; the opposite can be the case for pastell colors.

[vii] R contains 657 built-in color names (the `colors` function lists them) and also supports hexadecimal RGB literals.

[viii] The selection process is based on theories derived from the use of color in maps,[253] which has a long history.

[ix] Red-Green-Blue (RGB) is a specification based on the display of color on computer screens; Hue-Saturation-Value (HSV) is a transformation of RGB that attempts to map to the human perceptual system and is used by some other software packages.

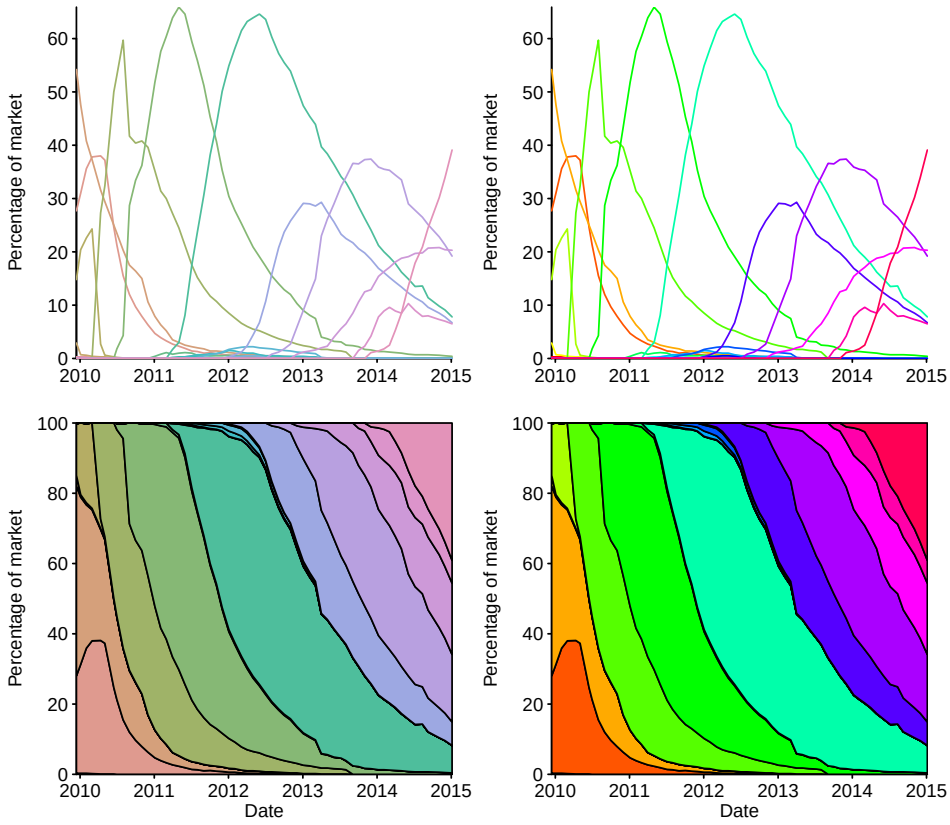


Figure 8.37: Percentage share of Android market by successive Android releases, by individual version (top) and by date (lower); pastell colors on left and bold on right. Data from Villard.[1902] Github–Local

8.3.2.3 Plot axis: what and how

The choice of plotting axis can have a dramatic impact on the visual perception of displayed data.

Linear and logarithmic are the two commonly used axis scales; square-root is common in some application domains. When the range of plotted values span several orders of magnitude, using a logarithmic axis can produce a more informative visualization; compare the use of linear and log axis in figure 8.38. Plotting values drawn from an exponential, or power law-like distribution, using a linear scale often results in many points being visually clumped together in a small area of the plot; use of a log scaled axis has the effect of spreading out these clumped values.

The `plot` function (and many other R plotting functions) automatically selects the minimum/maximum range of each axis, based on the range of data passed; by default 4% is added at each end of the range.

The choice of quantity plotted along each axis is driven by the relationship, between the two quantities, that the data analyst is seeking to highlight; the purpose of the plot is to visually communicate this relationship.

Care needs to be taken to ensure that artificial relationships are not generated by the choice of quantity used for one axis. An example of the wasted effort that can occur, when the relationship implied by the quantities plotted along an axis are not carefully analysed, is provided by the saga of program fault density vs. lines of code.

It was noticed that when fault density (i.e., number of faults divided by lines of code in functions) was plotted against lines of code (in functions), the distribution of points had a pattern that resembled a lopsided U. Some researchers proposed that the minimum of this U represented an optimum for the length of a function.[785]

A study by El Emam, Benlarbi, Goel, Melo, Lounis and Rai[537] showed that this U-shape was an artefact generated by the choice of quantities plotted along each axis. Plotting the ratio $\frac{F}{LOC}$ against LOC, with F constant, will produce a tilted U-shape (blue line in figure 8.39). If the number of faults grows faster than the number of lines of code (which has been found to occur for large line counts) then U-shaped curves such as the red line in figure 8.39 can occur (a growth rate was picked to illustrate one possibility; as in the following code):

```
x=1:100 ; inv.x=1/x
```

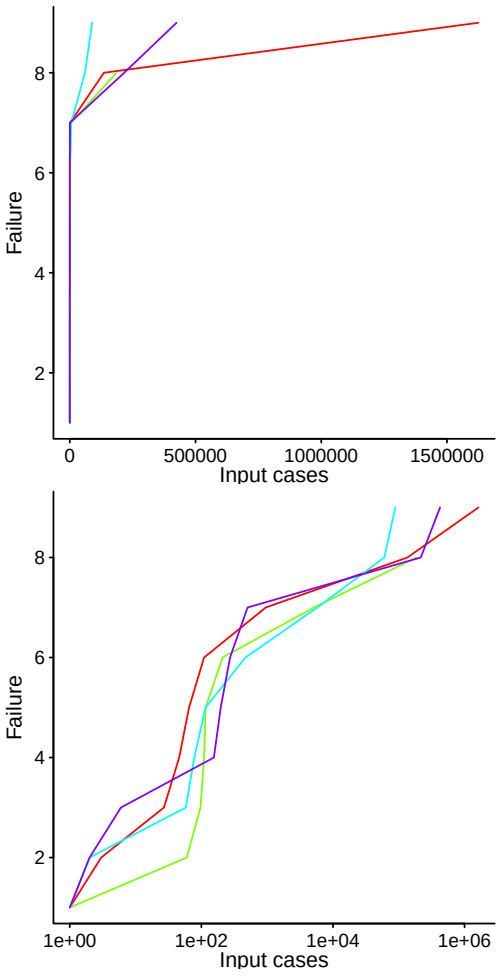


Figure 8.38: Input case on which a failure occurred, for a total of 500,000 inputs; plotted using a linear (upper) and logarithmic (lower) x-axis. Data from Dunham et al.[516] Github–Local

```
plot(x, 3*inv.x, type="l", col=pal_col[1],
                          xlab="LOC", ylab="Faults/LOC\n")
lines(x, ((x+50)^3/5e4)*inv.x, col=pal_col[2])
```

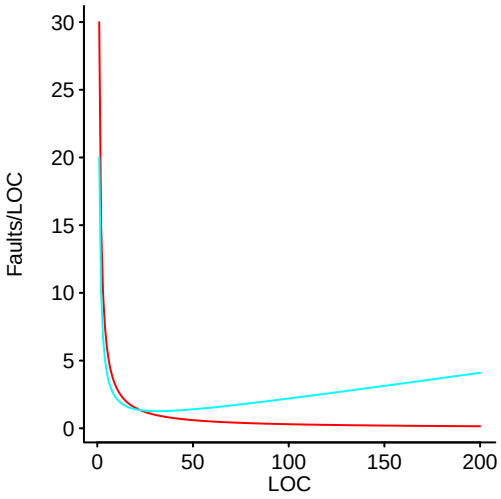


Figure 8.39: Example of U-shape created when y-axis values are a ratio calculated from x-axis values. Github–Local

The idea suggested by the U-shaped pattern, that there might be an optimal function length, is the result of misinterpreting the mathematical behavior of a ratio quantity plotted against one of the values used in the ratio calculation, i.e., the pattern seen in plots is an artefact of the choice of quantities chosen for each axis.

By spreading data out, a log transform of an axis can sometimes visually hide potentially useful information, rather than help reveal it. Figure 8.40 is from a study by Putnam and Myers[1538] (their Figure 8.3); in both cases the x-axis is log transformed. In the right plot the y-axis is linear, and there is a visually distinct cluster of measurements across the top; in the lower plot, where both axes are log transformed, this cluster is visually less prominent.

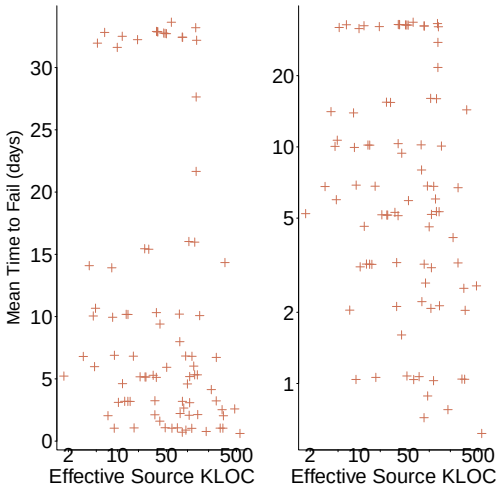


Figure 8.40: Mean time to fail for systems of various sizes (measured in lines of code); linear y-axis left, log y-axis right. Data extracted from Figure 8.3 of Putnam et al.[1538] Github–Local

8.3.3 Communicating numeric values

The output from statistical analysis can include visual plots, and a small collection of numbers. What is the best way to communicate a story involving a small collection of numbers?

The uncertainty associated with using descriptive phrases to denote probabilities is discussed in section 2.7.1 and section 6.1.4. Confidence intervals are a practical means of communicating uncertainty and are discussed in section 11.2.1. Some government organizations publish guidance on communicating uncertainty.[780]

Regression modeling (chapter 11) finds a best fit of an equation to data, according to some specified definition of the error between the equation and data. While the numeric values (often referred to as *parameters*) are the output of model building, the information being communicated is equation+parameter values. Giving the final fitted equation, with confidence intervals, provides readers with all the information in one place, i.e., they don't have to decode wording in the text and mentally plug in the fitted values; see section 2.4.1

Complicated equations can exhibit unexpected behavior; figure 8.41 shows the result of plotting the following set of equations, for $-4.7 \le x \le 4.7$:

$$y_1 = c(1,-0.7,0.5)\sqrt{c(1.3,2,0.3)^2 - x^2} - c(0.6,1.5,1.75)$$

$$y_2 = \frac{0.6\sqrt{4-x^2} - 1.5}{1.3 \le |x|}$$

$$y_3 = c(1,-1,1,-1,-1)\sqrt{c(0.4,0.4,0.1,0.1,0.8)^2 - (|x| - c(0.5,0.5,0.4,0.4,0.3))^2} - c(0.6,0.6,0.6,0.6,1.5)$$

$$y_4 = \frac{c(0.5,0.5,1,0.75)\tan\left(\frac{\pi}{c(4,5,4,5)}(|x| - c(1.2,3,1.2,3))\right) + c(-0.1,3.05,0,2.6)}{c(1.2,0.8,1.2,1) \le |x| \le c(3,3,2.7,2.7)}$$

$$y_5 = \frac{1.5\sqrt{x^2+0.04} + x^2 - 2.4}{|x| \le 0.3}$$

$$y_6 = \frac{2||x| - 0.1| + 2||x| - 0.3| - 3.1}{|x| \le 0.4}$$

$$y_7 = \frac{-0.3\,(|x| - c(1.6,1,0.4))^2 - c(1.6,1.9,2.1)}{c(0.9,0.7,0.6) \le |x| \le c(2.6,2.3,2)}$$

A table of numbers covering a wide range of values (e.g., table 8.2) can be difficult to interpret quickly, unless it is something readers regularly do. An alternative representation separates out the mantissa and exponent, and combines them using area and color, allowing a visual same/different comparison to be made: as in figure 8.42.

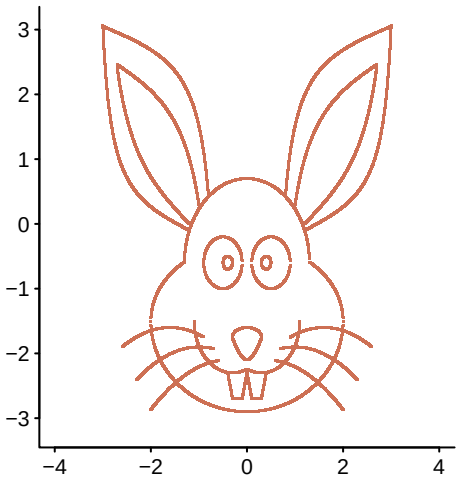


Figure 8.41: What's up doc? Perhaps, not the expected pattern in the data. Equations from White.[1946] Github–Local

Packages are available for integrating the output from R programs into the workflow of various document preparation systems, for instance, the `ascii` package provides functions for producing Asciidoc compatible output, and the `knitr` package produces LaTeX output.

Operation	Approximate runtime
L1 cache reference	1 ns
Branch mispredict	3 ns
L2 cache reference	4 ns
Mutex lock/unlock	17 ns
Main memory reference	100 ns
Send 2K bytes over commodity network	177 ns
Compress 1K bytes with Zippy	2,000 ns
Read 1 MB sequentially: memory	7,000 ns
SSD random read	16,000 ns
Round trip within same datacenter	500,000 ns
Read 1 MB sequentially: magnetic disk	1,000,000 ns
Seek: magnetic disk	3,000,000 ns
Send packet CA→Netherlands→CA	150,000,000 ns

Table 8.2: Numbers Everyone Should Know, circa 2016. Data from Scott.[1660]

8.3.4 Communicating fitted models

What is the most effective way of communicating information about a fitted model to readers?

Software developers are likely to have had lots of experience reading and interpreting equations (which are essentially a form of code). As casual users of statistical analysis, software developers will probably have to put some effort in to correctly interpreting the output produced by the `summary` function, for a fitted model; chapter 11 lists `summary` output because readers need some practice at interpreting it.

Looking at equation 11.2 (copied below), it is not necessary to search through a block of unfamiliar numbers for information about the fitted model parameters:

$$sloc = 1.139 \cdot 10^5 + 3.937 \cdot 10^2 Number\_days$$

If more information needs to be communicated, such as the uncertainty in fitted coefficients, equations enable this information to be specified at the point it applies, e.g., equation 11.3 (copied below):

$$sloc = (1.139 \cdot 10^5 \pm 1.171 \cdot 10^3) + (3.937 \cdot 10^2 \pm 4.205 \cdot 10^{-1}) Number\_days$$

The complete `summary` output (copied below) could be edited down, to remove information that is unlikely to be of interest. But trying to maintain the visual form of the `summary` output serves no useful purpose. Any additional statistical information (e.g., deviance explained) can be listed in a line of text. Github–Local

```
Call:
glm(formula = sloc ~ Number_days, data = kind_bsd)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-82990  -32136   -3609   35389   87324

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.139e+05  1.171e+03   97.24   <2e-16 ***
Number_days 3.937e+02  4.205e-01  936.33   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 1657283104)

    Null deviance: 1.4610e+15  on 4826  degrees of freedom
Residual deviance: 7.9964e+12  on 4825  degrees of freedom
AIC: 116172

Number of Fisher Scoring iterations: 2
```

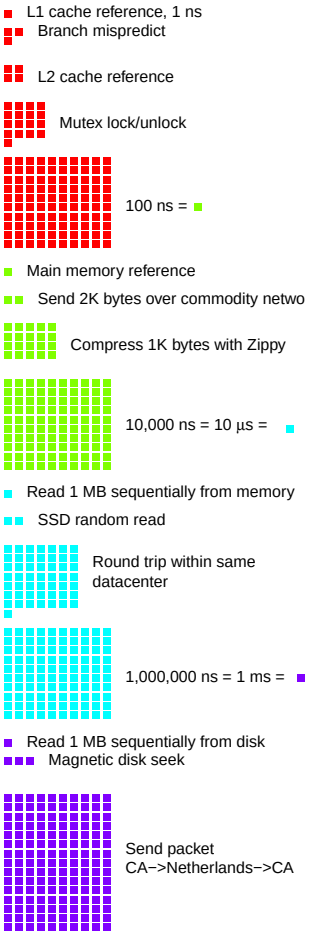


Figure 8.42: Alternative representation of numeric values in table 8.2. Data from Scott.[1660] Github–Local

As model complexity increases, readers have to invest more effort to correctly interpret equations (e.g., equation 11.6, copied below), and the number of readers willing to spend even more effort interpreting `summary` is likely to be less.

$$\begin{aligned} Actual = -274.8 + 1.21 Estimated + 2625 \times !D + \\ C_{fp}(1862 \times !D - 197.6 \times D) + \\ C_{tp}(-2270 \times !D - 462.2 \times D) + \\ C_{ot}(-2298 \times !D - 234.3 \times D) \end{aligned}$$

Chapter 9

Probability

9.1 Introduction

What are the chances of an event occurring?

Probability is the mathematics used to answer this question. Reasons for being interested in the estimation of probabilities include:

- betting, making an insurance decision, and all decisions and predictions involving questions about whether particular cases need to be handled,
- deciding the extent to which an event is surprising. The level of surprise might be used to decide whether something provides an opportunity or is going wrong or, when performing statistical analysis, discriminating between hypotheses.

Readers are assumed to have some basic notion of the concepts associated with probabilities, and to have encountered the idea of probability in the form of likelihood of an event occurring; classic examples involve calculating the probability of a given combination or sequence of values occurring when flipping a coin or rolling a die, e.g., two heads or rolling two sixes, or the probability of having to make N flips/rolls before some event occurs.

What is the difference between probability and statistics?

Probability makes inferences about individual events based on the characteristics of the population, while statistics makes inferences about the population based on the characteristics of a sample of the population[i].

Another way to compare the two is that probability makes use of deductive reasoning, while statistics makes use of inferential reasoning.

Probability and statistics are intertwined in that ideas and techniques from probability, about individual events, may be used when solving problems involving statistics and results about the characteristics of a population, obtained from statistical analysis, may be used to help solve problems involving probability.

People make use of various phrases to express their view of the likelihood of an event occurring, e.g., "almost impossible" and "quite possible". Studies have found large cultural and personal differences in the numeric probabilities assigned to such phrases; see fig 6.7 and fig 2.57.

This book is data driven, and so primarily makes use of statistical analysis. The following example is a problem for which possible answers can be suggested using a probability model (data on developer behavior would provide evidence).

Say, a vendor of a static analysis tool wants to add support for detecting a newly discovered pattern of mistakes made by developers. An occurrence of this pattern, in code, is not always a mistake. What is the upper bound on the probability of generating a false positive, that keeps the likelihood of developers continuing to use the tool above some limit (say 90%)?[ii]

[i] Statistics could be defined as the study of algorithms for data analysis.

[ii] Experience shows that tool false-positives are sufficiently unpopular (they are a source of wasted effort), that a developer will stop using the tool concerned if they are encountered too often. Higher false-positive rates for Tornado warnings result in more deaths and injuries,[1709] through people ignoring the warning.

Answering this question requires knowledge of the mental model used by developers to evaluate analysis tool performance. The following are two possible mental models (both assume zero correlation between difference warning occurrences and that developers assign the same importance to all warning messages):

- an *economic* developer who tracks the benefit of processing each warning (e.g., false positive warning -1 benefit, else $+1$ benefit), starting in an initial state of zero benefit this economic developer stops processing warnings if the current sum of benefits ever goes negative.

  The Ballot theorem gives the probability that, when sequentially processing warnings, the number of true warnings is always greater than the number of false positive warnings (assuming equal weight is given to both cases, the alternative being more complex to analyse). Let C be the number of correct warnings and F the number of false positive warnings and assume $C > F$, then the probability is given by:

  $$\frac{C-F}{C+F}$$

  rewriting in terms of the probability of the two kinds of warning (i.e., $C+F=1$), we get: $C_p - F_p$

  so, for instance, when the false positive rate is 0.25 the probability of a developer processing all the warning generated by a tool is $0.75-0.25 \to 0.5$, and does not depend on the total number of warnings.

- an *instant gratification* developer who processes each warning and stops when a sequence of N consecutive false positive warnings have been encountered. This kind of thinking is analogous to that of the *hot hand in sports* (what psychologists call the clustering illusion).

  What is the probability that a sequence of N consecutive false positive warnings is not encountered?

  If the total number of warnings is k and q is the probability of a false positive occurring, then the probability of a run of N consecutive false positive warnings occurring can be calculated using the following recurrence:

  $$P(k,q,N) = P(k-1,q,N) + q^N(1-q)(1-P(k-N-1,q,N))$$

  with initial values:

  $$P(j,q,N) = 0, \text{ for } j = 0,1,\ldots,N-1$$

  $$P(j,q,N) = q^N, \text{ for } j = N$$

  Figure 9.1 shows the probability of not encountering a sequence of three (red) or four (blue) consecutive false positive warnings when processing some total number of warning messages, for various underlying false positive rates (ranging from 0.5 to 0.2).

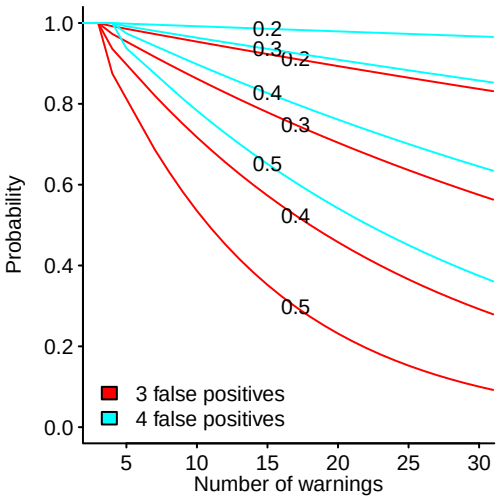


Figure 9.1: Probability that three (red) or four (blue) consecutive false positive warnings occur in some total number of warnings (false positive rate appears on the line). Github–Local

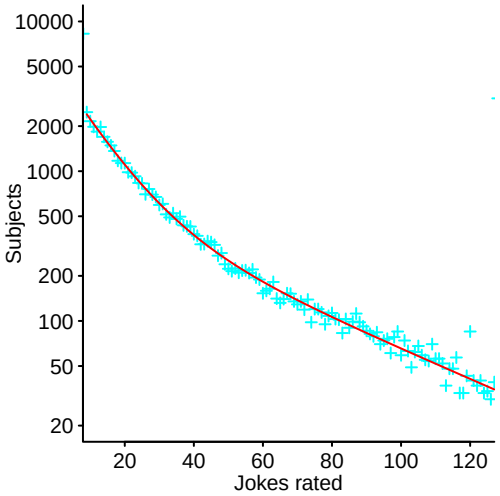


Figure 9.2: Number of subjects rating more than eight jokes, with fitted bi-exponential model; line is a fitted regression model of the form: $Subjects \propto 4200e^{-0.09Jokes} + 650e^{-0.02Jokes}$. Data from Goldberg et al.[695] Github–Local

When dealing with warnings involving complex constructs, a developer may be unwilling to put the effort into understanding the situation and either goes along with what the static analysis tool reports, thus underestimating the actual false positive rate, or defaults to assuming the warning is a false positive, thus overestimating the actual false positive rate.

A study by Goldberg, Roeder, Gupta and Perkins[695] investigated the ratings given to 150 jokes by 54,905 subjects. Subjects rated the jokes online, could choose whether to rate a particular joke or not, and could stop rating at any time. Figure 9.2 shows the number of subjects who rated more than eight jokes; numbers above 127 are somewhat erratic.

Finding an equation, or technique, to use in solving a problem involving probability requires some knowledge of the terminology used in this field. Possible phrases to try in search queries include: birth and death process, coin tossing, colored balls, combination, ergodic, event, fair games, first passage time, generating function, Markov chain, Markov process, occupancy problem, partitions, permutation, random walk, stochastic, trials and urn model.

Finding a closed form solution to an equation can be difficult, even when one exists. Sometimes the processes being studied contains so many interacting components that it is not possible to model them analytically; an alternative approach is simulation, discussed in section 12.5.

9.1.1 Useful rules of thumb

If the distribution of the values taken by some attribute, in a population, is not known, the following inequalities can be used as worst case estimates of the probability of various relationships being true. Both inequalities are distribution independent (the price of this generality is that the bounds are loose).

Markov inequality:

The Markov inequality uses the sample mean, μ, to calculate the maximum probability that X (which is required to be nonnegative) is larger than some constant. The inequality does not make any assumptions about the sample distribution:

$$P(X \geq k) \leq \frac{\mu}{k}$$

where: μ is the sample mean.

Example. If a sample has $\mu = 10$, then the probability of the sample containing a value greater than or equal to 20 (i.e., twice the mean) is: $\frac{10}{20}$.

Chebychev's inequality:

If the standard deviation, σ, of a sample is known, then Chebychev's inequality can be used to calculate a tighter bound than that given by the Markov inequality, as follows:

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

alternatively:

$$P(|X - \mu| \geq k) \leq \frac{\sigma}{k^2}$$

Using the above example, the probability of the sample containing a value that differs from the mean by at last 10 is less than or equal to: $\frac{\sigma}{10^2}$.

Example: an analysis of the number of mutants needed to estimate test suite adequacy to within a specified error and confidence bounds.[711]

Fréchet inequalities:

Bounds on the union and disjunction of two or more probabilities are given by the Fréchet inequalities, as follows:

Logical conjunction: $\max(0, P(a_1) + P(a_2) - 1) \leq P(a_1 \wedge a_2) \leq \min(P(a_1), P(a_2))$

Logical disjunction: $\max(P(a_1), P(a_2)) \leq P(a_1 \vee a_2) \leq \min(1, P(a_1) + P(a_2))$

Correlation between three variable pairs: If the correlation between two pairs of three variables is known, say r_{12} and r_{13}, the bounds on the correlation of the remaining pair, r_{23}, is given by:

$$r_{12}r_{13} - \sqrt{(1 - r_{12}^2)(1 - r_{13}^2)} \leq r_{23} \leq r_{12}r_{13} + \sqrt{(1 - r_{12}^2)(1 - r_{13}^2)}$$

As the number of variables involved increases, the expressions become more complicated.[273]

Rule of three: Say N colored balls are drawn from a box, and the number of balls of each color counted. If there are r red balls, a reasonable estimate of the expected percentage of red balls remaining in the box is: $\frac{r}{N}$. If no green balls have been drawn, what is a reasonable estimate for the number of green balls remaining in the box?

If the fraction of green balls in the box is g, the probability of not having drawn a green ball is: $(1 - g)^N$. The 95% confidence bounds on this occurring is: $(1 - g)^N \leq 0.05$.

$$N \log(1 - g) \leq \log(0.05) \approx -Ng \leq -2.9957 \approx g \leq \frac{3}{N}$$

The non-appearance of any green balls suggests that g is very small, so: $\log(1 - g) \approx -g$.

9.1.2 Measurement scales

Mathematically, measurement values can be characterised as discrete or continuous, along with the properties of the scale used. Possible scales include the following:

- Discrete

– *nominal scale*: each measurement value has an arbitrary number or name. Because the choice of number/name is arbitrary, no ordering relationship exists between different numbers/names. A nominal scale is not a scale in the usual sense of the word.

  Examples: the numbers on the back of footballers' shirt, or the various sales regions in which a product is sold.

– *ordinal scale*: each measurement value is a number or name of an item, and an ordering relationship exists between the numbers/names. The distance between distinct values need not be the same. When names are assigned to entities, there may be cultural differences in the selection process. Figure 9.3 shows how words are assigned to tracts of trees having occupying various surface areas.

  Example: Classifying faults by their severity, e.g., minor, moderate, serious.

  If a minor fault is considered less important than a moderate fault, and a moderate fault is less important than a serious fault, we can deduce that a minor fault is less important than a serious fault.

  Example: The addresses of members of a C structure type is increasing, for successive members, but the difference between member addresses is not fixed because different members can have different types.

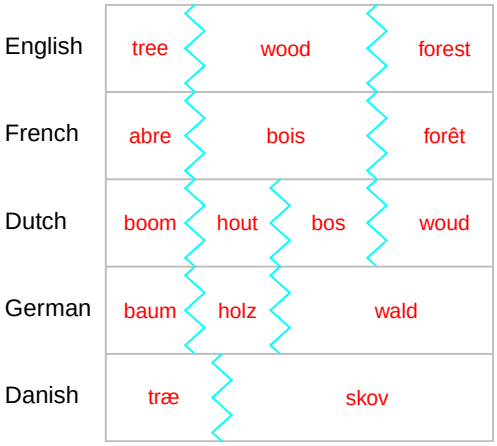


Figure 9.3: The relationship between words for tracts of trees in various languages. The interpretation given to words (boundary indicated by the zigzags) in one language may overlap that given in other languages. Adapted from DiMarco et al.[500] Github–Local

- Continuous

– *interval scale*: each measurement is a number, a relative ordering exists, and a fixed length interval of the scale denotes the same amount of quantity being measured.

  A data point of zero does not indicate the absence of what is being measured.

  Example: the start date of some event is an interval scale. If the start date of events A, B and C are known, and the difference in start date between events A and B is the same as between events C and D, it is possible to calculate the start date of event D.

  Addition and subtraction can be applied to values on an interval scale but not multiplication or division, e.g., it makes no sense to say that the start date of event A is twice that of event C.

– *ratio scale*: each measurement assigns a number to an item and this numeric scale preserves: the ordering of items, the size of the interval between items and the ratios between items. It differs from the interval scale in that a measurement of zero denotes the lack of the attribute being measured.

  The time difference between two events is a ratio scale.

The kinds of statistical analysis that can be legitimately performed on the values in a sample will depend on the kind of measurement scale used.

9.2 Probability distributions

Probability distributions are mathematical descriptions of the properties of values calculated by following a pattern of behavior, i.e., an algorithm. For instance, the flipping a coin pattern of behavior generates one of two results, a fixed probability of either result, with each result being independent of the previous one, and a count of the number of heads and tails has a binomial probability distribution.

If a sample of values can be fitted to a known probability distribution, then information about the pattern of behavior that generated them can be inferred from what is known about processes known to generate values having that particular distribution. For instance, given a list of pairs of numbers, if the ratio formed from each pair (i.e., $\frac{a}{a+b}$) can be fitted to a binomial distribution, there is strong evidence that the pairs are counts of a process producing one of two possible values (e.g., heads/tails, yes/no, etc.), and the probability of producing each value can be calculated from the fitted distribution.

While many probability distributions have been created,[521] only a handful of them are regularly used by analysts; R packages tend to support commonly occurring distributions, with a few packages supporting a wide range of distributions.[521]

Fitting a distribution to a sample is a step towards understanding the processes that generated the measurements, not an end in itself.

Failure to fit a known distribution may mean that more than one distribution is involved, e.g., two different coins are being used and both are biased in some way. Given enough data it is sometimes possible to obtain a reasonable fit that involves two or more distributions.

If there is reason for believing the processes being measured are driven by a known behavior, the quality of fit of the predicted probability distribution to the measured values can be compared; perhaps also against the quality of fit to other distributions.

If there is no expectation of a particular behavior, then finding an acceptable fit of some probability distribution to the measurement values is a starting point for understanding the processes that are driving the measurements observed.

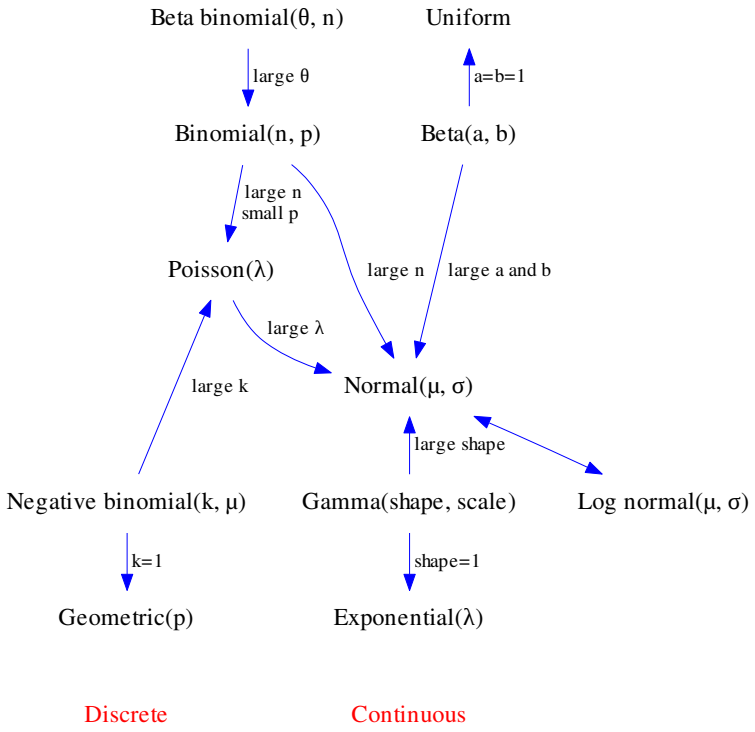


Figure 9.4: Relationships between commonly used discrete and continuous probability distributions.

Every family of probability distributions is completely characterised by a small set of numbers (often one or two) and a formula that the numbers parameterise. For instance, everything about a Normal probability distribution can be calculated by plugging values for the mean, μ, and standard deviation, σ, into the formula: $P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$ (this formula is often abbreviated as $N(\mu, \sigma)$). Fitting data to a Normal distribution involves finding appropriate values for μ and σ.

In practice, a few probability distributions are encountered much more often than others. One of these common cases will often fit reasonably well to a wide spectrum of commonly encountered samples, and unless there are theoretical reasons for expecting a less commonly encountered distributions, there is nothing to be gained by searching through all known distributions to find the one that best fits a sample.

Some characteristics of sample values, that may or may not correspond to a known probability distributions include:

- mean value (sometimes called the *arithmetic mean*, *central tendency* or *location value*, the last two terms may also be used to refer to the median): many distributions have a finite mean (examples that don't include power laws with an exponent greater than or equal to -1 and the Cauchy distribution),
- scale parameter, *variance* (*standard deviation* is the square-root of variance): how spread out the distribution is; a few distributions do not have a finite variance, e.g., power laws with an exponent between zero and -2,
- the extent to which the distribution is symmetrical/asymmetrical about its mean, the *skew* of a distribution is a measure of how asymmetrical it is; a symmetric distribution has a skew of zero, while a positive skew has a tail pointing towards larger positive values, and a negative skew has a tail pointing towards negative values,
- where most of a distribution's density resides, e.g., around the mean or in the tails. The *kurtosis* of a distribution is a measure of how spiky the distribution is; possibilities include tall and slim (known as *leptokurtic*; slender-curved), short and flat (known as *platykurtic*) or medium-curved (known as *mesokurtic*; the Normal distribution has a Kurtosis of three),

- number of distinct peaks, known as the *modality* of a distribution; a distribution with one distinct peak is said to be unimodal, two distinct peaks bimodal (such as measurements from two different distributions, e.g., height of men/women).

The `moments` package contains functions for calculating skewness, kurtosis, and moment related attributes of a numeric vector.

Probability distributions can be divided into discrete and continuous distributions, with discrete distributions only being defined at specific points (usually integer values). In R, functions that involve discrete distributions usually require integer values while functions involving continuous distributions take floating-point values.

There are various ways of representing a probability distribution, and the following are often encountered:

- density function: for discrete distributions (see figure 9.5) this can be viewed as the probability that x will have a given *value*, $P(x = value)$; for continuous distributions (see figure 9.7) the probability of any particular value occurring is zero, however there is a finite probability of a measurement returning a value within a specified interval,
- cumulative density function: the probability that x will be less than or equal to a given *value*, $P(x < value)$, see figure 9.6,
- equation: an equation for the probability distribution. For the majority of people (including your author), this is little more than eye candy, e.g., the equation, $\frac{\lambda^k e^{-\lambda}}{k!}$, is very difficult to visualize and is only of use to developers wanting to implement the Poisson distribution.

Discrete distributions: Commonly encountered discrete distributions include the following (see figure 9.5):

- *Binomial distribution*: for a random variable X,

  1. the process involves a sequence of independent trials,
  2. each trial produces two possible outcomes, e.g., heads/tails,
  3. the probability of either outcome (p, say, for heads) does not change,

  – X counts the number of success (where success might be defined as a head occurring) in n fixed trials.
  The Binomial distribution is completely described by two parameters: $B(n, p)$,
  This process is sometimes described using the analogy of, drawing n objects from a pool containing a finite number of two kinds of object, where the object is placed back in the pool after it has been drawn (this kind of draw is said to be: *with replacement*). The Hypergeometric distribution is the result, if objects are not returned to the pool once they are drawn (this kind of draw is said to be: *without replacement*).
  A distribution supporting more than two discrete values is known as a *Multinomial distribution* (again with a fixed probability of each value occurring). The `XNomial` package provides support for multinomial distributions,

- *Negative Binomial distribution*: this has the same three requirements as the Binomial distribution, but differs in what is counted,

  – X counts the number of trials up to and including the k^{th} success (where success might be defined as a head occurring after a continuous sequence of tails).
  A process that produces values having a Negative Binomial distribution is randomly drawing from a mixture of Poisson distributions, where the mean of the mixture of Poisson distributions has a Gamma distribution,
  This distribution is a generalised version of the Geometric distribution (which is based on the probability of observing the first success on the n^{th} trial).

- *Poisson distribution*: for a random variable X,

  1. the process involves independent events,
  2. only one event can occur at any time,

  – X counts the number of events that occur within a specified time.
  The Poisson distribution is complete described by one parameter (λ, the distribution mean): $P(\lambda)$,

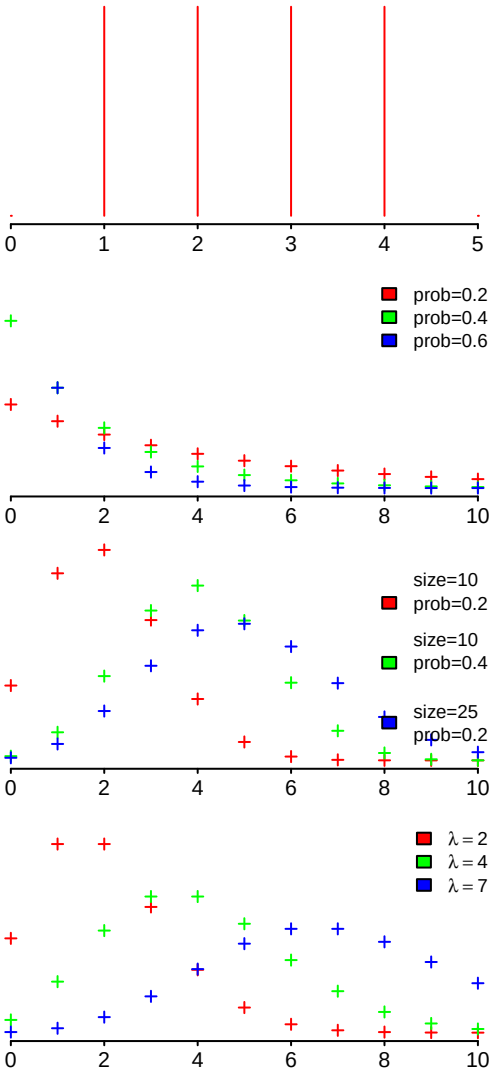


Figure 9.5: Shapes of commonly encountered discrete probability distributions (upper to lower: Uniform, Geometric, Binomial and Poisson). Github–Local

The sum of two independent Poisson distributions $P(\lambda_1)$ and $P(\lambda_2)$ is the Poisson distribution: $P(\lambda_1 + \lambda_2)$.
The Binomial and Poisson distributions are related in that as $n \to \infty$ and $p \to 0$, then $B(n, p) \to P(np)$, i.e., The Poisson distributions is a limit case of a Binomial distribution having a very low probability of success over a long period.

Continuous distributions: Commonly encountered continuous distributions include the following (in all but one case, the generating process clusters the values around a single peak; see figure 9.7):

- *Uniform distribution*: all values between the lower and upper bounds of the interval have an equal probability of occurring, i.e., no value is more likely to occur than any other. For discrete values between 1 and n the probability of any value occurring is $\frac{1}{n}$.

  One process that generates a uniform distribution is a random number generator, such as calling R's `runif` function.

- *Normal distribution*: can be generated by adding together contributions from many independent processes; a consequence of the Central limit theorem. This distribution crops up with great regularity, it has a mathematical form that is easier to manipulate analytically, than many other distributions, resulting in it being widely used before computers reduced the need for analytic solutions to equations. This distribution is described by its mean and variance.

  While the Normal distribution is the result of adding contributions from many independent processes, it is not true to say that adding contributions from many different kinds of processes will result in this distribution (similarly, for multiplicative contributions and a lognormal distribution). For instance, given the right conditions, adding values drawn from many different Poisson distributions can result in a Negative Binomial distribution, a Geometric distribution or other distributions,[976]

- *Lognormal distribution*: the logarithm of a Normal distribution, which can be thought of as being generated by multiplying together the sum of contributions from many independent processes;[1299] samples drawn from a Lognormal distribution can produce a straight line, over some of their range, when plotted using log-log axis,

- *Exponential distribution*: generated by a memoryless process, e.g., the waiting time for an event to occur is independent of the amount of time that has passed since the last event. This is the continuous form of the Geometric distribution, and like it, is described by a single parameter.

  Over some of its range the exponential distribution is visually similar to a power law, which has led researchers to incorrectly claim that their sample fits a power law (a fashionable distribution to have one's sample following); see section 7.1.3. Power laws, and associated scale-free networks are rare in many application domains, but common in a few technological networks.[261]

  The sum of a reasonably large number of independent exponential distributions has an Erlang distribution, e.g., the interval between incoming calls to a telephone exchange, where the interval between calls from any individual have an exponential distribution,

- *Beta distribution*: applies to processes where the explanatory variable is restricted to a finite interval, e.g., the interval zero to one. This distribution is defined by two, non-negative, shape parameters.

- *Gamma distribution* (Γ is the Greek uppercase Gamma, the symbol often used to denote the Gamma function, is the lowercase version, γ): used to describe waiting times, e.g., `Gamma(shape=3, scale=2)` is the distribution of the expected waiting time (in some units) for three events to occur, given that the average waiting time is 2 time units (yes, the `Gamma` function differs from most other distribution names in the base system by starting with an uppercase letter).

  When `shape=1`, the Gamma distribution reduces to the Exponential distribution.

  The Gamma distribution is the continuous equivalent of the Negative binomial distribution.

- *Chi-squared distribution* (sometimes written using χ, the Greek lowercase letter of that name): is more often encountered in the mathematical analysis of statistics, than as a distribution of a sample. A random variable has a chi-squared distribution, with d degrees of freedom, if it is produced by a process which generates the values: $Z_1^2 + Z_2^2 + \cdots + Z_d^2$, where Z_i are independent random variables having a Normal distribution.

  The chi-squared distribution is a special case of the Gamma distribution.

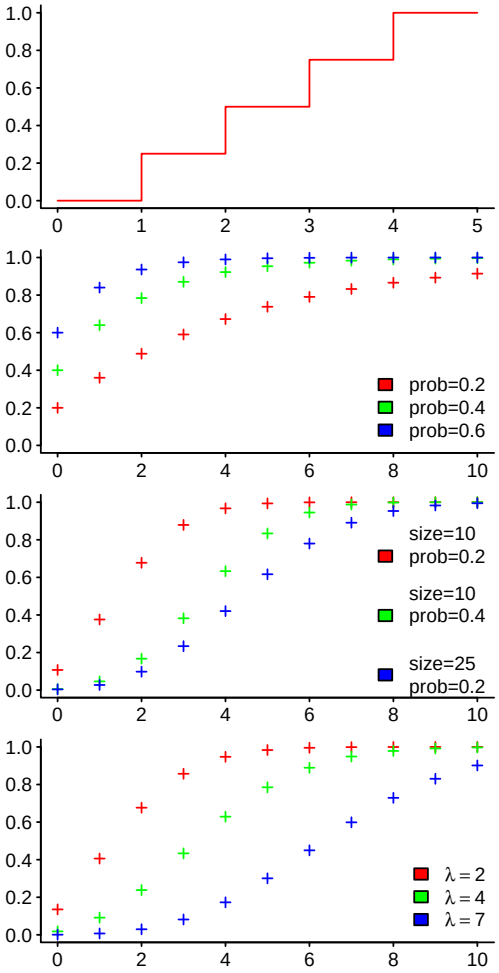


Figure 9.6: Cumulative density plots of the discrete probability distributions in figure 9.5. Github–Local

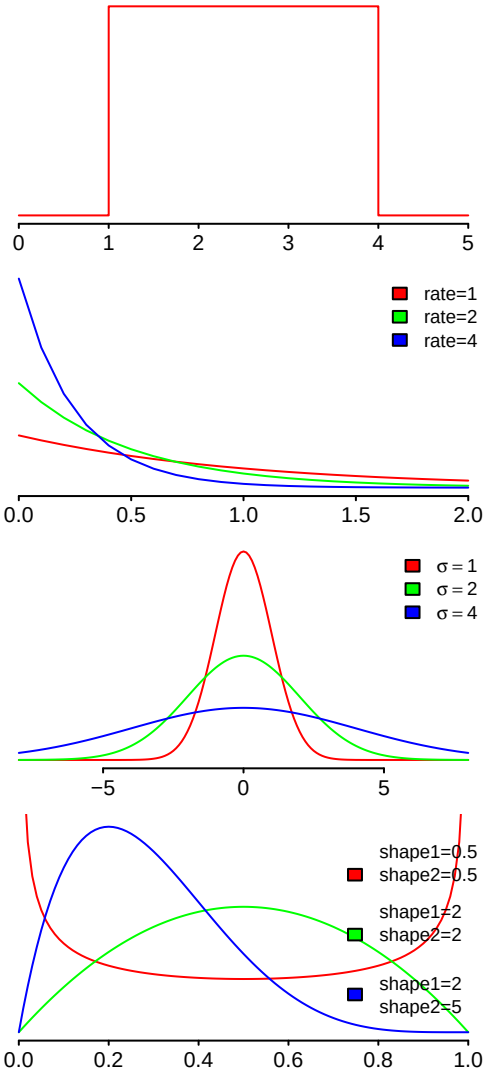


Figure 9.7: Commonly encountered continuous probability distributions (upper to lower: Uniform, Exponential, Normal, beta). Github–Local

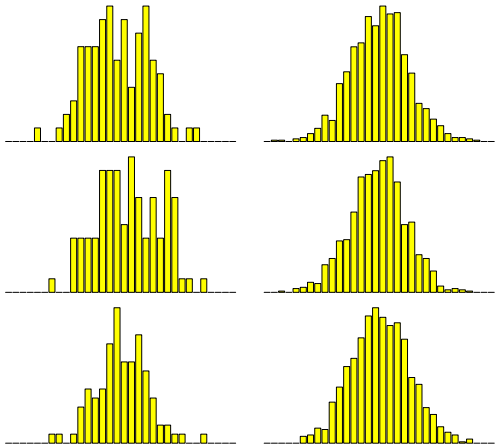


Figure 9.8: Samples of randomly selected values drawn from the same normal distribution (left: 100 points in each sample, right 1,000 points in each sample). Github–Local

- *Weibull distribution*: this distribution drops out as the solution to various problems in hardware reliability, e.g., time to failure, and is often used as the hazard function in survival analysis. The Exponential and Rayleigh distributions are special cases of the Weibull distribution,
- *Cauchy distribution*: this distribution is more famous for its unusual characteristics, e.g., having an undefined mean and variance (because of its very fat tail), than through its uses. The density function for the average of two random variables each having a Cauchy density is a random variable with a Cauchy density; this self mapping is unique to the Cauchy distribution. One consequence is that, if the error in a measurement has a Cauchy density, then the average of many measurements will not be more accurate than the individual measurements.

9.2.1 Are two sample drawn from the same distribution?

As always, visualization is a useful first step in judging whether two samples might be drawn from the same distribution. However, be warned, small datasets can produce visualizations showing little resemblance to the distributions from which they were drawn; as can be seen from figure 9.8, where all the samples are drawn from the same Normal distribution.

A study by Veytsman and Akhmadeeva[1895] measured subject reading rate, in words per minute, for text printed using a Serif or Sans Serif font. Words per minute is a discrete distribution and subject performance is likely cluster around similar values, i.e., there will be duplicates. Figure 9.9 shows a density plot of the normalised data.

The various comparison methods are based on some measure of difference between the *shape* of the sample distributions. The following tests are based on comparing the edf (empirical distribution function) of the samples.

- The Anderson-Darling test is based on the largest difference between the edf of the two distributions, it uses weights to ensure that the tails of the distribution have as much influence as other parts of the distribution; it is possible to use this test to compare more than two distributions. While the Kolmogorov-Smirnov test is often encountered, it has been found to be less sensitive than the Anderson-Darling test[1776] because it primarily detects differences in the main body of the distribution, rather than over the complete range of values.

  The `ad.test` function in the `kSamples` package implements the Anderson-Darling test for two or more samples.

  The `ks.test` function, part of the base system, implements the Kolmogorov-Smirnov test; other implementation include the `ks.test` function in the `dgof` package whose interface is the same but includes support for discrete distributions.

  Samples drawn from a continuous distribution are very unlikely to contain identical values, and many implementations warn if a sample contains duplicate values.
- The Cramér-von Mises test is based on summing (the square of) differences between edfs, rather than using a single maximum value, and can be more powerful against a large class of alternative hypothesis.[75]

  The `cvm.test` function in the `dgof` package implements the Cramér-von Mises test.

The bootstrap can be used to estimate the probability of two sample distributions differing by the amount reported by the statistical test used.

The choice of statistical test depends on whether differences over the range of values in the samples are of interest, whether tail values are uninteresting (perhaps because there are few measurements in the tail, and so what is there is noisey), or the amount of difference between sample distributions is the primary differentiator.

Comparison of samples drawn from discrete distributions is provided by the `WRS` package (on Github), which implements a version of the Kolmogorov-Smirnov test (the `ks` function) that supports discrete data, and also the `bmpmul` function that uses the Brunner-Munzel test (also see the `ks.test` function in the `dgof` package).

The following code shows various tests that check whether two samples are likely to have been drawn from the same distribution (see Github–group-compare/tb104veytsman-dist.R):

```
library("dgof")
library("kSamples")
library("WRS")

# From WRS
ks(serif$Standard_WPM, sansserif$Standard_WPM)
# In fact unscaled measurements give the same result, i.e., not different
ks(serif$WordsPerMinute, sansserif$WordsPerMinute)

dgof::ks.test(serif$Standard_WPM, ecdf(sansserif$Standard_WPM))

# From base system
ks.test(serif$Standard_WPM, sansserif$Standard_WPM)

# Only applicable to continuous distributions
ad.test(serif$Standard_WPM, sansserif$Standard_WPM)
```

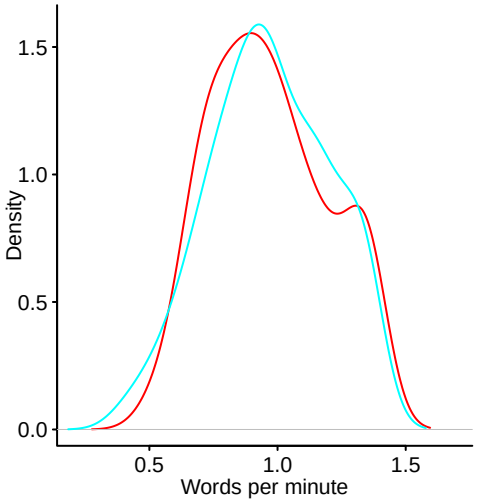


Figure 9.9: Reading rate for text printed using a serif (blue) and sans-serif (red) font, data has been normalised and displayed as a density. Data from Veytsman et al.[1895] Github–Local

The hypothesis that the samples plotted in figure 9.9 are drawn from populations having different distributions is rejected.

Note that many measurement points may be needed to reliably detect a difference in distributions, when one exists. For instance, when one sample is drawn from an Exponential distribution and the other from a Normal distribution, two samples of 150 points are needed to obtain a 95% confidence level, using `ad.test`, that the samples are drawn from different distributions (550 points are needed when the samples are drawn from Normal and Uniform distributions); see Github–group-compare/ad-check.R.

For some analysts, testing whether a sample is drawn from a Normal distribution is a common activity (techniques that are practical to perform manually often require that samples be drawn from this distribution[iii]).

The result of testing whether a small sample is drawn from a Normal distribution has a high degree of uncertainty. The points in figure 9.10 was obtained by testing samples, all drawn from the same distribution (e.g., via a call to `rexp`), using the `shapiro.test` function (replicated 1,000 times for each sample size). The y-axis shows the probability of the Shairo-Wilk test detecting that the sample values are not drawn from a Normal distribution (p-value < 0.05; when the values have been drawn from another distribution); for the case when the values are drawn from a Normal distribution (e.g., a call to `rnorm`) the y-axis gives the probability of this fact not being detected.

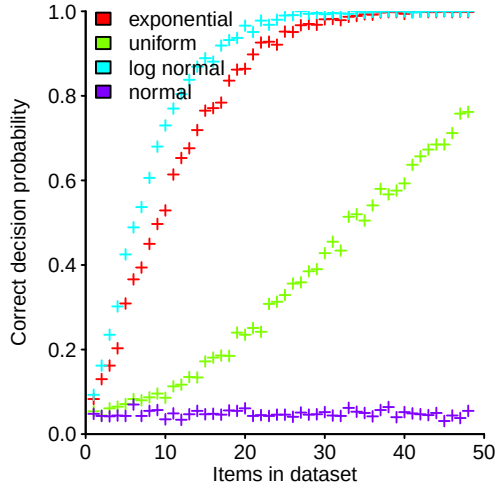


Figure 9.10: Probability, with p-value < 0.05, that `shapiro.test` correctly reports that samples drawn from various distributions are not drawn from a Normal distribution, and probability of an incorrect report when the sample is drawn from a Normal distribution; 1,000 replications for each sample size. Github–Local

There is no guarantee that the values in a sample have a distribution that even closely resembles any known probability distribution.

A study by Berger, She, Czarnecki and Wąsowski[179] investigated the use of feature macros used in the configuration of software product lines. Figure 9.11 shows the number of conditionally compiled sections of source code that were dependent on a given number of feature macros.

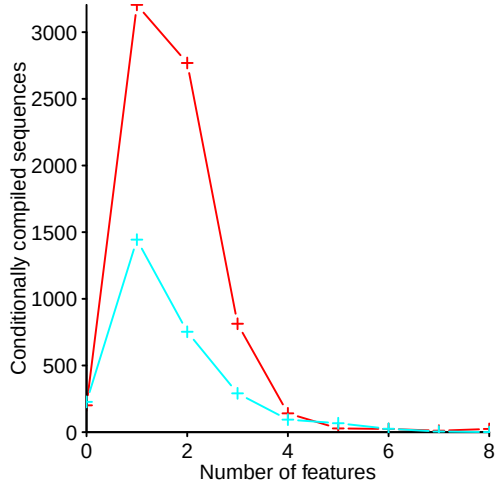


Figure 9.11: Number of conditionally compiled code sequences dependent on a given number of feature macros (red overwritten by blue: Linux, blue: FreeBSD). Data from Berger et al.[179] Github–Local

A Cullen and Frey graph shows that the characteristics of neither sample are close to matching any common discrete distributions. A Kolmogorov-Smirnov test considers them to be sufficiently different, that they are likely to have been drawn from different distributions; see Github–group-compare/cond-compile/2010-berger.R.

Samples may appear to have a similar shape, but have different mean values. Technically, samples with different mean values (or standard deviations) are considered to be drawn from different distributions. There may be theoretical reasons for believing that samples have been generated by the same processes and normalizing mean values (or even variance) enables the shape of the sample distributions to be compared.

A study by Zhu, Whitehead, Sadowski and Song[2026] counted the number of various kinds of statements in a corpus of C, C++ and Java programs (approximately 100 programs, around 10 million lines, for each language). Figure 9.12 shows the distribution of occurrence (expressed as a density on the y-axis) of various statements (expressed as a percentage on the x-axis), over the programs measured; a different color for each language, figure out which is which, before looking at the code.

[iii] Readers of this book learn techniques that don't have this precondition

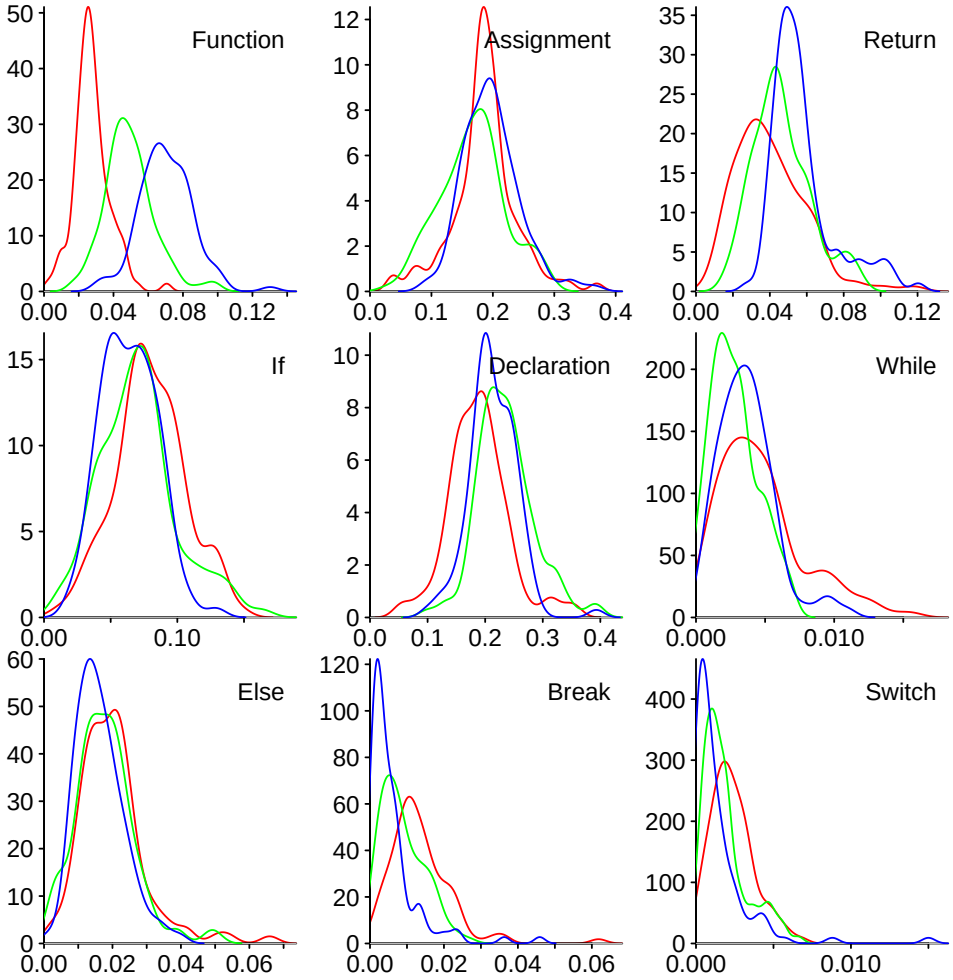


Figure 9.12: Percentage occurrence of statements (x-axis) for each of 100 or so C, C++ and Java programs (colored lines, figure it out or look at the code), plotted as a density on the y-axis. Data from Zhu et al.[2026] Github–Local

Differences in the probability of various kinds of statements being used, over a sample of programs written in various languages, is evidence that language has an impact on what code gets written (either because particular kinds of applications are written using a given language, particular algorithm selection is influenced by language, or the impact of differences in language semantics).

Might two or more of the languages measured be said to have the same distribution of `if-statement` and/or `assignment-statement` usage? The interactions between different statements makes the analysis non-trivial.

The takeaway from this section is that for small sample sizes, distribution comparison produces unreliable answers, and for large samples comparison may be complicated.

Comparison of particular characteristics of sample distributions, e.g., sample means, is discussed in section 10.5.

9.3 Fitting a probability distribution to a sample

Given a sample of values, which of the known, supported by R,[521] probability distributions is the best fit?

There is no universal best-test statistic, for goodness-of-fit of a sample to a probability distribution. The performance of the available tests depends on the (unknown) distribution from which the sample was drawn.[1768]

The Normal distribution is often the default answer given, when people are asked about the distribution of a sample. There are several reasons for this, including: historically many techniques designed to be performed by a human calculator were derived from theory that assumed normally distributed data (which often appeared to work reasonably well, when the data only approximated a Normal distribution), along with a misunderstanding of what the Central Limit theorem is about, driving a belief that a complex process provides the mixing needed to produce a Normal distribution.

As always, knowledge of the processes driving the production of measured values can be very useful. For instance, measurements of arrival times that are driven by a Poisson process will result in inter-arrival times that are exponentially distributed, values created via the multiplicative effect of many contributions may have a Lognormal distribution, and a preferential attachment process often results in links or what they link-to following a power law.

If there is no theoretical justification for a particular distribution, limiting the selection process to those distributions having some degree of name recognition is likely to make the one chosen an easier sell to readers. For instance, the Delaporte distribution[iv] might happen to fit a particular sample slightly better than the Negative Binomial distribution, but its lack of name recognition means that extra effort will have to be invested, justifying its use.

A study by van der Meulen[1876] posted the $3n+1$ problem on a programming competition website: 95,497 solutions were submitted and van der Meulen kindly sent me a copy of these solutions (11,674 solutions were written in Pascal, the rest in C). The $3n+1$ problem is: write a program that takes a list of integers and outputs the *length* of each value, where length is the number of iterations of the following algorithm:

```
for input integer ++pass:[n]++;
   while (n != 1)
      n = (is_even(n) ? n/2 : n*3+1);
```

Which distribution is a good approximation, to the number of lines of code contained in the programs submitted as answers to this problem?

The first step of visualizing the sample provides basic information about the shape of the distribution, e.g., decreasing/increasing, single/multiple peak, symmetric/skewed or appearing to be nothing but random noise; see figure 9.14.

A method of narrowing down the list of possible distributions, is to plot a Cullen and Frey graph. The `descdist` function, in the `fitdistrplus` package, plots this graph and returns some descriptive distribution characteristics of the values (mean, median, sd, skewness and kurtosis). Skew and kurtosis are not reliable estimators and `descdist` includes an option to create and test bootstrap samples.

The blue circle and yellow points in figure 9.13 denote the sample and various bootstrapped results for the $3n+1$ program lengths, assuming a continuous distribution (the average number of lines is large enough that the difference between discrete/continuous is likely to be small). The sample does not overlay any of the grey lines/areas on the plot that denote commonly occurring distributions. The code is:

```
library(fitdistrplus)

# Default is to check continuous distributions
# dummy=descdist(li, discrete=TRUE, boot=500)
dummy=descdist(li, boot=500)
```

The `fitdist` function[v] in the `fitdistrplus` package can be used to fit a distribution to the data, i.e., find values of the specified distribution's parameters, such as mean and variance, that minimise some measure of goodness-of-fit (the AIC of the fit is returned). The `gamlss` package supports a wider range of distributions (see the help information for the `gamlss.family` function) that `fitdist` can use to fit data.

Figure 9.14 shows fits for the Normal, Poisson, Lognormal and Negative binomial distributions.

```
library(fitdistrplus)

tp=fitdist(li, distr="pois"); tnb=fitdist(li, distr="nbinom")
tn=fitdist(li, distr="norm"); tln=fitdist(li, distr="lnorm")

# gofstat is a way of getting all the values used for plotting
theo_vals=gofstat(list(tn, tp, tln, tnb), chisqbreaks=1:120,
                  fitnames=c("Poisson", "Negative binomial",
```

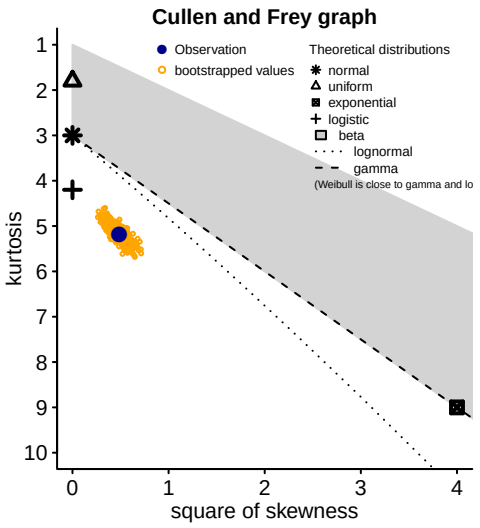


Figure 9.13: A Cullen and Frey graph for the $3n+1$ program length data. Data kindly provided by van der Meulen.[1876] Github–Local

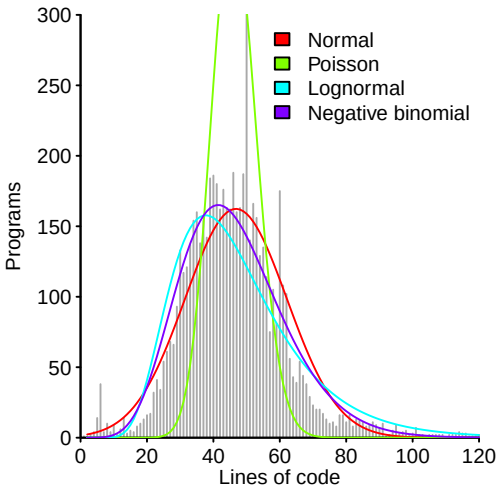


Figure 9.14: Number of 3n+1 programs containing a given number of lines, with four distributions fitted to this data. Data kindly provided by van der Meulen.[1876] Github–Local

[iv] A compound distribution derived from a Poisson distribution whose mean has a shifted Gamma distribution.

[v] The `MASS` package contains the `fitdistr` function and the `gamlss` package contains the `fitDist` function, both of which fit distributions to data.

```
                                        "Normal", "Lognormal"))

plot_distrib=function(dist_num)
{
lines(theo_vals$chisqbreaks, head(theo_vals$chisqtable[, 1+dist_num], -1),
                col=pal_col[dist_num])
}

plot(theo_vals$chisqbreaks, head(theo_vals$chisqtable[, 1], -1), type="h",
             xlab="Program length", ylab="Number of programs\n")
plot_distrib(1); plot_distrib(2)
plot_distrib(3); plot_distrib(4)
```

The large spike at 50 lines might be caused by solutions all doing the same thing, but with different statement orderings, e.g., multiple submissions derived from a common solution.

Based on minimizing AIC, the Normal distribution is the best fit, with the Negative binomial distribution a close second. Should either distribution be chosen as the best fitting, or is it worthwhile attempting to fit other distributions? The answer depends on what the fitted distribution will be used for, e.g., making predictions or building models. Jumping to any conclusions based on one data-point (i.e., set of length measurements for one problem) is always problematic.

9.3.1 Zero-truncated and zero-inflated distributions

Some distributions only make use of non-negative values, they start at zero, e.g., the Poisson distribution. While zero is a common lower bound for measurement values, other lower bounds occur, e.g., the number of minutes to complete a task (the zero time tasks, that are never started, are not measured).

It is possible to adjust the equations that describe zero-based distributions, to have a non-zero lower bound. Rebasing a distribution to start at one (rather than zero) is the common case and after such an adjustment the distribution is said to be *zero-truncated*, e.g., *zero-truncated Poisson distribution*.

The `gamlss.tr` package contains functions that support the creation of zero-truncated (or truncation to the right or left of any value) distribution functions. The following code creates a set of functions relating to the zero-truncated type II Negative binomial distribution; the name of the created function is `NBIItr` and like other distribution functions in R, the associated density, distribution, quantile and random functions are obtained by prefixing the letters `d`, `p`, `q` and `r`, respectively, to `NBIItr`:

```
library(gamlss)
library(gamlss.tr)

gen.trun(par=0, family=NBII) # Bring various functions into existence
```

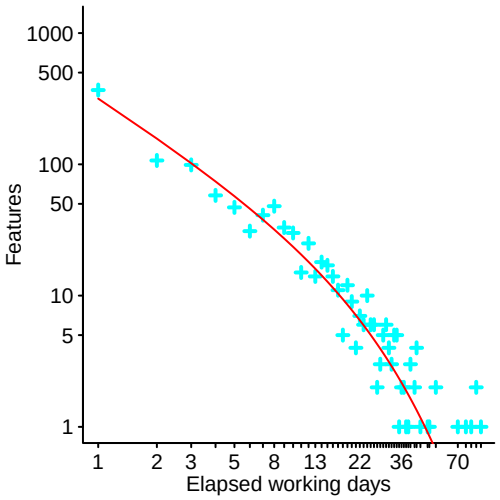


Figure 9.15: A zero-truncated Negative Binomial distribution fitted to the number of features whose implementation took a given number of elapsed workdays; first 650 days used. Data kindly provided by 7digital.[1] Github–Local

The 7Digital data[1] (discussed in more detail in section 5.4.6) contains information on 3,238 features implemented between April 2009 and July 2012; the information consists of three dates (Prioritised/Start Development/Done), from which a non-zero duration can be calculated.

The Cullen and Frey graph suggests a negative binomial distribution might be a good fit.

The functions returned by `gen.trun` do not have a form that can be used in calls to the `fitdist` function. The `gamlss` function in the `gamlss.tr` package has a special form for handling these created functions, as shown in the following code (where `day_list` contains the list of values and `NBIItr` was created by an earlier call to `gen.trun`). The following code was used to produce figure 9.15:

```
library(gamlss)
library(gamlss.tr)

g.NBIItr=gamlss(day.list ~ 1, family=NBIItr)

NBII.mu=exp(coef(g.NBIItr, "mu"))       # get mean coefficient
NBII.sigma=exp(coef(g.NBIItr, "sigma")) # standard deviation

plot(table(day.list), log="xy", type="p", col=point_col,
```

```
                xlab="Elapsed working days", ylab="Features\n")

lines(dNBIItr(1:93, mu=NBII.mu, sigma=NBII.sigma)*length(day.list), col="red")
```

One process generating values having a Negative binomial distribution is based on a mixture of Poisson distributions, whose means have a Gamma distribution. It is possible to generate other distributions by combining a mixture of Poisson distributions, are any of these a better fit to the data? The Delaporte distribution sometimes fits slightly better and sometimes slightly worse; the difference is not large enough to warrant switching from a relatively well-known distribution, to one that is rarely covered in text books or supported in software; if data from other projects is best fitted by a Delaporte distribution, then it may be worthwhile spending time analysing how this distribution might be a better model of project scheduling.

If the processes generating these values can be modeled by a mixture of Poisson distributions, it is unlikely that a single subprocess is responsible for a large percentage of the quantity measured, many subprocesses are involved.

Sometimes count data contain many more zero values than are expected, from the distribution that the generating process is believed to follow. Two kinds of behavior that can cause an excess of zeroes to appear in the measurements are:

- a process that generates zeroes, and a process that generates non-negative values; this situation can be modeled by what is known as *zero-inflated model*. The `gamlss` package supports zero-inflated distributions,
- the measurements involve two processes, one where the values are zero or non-zero, and the other where values are always non-zero (i.e, zero-truncated); this situation can be modeled by what is known as a *hurdle model* (the hurdle that has to be got over is moving from zero to non-zero). The `gamlss` package supports what it calls *zero altered* (or *zero adjusted*) distributions, while the `pscl` package uses the term hurdle.

Your author's search for software engineering measurements containing an excess of zeroes located a few that appeared to contain an excess, but none could be fitted by the models discussed above; see Github–probability/bolz_data_struct_racket.R for an example.

9.3.2 Mixtures of distributions

Sometimes sample measurements are generated by two or more distinct processes, resulting in values that appear to be drawn from two or more distinct distributions, e.g., a plot shows multiple peaks. A model built using a mixture, or weighted sum, of distributions is known as a *finite mixture model* or just a *mixture model*; a continuous mixture of distributions is known as a *compounded distribution* (the Negative Binomial distribution is a compounded distribution).

The `mixtools` and `rebmix` packages contain functions for fitting samples drawn from two or more of the same kind of distribution family, e.g., multiple Normal distributions. The two packages differ in the structure of their API, e.g., one having many functions, and the other having one main function taking many arguments (neither would win a prize for user interface design).

A study by Hunold, Carpen-Amarie and Träf[876] investigated the impact of external factors on the performance of an MPI micro-benchmark. Figure 9.16 shows the runtime variation of two different MPI calls, with each having two distinct peaks. The two peaks in the left curve appear to be symmetrical and perhaps a mixture of two Normal distributions is a good fit. Figure 9.17 shows the two distributions fitted by a call to the `normalmixEM` function (in the `mixtools` package), along with a histogram (all produced by the same call to the `plot` function provided by the package).

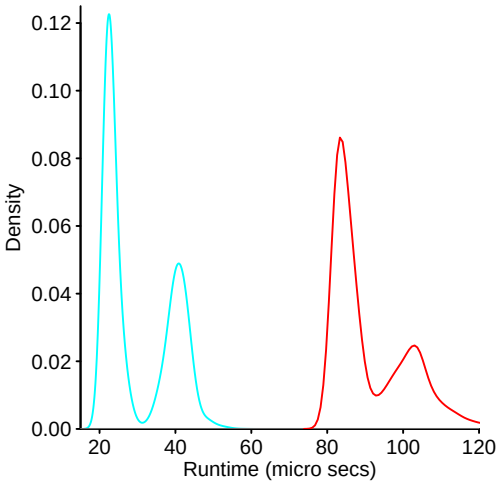


Figure 9.16: Density plot of MPI micro-benchmark runtime performance for calls to `MPI_Allreduce` with 1,000 Bytes (left curve) and to `MPI_Scan` with 10,000 Bytes (right curve). Data kindly supplied by Hunold.[876] Github–Local

```
library("mixtools")

scan_dist=normalmixEM(fig1_Allreduce$time)

plot(scan_dist, whichplots=2, main2="", col2=pal_col,
        xlab2="Time (micro secs)", ylab2="Density\n")
```

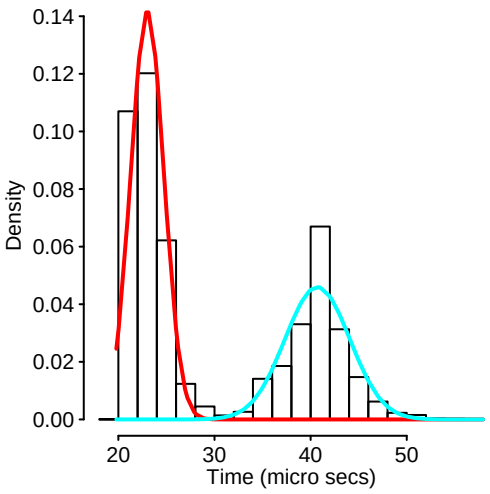


Figure 9.17: Mixture model fitted by the `normalmixEM` function to the performance data from calls to `MPI_Allreduce`. Data kindly supplied by Hunold.[876] Github–Local

A call to `summary` returns the parameters of the fitted model; the first row (prefixed by `lambda`) is the fraction contributed by each distribution, followed by the mean, standard deviation and log likelihood (rather than AIC): Github–Local

```
number of iterations= 17
summary of normalmixEM object:
          comp 1    comp 2
lambda  0.611002  0.388998
mu     23.011364 40.703293
sigma   1.720527  3.378665
loglik at estimate:  -28873.39
```

A plot of a sample drawn from a mixture of distributions does not always have visually distinct peaks; if f_1 and f_2 are normal densities with means μ_1 and μ_2, respectively, and both have the same variance σ^2, then the mixture density $f = 0.5f_1 + 0.5f_2$ will have a single peak if, and only if: $abs(\mu_2 - \mu_1) \leq 2\sigma$.

A study by Kaltenbrunner, Gómez, Moghnieh, Meza, Blat and López[964] analysed the pattern of user activity of the Slashdot technical community news site. The black curve in figure 9.18 shows the density of the number of accesses to one article in each minute after first publication (a total of 1,567 accesses).

A possible explanation for the multiple upticks in number of accesses, is the article being linked to from other websites, driving a fresh batch of readers to Slashdot. Which mixture of distributions might best fit the access times of this Slashdot article? The Poisson distribution is often used to model arrival times and is the obvious first choice, but in this particular case turns out not to provide the best fit.

Figure 9.18 shows several Normal distributions fitted to data, on a log scale, using functions from the `rebmix` and `mixtools` packages. The algorithms used by packages do not guarantee to find the globally optimal solution and differences in the mix of distributions selected can occur because of differences during the search process.

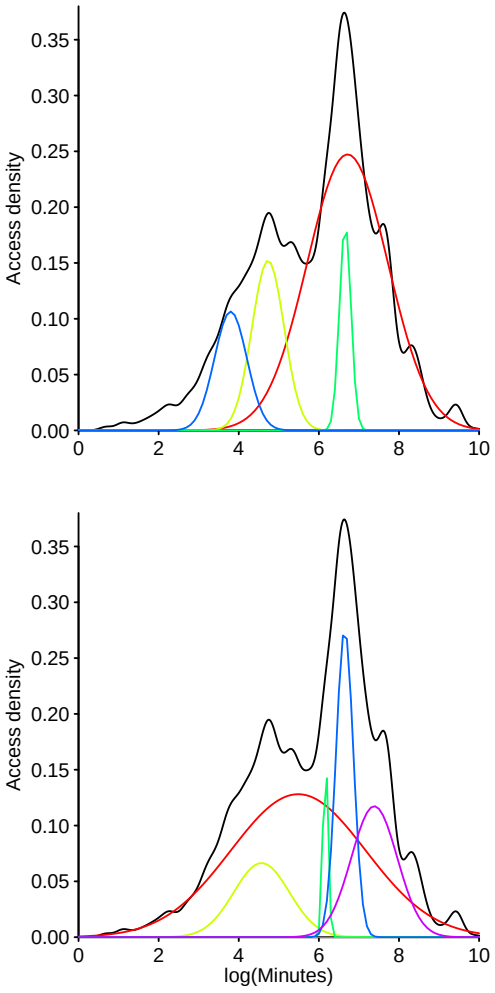


Figure 9.18: Density plots of accesses to one article on Slashdot, in minutes since its publication. The distinct Normal distributions (colored and fitted to the log of the data) contained in the mixture models fitted by the `REBMIX` (upper) and `normalmixEM` (lower) functions. Data kindly supplied by Kaltenbrunner.[964] Github–Local

```
library("rebmix")

slash_mod=REBMIX(Dataset=list(data.frame(users=log(slash$users))),
                 Preprocessing="histogram", cmax=5,
                 Variables="continuous", pdf="normal", K=7:45)

plot_REBMIX_dist=function(dist_num)
{
y_vals=dnorm(x_vals, mean=as.numeric(slash_mod$Theta[[1]][2, dist_num]),
                  sd=as.numeric(slash_mod$Theta[[1]][3, dist_num]))
lines(x_vals, slash_mod$w[[1]][1, dist_num]*y_vals, col=pal_col[dist_num])
}


plot(work_den, main="", xlim=c(0, 10), ylim=c(0, 0.36),
       xlab="", ylab="Access density\n")
plot_REBMIX_dist(1); plot_REBMIX_dist(2)
plot_REBMIX_dist(3); plot_REBMIX_dist(4)
```

Fitting a Normal distribution to log scaled data means that the sample actually has a Lognormal distribution. Is the Lognormal distribution a good representation for the processes driving readers to access Slashdot articles? As always in model building the answer depends on what the model is to be used for. If the purpose is to make predictions, the accuracy of prediction is of more interest than any underlying assumptions; if the purpose is to understand what is going on, then a theory containing processes generating Lognormal distributed behavior is needed.

It can take a lot of analysis, over many years, to settle on the distribution, or combinations of distributions, that best describes the measured properties of a system. The study of file-system characteristics[18] is an example of how researchers' ideas and models changed over time,[507,896,1300] becoming more sophisticated as more data became available, from various platforms,[447] and more analysis was performed.

9.3.3 Heavy/Fat tails

Heavy tailed is the term used to describe distributions where the majority of values occur a long way from the mean value (*fat tails* and *long tail* are also used).[vi] When the 80/20

[vi] The term *sub-exponential* is sometimes used to describe tails that decay slower than exponential and *super-exponential* for tails that decay faster than exponential.

rule applies the distribution is heavy tailed, and the frequency with which this rule is encountered suggests that such data is not rare. The `poweRlaw` package supports operations involving a variety of heavy tailed distributions, including power laws.

Averaging the performance of multiple subjects can produce values that are well fitted by a power law, while individual subject performance is well fitted by any of a variety of other distributions.[1334]

The Pareto distribution is the mathematical name of a particular instance of a heavy tailed distribution (sometimes going by the name *power law* in popular culture); Zipf's law is a particular instance of this distribution.

The mean value of a heavy tailed distribution may not exist (because it is infinite). Any finite dataset has a finite mean, and if a sample is drawn from a heavy tailed distribution, its mean value will jump around erratically.

It is more difficult to narrow down a distribution that best fits a sample drawn from a heavy tailed distribution (because several fit equally well), compared to one without a heavy tail; a sample may contain many values, but their density may be low because values are spread out over a long tail (rather than in a high density cluster around a central location).

Figure 9.19 is from a survey[896] of file sizes and shows that a small percentage of files account for most of the disk spaced occupied (the vertical line meets the bytes line where 89.9% of disk space has yet to be consumed, and the files line where 12.5% of files still remain to be accounted for). Another way of describing the situation is to say that there is a mass/count disparity, i.e., a few files occupy most of the space.

Care needs to be taken to separate out concepts that are popularly associated with power laws, e.g., *scale invariant*, which are a property of the distribution, not the generating process. The process generating data fitted by a power law can be remarkably random, e.g., the length of words in text produced by monkeys typing.[395]

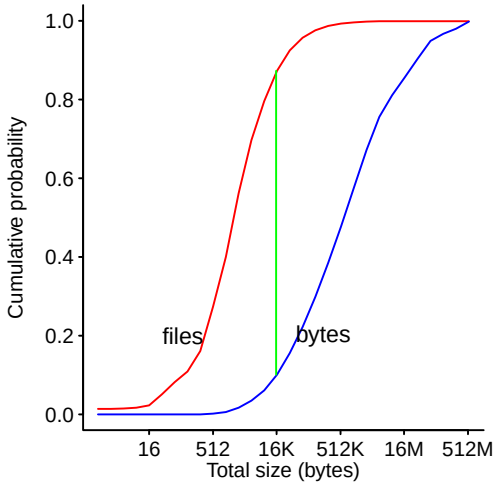


Figure 9.19: Cumulative probability distribution of files size (red) and of number of bytes occupied in a file system (blue). Data from Irlam.[896] Github–Local

9.4 Markov chains

A *finite state machine* (FSM) is a machine represented by a set of distinct states, connected by edges denoting the possible transitions that can occur when a given event occurs, such as when a particular character is input (FSMs are deterministic).

A *Markov chain* (MC) is also a machine represented by a set of distinct states connected by edges, but the possible transition is chosen at random based on the transition probability of each edge (the transition probabilities, out of any state, that is not an absorbing state, add to one); the next state only depends on the current state, i.e., the system is memoryless.

A Markov chain is a *discrete-time Markov chain* (DTMC), if the transition between states occurs at fixed time intervals; if the time interval between state transitions is not fixed, the Markov chain is a *continuous-time Markov chain* (CTMC) (the memoryless requirement means that transition times must have an exponential distribution). If the transition time depends on how long the system has been in the current state, it is a *semi-Markov process* (SMP).

Finite state machines provide a useful abstraction for modeling user interfaces. A study by Oladimeji[1412] investigated the user interface of the Alaris volumetric infusion pump (a medical device used for controlled automatic delivery of fluid medication, or blood transfusion, to patients); the user interface includes 14 buttons and an LCD display. Figure 9.20 shows the available transitions between states.

A FSM can be represented as a control flow graph. By using this representation functions in the `igraph` package can be used to answer questions such as: the maximum number of button presses needed to get to any state (12; the `path.length.hist` function returns a count of all possible path lengths), and the average number of presses to transition between any two states (4; using the `average.path.length` function).

If the behavior of a system (that can be represented using a FSM) is monitored, the probability of occurrence of every transition between states can be calculated. If the behavior represents typical user interaction with the system, then the probabilities can be used to create a Markov chain for this typical behavior.

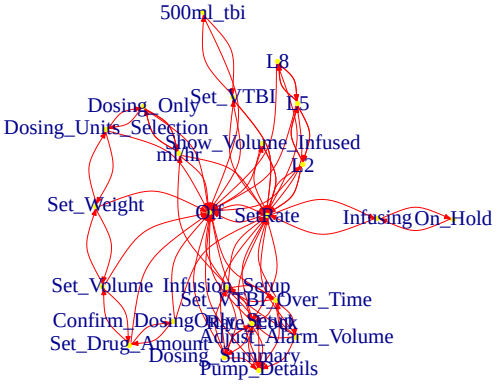


Figure 9.20: Graph of available state transitions for Alaris volumetric infusion pump (the button presses that cause transitions between states are not shown). Data kindly supplied by Oladimeji.[1412] Github–Local

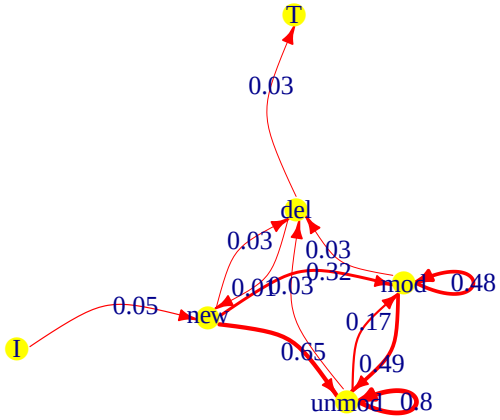


Figure 9.21: Discrete-time Markov chain for created/modified/deleted status of Linux kernel files at each major release from versions 2.6.0 to 2.6.39. Data from Tarasov.[1814]
Github–Local

A study by Tarasov, Mudrankit, Buik, Shilane, Kuenning and Zadok[1814] used data on the lifetime of source files in various systems, such as the Linux kernel, to generate realistic filesystem contents (for deduplication analysis). Figure 9.21 shows a Markov chain representing the life of source files in the Linux kernel (from being Initialised to new, through modified/unmodified to deleted and reaching the Terminal state). The measurement snapshot occurred at each of the 40 releases between versions 2.6.0 and 2.6.39, with an average of 23k files per snapshot; the time between releases is roughly constant, so this might be considered a discrete-time Markov chain.

The `graph.data.frame` function assumes there is a link between the row values in two columns (`from` and `to` vertices) and builds a graph based on this assumption. The `V` and `E` functions access the vertices and edges of the graph and various attributes can be set and may be subsequently used by `plot`.

```
library("igraph")

atc=read.csv(paste0(ESEUR_dir, "probability/atc12-gra.csv.xz"), as.is=TRUE)
atc_gra=graph.data.frame(atc, directed=TRUE)

V(atc_gra)$frame.color=NA
V(atc_gra)$size=12 ; V(atc_gra)$color="yellow"
E(atc_gra)$arrow.size=0.5 ; E(atc_gra)$color="red"
E(atc_gra)$weight=E(atc_gra)$linux
E(atc_gra)$label=E(atc_gra)$weight/100

# layout.lgl outperforms the default layout for this graph
plot(atc_gra, edge.width=0.3*sqrt(E(atc_gra)$weight),
                          edge.curved=TRUE, layout=layout.lgl)
```

The algorithm used by `plot` to layout a graph makes use of randomization, which means that the layout returned by every call is different.

9.4.1 A Markov chain example

A study by Perugupalli[716, 1474] investigated the reliability of gcc, based on the reliability of its major subsystems. Information on the probability of a subsystem experiencing a failure was calculated, using the regression suite for gcc version 3.3.3 (which contains tests for 110 faults present in gcc version 3.2.3, out of 2,126 tests, of which 55 were traced back to the source code of a single subsystem; the others faults involved multiple subsystems). The researchers did not attempt to analyse failures involving more than one subsystem, and assumed that subsystems fail independently of each other.

Subsystems were identified by instrumenting gcc to count the number of calls between pairs of functions, made while executing the regression suite (this is not actually Markov chain-like behavior because the called functions return, which is not transition-like behavior). The 1,759 traced functions were manually assigned to one of 13 internal subsystems (e.g., parsing, tree optimization and register allocation),

The reliability of gcc version 3.2.3 might be estimated using:

$$R = 1 - \frac{F_c}{T_c} = 1 - \frac{110}{2126} = 0.948$$

where: F_c is the number of source files that it did not correctly compile and T_c is the total number of files compiled.[vii]

This approach has the advantage of being simple to calculate, but does not provide any information on the impact of individual subsystems on overall reliability, for instance, what is the sensitivity of overall system reliability to behavioral changes to one subsystem?

The probability of reaching subsystem n from subsystem 1 after k transitions is Q^k (where Q is the matrix of transition probabilities). Summing over all transitions (using an infinite upper bound for the total number of transitions simplifies the mathematics), we get:[1846]

$$S = \sum_{k=0}^{\infty} Q^k = (I - Q)^{-1}$$

[vii] The calculated reliability is very low because it is based on compiling a test suite of short code samples designed to reveal faults.

where: I is the identity matrix. The expression $(I-Q)^{-1}$ is easily calculated (i.e., inverting the result of a matrix subtraction). The matrix S, is known as the *fundamental matrix*, and can be used to calculate a variety of properties of systems modeled by the Markov chain.

The composite and hierarchical methods are two techniques for combining information on subsystem usage (i.e., subsystem transition probabilities and subsystem reliability, calculated using the above formula), to calculate the reliability of a complete system:

- composite method:[350] this calculates the probability of a successful transition between each subsystem, by multiplying the transition probabilities of each subsystem by the probability of the subsystem executing successfully. These individual successful transition probabilities are used to calculate the successful transition probability from the initial subsystem to the final subsystem, i.e., the system's fundamental matrix. The estimated reliability calculated for gcc is 0.9972,[viii]; see Github–reliability/gcc-reliability.R.
- hierarchical method: if R_i is the reliability of a subsystem, the probability of all executions of that subsystem being successful is $R_i^{N_i}$, where N_i is the number of transitions to subsystem i during one execution of the system. Assuming that subsystems fail independently, the expected value of system reliability is:

  $$R = E\left[\prod_{i=1}^{n} R_i^{N_i}\right]$$

  Assuming subsystems are highly reliable, and the variance in the number of subsystem transitions is very small, the first order Taylor approximation can be used:

  $$R \simeq \prod_{i=1}^{n} R_i^{V_i}$$

  where: $V_i = E[N_i]$ is the expected number of times a transition occurs to subsystem i, during a single execution of the complete system; V_i is obtained by solving:

  $$V_i = q_i + \sum_{j=1}^{n} V_j p_{ji}$$

  where: q_i is the probability that execution starts with subsystem i, and the p_{ji} are obtained from the subsystem transition probability matrix; see Github–reliability/gcc-reliability.R.

The `markovchain` package supports discrete time Markov chains, and the `msm` package supports continuous time through the use of multi-state models.

9.5 Social network analysis

The popularity of web based social networks has made the mathematics of social network analysis a fashionable research topic. Unfortunately many published papers involve little more than claiming to have found a power law, with only pretty pictures and hand waving to show. Table 7.1 is an example of the kind of descriptive statistics encountered in social network analysis.

Social networks are represented as graphs, and the `igraph` package supports reading many graph data representation formats, along with a wide range of operations and analysis on graphs.

A study by Canfora, Cerulo, Cimitile and Di Penta[300] analysed the developer's mailing lists for FreeBSD and OpenBSD, to obtain information on what they called *Cross-System-Bug-Fixings*; the data contains information on 861 unique developers sending email and 1,062 unique developers receiving email. Both FreeBSD and OpenBSD were forked from a common base and not only continue to share common code but faults fixed in one are often applied, some time later, to the other. Figure 9.22 was produced using code very similar to that used for the Markov chains in figure 9.21.

Many real world collections of linked node contain subgroups (e.g., clusters of developers or related code modules), and there are a variety of algorithms for detecting these subgroups. Care needs to be exercised in interpreting the clusters returned by these algorithms, as there may be many distinct high-scoring solutions, and a clear global maximum may not exist.[705]

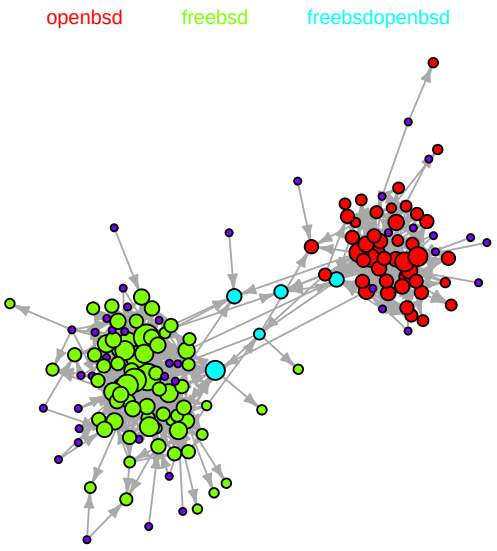


Figure 9.22: Directed graph of emails between FreeBSD and OpenBSD developers, plus a few people involved in both discussions, with developers who sent/received less than four emails removed. Data from Canfora et al.[300] Github–Local

[viii] If the 55 fault count used in this analysis is plugged into the simple formula used above, the reliability estimate is 0.974.

9.6 Combinatorics

The analysis of some systems makes it necessary to consider combinations of various items, and there is a need to enumerate all possible sequences, to find the total number of different sequences of items that could occur (or other related questions). The mathematics used to solve this kind of problem is known as *combinatorics*.

A few of the functions frequently used in combinatorial problem solving are included in R's base system, including:

- the `choose` function takes two arguments, n and k and returns the value $\frac{n!}{k!(n-k)!}$, often written as $\binom{n}{k}$; the number of ways of selecting k items from n items,
- the `combn` function takes two arguments, x and k and returns an array containing all combinations of the elements of x taken k at a time.

When an item is drawn, with replacement, from a pool of items the probability of drawing the same item again is unchanged, when drawing without replacement the probability will decrease by the appropriate amount. An item is distinct if it is treated as being different from all other items in the pool (even when drawing with replacement), e.g., there are four items in the pool `x=c("a", "a", "b", "c")`, but only three of them are distinct.

Table 9.1 show how the `iterpc` function in the `iterpc` package can be used to generate sequences based on the distinctness of items and whether they are drawn with replacement or not.

		Distinct	
		True	**False**
Replacement	**True**	`I=iterpc(5, 2, replace=TRUE)`	`x=c("a", "a", "b", "c")` `I=iterpc(table(x), 2, replace=TRUE)`
	False	`I=iterpc(5, 2)`	`x=c("a", "a", "b", "c")` `I=iterpc(table(x), 2)`

Table 9.1: Example `iterpc` calls generating particular kinds of sequences of length two (by passing the value returned to `getall`, e.g., `getall(I)`).

The treatment of item ordering is another factor, when considering all possible permutations; is the ordering of items significant or not, e.g., are the sequences `a,b` and `b,a` treated as different or equivalent? When the ordering of items is significant calls to `iterpc` need to set the optional argument `ordered` to `TRUE`, e.g., `I=iterpc(5, 2, ordered=TRUE)`.

9.6.1 A combinatorial example

This example illustrates the kind of detailed analysis needed to solve a practical combinatorial problem.

A study by Jones[934] investigated developer preferences for ordering members within C `struct` types. The hypothesis was that members having the same type are likely to be grouped together within the same `struct` type.

The data contains enough instances of `struct` types containing between three and eight members, for the sample to be analysed with a reasonable level of confidence.[ix]

If a `struct` contains n members, the number of possible member sequences is $n!$. However, we are only interested in member types and don't care about permutations of members having the same type. The number of different member type sequences is $\frac{n!}{n_1!n_2!\cdots}$ where $n = n_1 + n_2 + \cdots$ and n_1, n_2, etc are the number of members having a given unique type.

Taking the example of a `struct` containing four members, two of type `x` and two of type `y` the possible sequences of member types within a `struct` type are:

[ix] The number of `struct` types containing a given number of members decreases approximately logarithmicly with increasing number of members,[930] i.e., most member sequences are relatively short.

```
xxyy xyxy yxxy xyyx yxyx yyxx
```

and if two members are of type x, one of type y and one of type z, the possible member type sequences are:

```
xxyz xxzy xyxz xyzx xzxy xzyx yxxz yxzx yzxx zxyx zxxy zyxx
```

In the first case members are grouped together in $\frac{1}{3}$ of cases and in the second, in $\frac{1}{2}$ of cases.

If there are t different types, there are $t!$ possible unique sequences of types. If the ordering of `struct` members is random, the probability of encountering a definition in which all members having the same type are grouped together is: $\frac{t!}{\frac{n!}{n_1!n_2!\cdots n_t!}}$. For the two examples above the probabilities of encountering a member ordering, where identical types are grouped together, are: $\frac{2!}{\frac{4!}{2!2!}}$ and $\frac{3!}{\frac{4!}{2!1!1!}}$ (which is already known from enumerating out all possible sequences).

When a `struct` contains four members, as in the above examples, it is not possible to distinguish between a developer intentionally choosing an order and random selection. For `structs` types containing five members, the probability of random selection of member order, grouping together the same member types is high; see the fifth column in table 9.2.

Total members	Type sequence	structs seen	Grouped occurrences	Random probability	Occurrence probability
4	1 1 2	239	185	0.50	2.83×10^{-18}
4	1 3	185	146	0.50	4.75×10^{-16}
4	2 2	98	61	0.33	4.58×10^{-09}
5	1 1 1 2	57	50	0.40	1.03×10^{-13}
5	1 1 3	94	61	0.30	3.13×10^{-12}
5	1 2 2	86	49	0.20	5.18×10^{-14}

Table 9.2: Various forms of `struct` types containing a given number of members, one possible type grouping, number of actual `struct` types measured, number having grouping, probability that one type will contain this grouping and probability that the number grouped, out of total seen, will be so grouped. Data from Jones.[934] Github–Local

Table 9.2 shows source code measurement counts and calculated probabilities for `struct` types containing four and five members: the column *Total members* lists the number of members in the type, *Type sequence* is a possible grouping of member types for a given number of member types, *structs seen* is the number of measured `structs` containing the given number of members/types, *Grouped occurrences* is the number of measured `structs` having the grouping listed in the first column, *Random probability* is the probability of this grouping occurring randomly in one `struct` declaration containing the given number of members and types, *Occurrence probability* is the probability of *Grouped occurrences* out of *structs seen* occurring, when the probability of a single instance occurring is *Random probability*.

This analysis shows it is not possible to confidently distinguish between random and intentional ordering, for individual `struct` types. However, programs contain many such type definitions, and if we label each one "Yes" or "No", depending on whether their member types are grouped or not, this list of Yes/No labels has a binomial distribution, and the probability of a given number of Yes/No labels occurring through chance can be calculated.

Taking the example of a `struct` containing four members, two of type x and two of type y, the probability of a single random occurrence of this sequence is $\frac{1}{3}$; the sample analyzed contains 98 `struct` types with four members having two distinct types, of which 61 have this sequence of member types; see columns 3 and 4 in table 9.2. The probability of this occurring is calculated using `pbinom(61-1, 98, 1/3, lower.tail=FALSE)`, whose value is `4.58272e-09` (the `lower.tail=FALSE` option is used because we are interested in the probability of seeing 60 or more occurrences).

Figure 9.23 shows the measured percentage of `struct` types whose members are grouped by type (red pluses), and the percentage that would occur with random ordering (blue line). The green line is the 99.9% probability bound, for the likelihood that 100 `structs`,

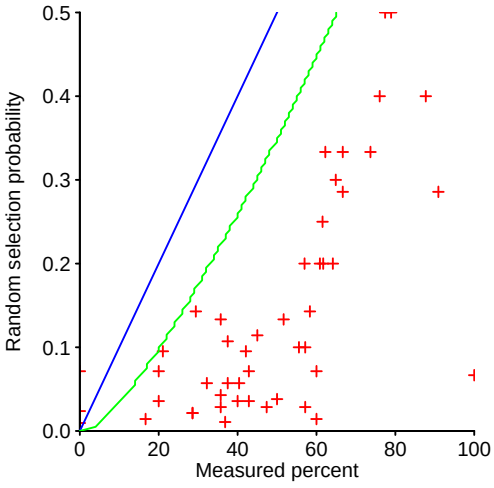


Figure 9.23: Expected probability of a single instance (y-axis) against the probability of a measured **struct** type having grouped member types (x-axis); when both probabilities are the same points will be along the blue line. Data from Jones.[934] Github–Local

all sharing the same member types, will all have their members grouped by type when member ordering is chosen at random. The distance of the red crosses from the 99.9% bound shows that grouping of members by type is very unlikely to have been driven by random selection.

9.6.2 Generating functions

Generating functions are discussed here purely to inform readers about a powerful technique, which is significantly different from the traditional approach to solving probability problems using factorials; this technique is capable of solving problems that appear to be otherwise intractable. If it is not possible to derive an expression specifying how many possibilities can occur in some situation, then a search for the appropriate generating function may provide an answer.

Generating functions are starting to be covered in texts on probability; some mathematical sophistication is required.

A generating function is a polynomial $a_0x^0 + a_1x^1 + \cdots + a_nx^n$, where the coefficients a_n encode information about the quantity of interest.

The following is a simple example that could just as easily be calculated using factorials, but illustrates the idea. How many ways can five items be selected, if `A` can be selected 0 or 1 times, `B` can be selected 0, 1 or 2 times and `C` can be selected 0, 1, 2, 3 or 4 times? The generating function is (see the suggested reading for why this works):

$$(1+x)(1+x+x^2)(1+x+x^2+x^3+x^4) = x^7+3x^6+5x^5+6x^4+6x^3+5x^2+3x+1$$

the coefficient of x^5 is 5, so five different items orderings are possible.

A more complicated example is when items have a particular value and sequences that sum to a specific total are required. If `A` is worth 1, `B` is worth 3 and `C` is worth 5, the generating function is:

$$(1+x+x^2+\cdots)(1+x^3+x^6+\cdots)(1+x^5+x^{10}+\cdots) = 7x^{11}+7x^{10}+6x^9+5x^8+4x^7+4x^6+ \\ 3x^5+2x^4+2x^3+x^2+x+1$$

the coefficient of x^{10} is 7, so there are seven different ways of selecting items that sum to ten.

The `polynom` package supports the symbolic manipulation of polynomials.

Chapter 10

Statistics

10.1 Introduction

Is a pattern of interest present in a population?

Statistics provides information about a population, based on a measurement sample drawn from that population.

The developer input to statistical analysis process is their domain knowledge, which may suggest patterns of behavior to search for, and provide one or more interpretations to any patterns that are found (the feedback given may be that the pattern found is not interesting).

The output from statistical analysis should be treated as a guide, not as a definitive statement.

Correlation does not imply causation, a common mantra that is always worth repeating.

Traditionally, statistical techniques have had to be practical to perform manually. This has resulted in general statistical problems being split into a profusion of specific subproblems, and the creation of techniques tailored to handle each case. Doing statistic analysis this way, involved mapping the sample characteristics to a particular subproblem and then applying the corresponding technique. Computer availability makes it practical to apply general solution techniques and general, powerful and robust statistical techniques are available;[550] however, many existing users of statistical techniques have simply switched from manual to computer based calculation of familiar historical techniques, without appreciating the original design rational for these techniques. Many statistical techniques appearing in this book are impractical to apply manually (e.g., the bootstrap) a computer is required.

The results from data analysis may vary with the person doing the analysis;[1705] for instance, people may use a technique because it is the one they know how to use, rather than the technique best suited to the data being analyzed.

Existing books often invest effort massaging data into a form that permits the use of techniques that depend on the data having a Normal distribution (also known as a _Gaussian distribution_[i]). The reasons for this are historical (assuming Normality made the analysis tractable in the days before computers), and data in the Social sciences (early adopters of statistical techniques and a major market for statistical books) appearing to be drawn from a Normal distribution (despite the claims made, data in this field often does not have a Normal distribution[1274]). It might be said that nobody ever got fired for assuming a Normal distribution.

Measurements of software engineering processes often produce values that are not drawn from a Normal distribution; the Exponential and Poisson distributions are relatively common; measurement samples that are best described by a Normal distribution do occur, but they do not have the dominant market share encountered in other, non-software related, domains, e.g., the social sciences.

The input to statistical analysis is a sample and usually some expectations of behavior; the expectations may be explicit (e.g., measurements are independent of each other) or

[i]This book uses the term Normal because it appears to be more widely used.

implicit, e.g., the choice of a statistical technique that only produces reliable results for samples drawn from a population having certain characteristics.

The possibility for detecting patterns that might be present in a sample depends on the quality and quantity of measurement data:

- quality: noise in the measurement process and errors in post measurement processing (e.g., incorrect conversion of file formats or inaccurate calculations of values derives from the raw data) are some of the problems that affect data quality,
- quantity: the number of measurements impacts the power and significance of statistical tests, and the confidence bounds on the results of statistical analysis.

Finding a pattern in the data having the desired level of statistical certainty, moves the discussion on to the practical engineering consequences of what has been found, e.g., mountain or molehill. A discussion of practical engineering consequences of patterns is outside the scope of this book.

10.1.1 Statistical inference

The most commonly used statistical inference technique makes use of *frequentist* methods, i.e., how often events occur and long-run averages. All techniques have problems associates with their use and frequentist, being the most widely used, has the greatest number of detractors; a common problem is misuse of the concept of p-value; any widely used technique will have a common failure mode, simply because of varying skills of the people using it. The p-value is the fall-guy of the frequentist approach to statistical analysis.

The frequentist approach is the technique predominantly used in this book because it is commonly used in statistical books and articles; it is used by most R packages and readers are likely to encounter it when interacting with other people involved in analysing and using data.

Another technique is *Bayesian statistics*, which is growing in popularity; some R packages use this approach internally. A Bayesian approach has the potential to extract more information from data, by making use of information about prior beliefs. What is known as the *prior*, is a reasonable value for the probability of the event occurring, estimated prior to any measurements being made (the measurements get factored in later); the selection of a suitable prior opens the door to the bias of opinion and policy guidelines,[234] e.g., a Bayesian approach to deciding whether the accused is guilty runs into the problem that many legal systems assume people are innocent until proven guilty (i.e., the prior is zero), a belief that percolates through calculations to always produce a not-guilty result.

A study by Furia[637] reanalyzes several software engineering datasets using Bayesian techniques.

Maximum likelihood estimation, MLE, is a technique for finding the set of parameters for a model that make the observed data most likely to have occurred.

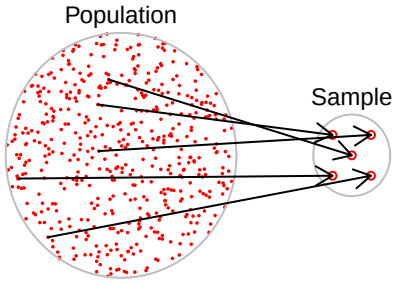


Figure 10.1: Example of a sample drawn from a population. Github–Local

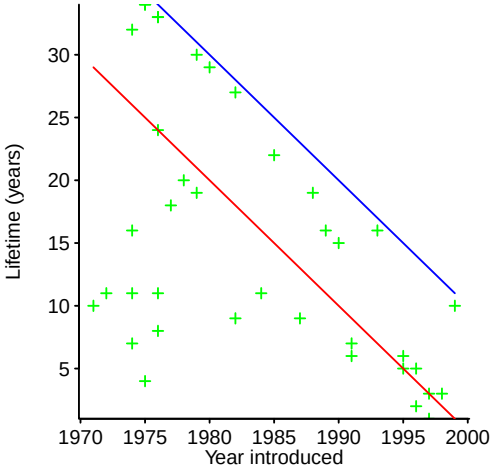


Figure 10.2: Date of introduction of a cpu against its commercial lifetime; processors ceasing production in 2000 or 2010 would appear along one of the lines. Data from Culver.[419] Github–Local

10.2 Samples and populations

It may not be practical to measure every member of a population, and the subset of the population measured is known as a *sample*; see figure 10.1.[ii] Depending on the question being asked, a set of measurements may be a population or a sample. For instance, measurements of one particular program yields the parameters of a population when the questions being asked concern just that one program, but they become the statistics of a sample when generalizing the findings to questions about other programs (including future versions of the one measured).

A sample is selected as a proxy for the entire population; experimental subjects are discussed in section 13.2.1. There are a variety of sampling techniques, including:

[ii]The term *statistic* applies to values calculated from a sample, while the term *parameter* applies to values calculated from a population. In some equations the value $N-1$ is used, when N might appear to be more appropriate. A mathematical distinction occurs between samples and populations, in that sample estimates are based on degrees of freedom of the sample, i.e., the number of members in the sample minus one, while population parameters are based on the number of members in the population, i.e., N.

- a *survey sample* is collected when the items to be measured (often via a questionnaire) are selected from a population assumed to share (unknown) fixed characteristics. The measured characteristics of the random sample, drawn from this fixed population, are used to estimate the characteristics of the population. The analysis of a sample obtained via a survey is *design-based* (rather than *model-based*). See section 13.4 for a discussion of questionnaire based surveys,
- a *prospective* study collects data as events unfold. Figure 10.2 shows the date of introduction of a cpu against its commercial lifetime, in years.[419] Processors that ceased production in 2000 or 2010 would appear along one of the two colored lines,[iii]
- a *retrospective* study collects data after events have taken place,
- a *convenience sample*, as its name implies, makes do with what is available,
- *snowball sampling*, or *chain sampling* starts with an initial list of subjects, who are asked to propose other subjects whom to them, with the process iterating until the number of new subjects falls below some threshold,
- stratified sampling divides the population into what are known as *strata*, with the strata chosen such that similar cases tend to cluster within each one; each of these strata are then sampled (using, say, random sampling) to produce the final sample (which is a set of distinct stratum, see figure 10.3),
- sequential sampling is covered in section 13.2.4,
- interval sampling divides the measurement interval into a series of fixed points and samples at just these points. The width of the sampling intervals puts a lower bound on the behavior that can be resolved. An experimental study[iv] by Kistowski, Block, Beckett, Lange, Arnold and Kounev[1012] measured power consumption, using programs from SPEC's Server Efficiency Rating Tool, at load level increments of 2% (crosses) and 10% (lines); see figure 10.4. A cost/benefit analysis would compare the greater accuracy obtained using finer measurement intervals against the likelihood of sudden jumps in the response value, that could have a noticeable impact on the results.

Occasionally the subjects of interest are not present in the sample. For instance, the damage experienced by aircraft returning from combat, during the second world war, was analysed with a view to improving aircraft survival rate. A statistician involved in the analysis pointed out that important subjects were missing from the sample,[1198] aircraft that had not returned. The return of a damaged aircraft provides evidence that the damaged areas are not critical to survival; it was those areas not damaged in returning aircraft that are likely to be critical to survival.

Guy[760] proposed a *strong law of small numbers*, "There aren't enough small numbers to meet the many demands made of them.", listing 35 examples of numeric patterns found in samples calculated using small integer values that disappear when larger integer values are used. The greater number of large values reduces the likelihood of coincidental correctness.

Figure 10.5 shows the four connected statistical characteristics of a sample; given the values of three of them, the fourth can be calculated.

While gathering a representative sample of the population as a whole is a common requirement, sometimes samples having other characteristics are of interest, e.g., being diverse,[1344] or intended to maximise the number of faults found.[1254]

The algorithm used to select the members of a sample can be non-trivial, even for uniform sampling, e.g., uniform distribution of points within a circle, or uniform sampling from Kconfig feature models.[1407]

10.2.1 Effect-size

Effect-size is the degree to which the characteristic of interest is present in the population (from which a sample is drawn), e.g., if we are interested in the difference in the performance of developers before and after attending a training course, how big is the difference (answering this question may be the reason for obtaining measurements)?

The question to ask about a calculated effect-size is: "Does it matter?" The larger the effect-size the more likely it is to be of interest in practice; in some cases a small effect-size may be of interest (e.g., a small difference multiplied over a large population can

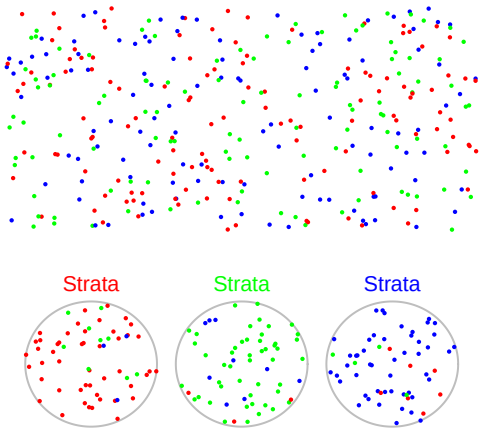


Figure 10.3: A population of items having one of three colors, along with samples of the three strata (imperfect item selection introduces noise in the samples). Github–Local

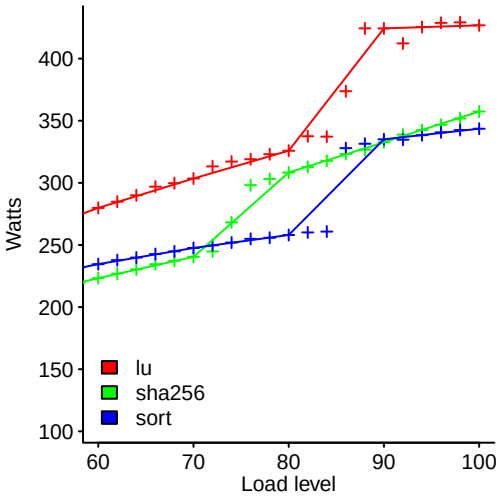


Figure 10.4: Power consumed by three SERT benchmark programs at various levels of system load; crosses at 2% load intervals, lines based on 10% load intervals. Data kindly provided by Kistowski.[1012] Github–Local

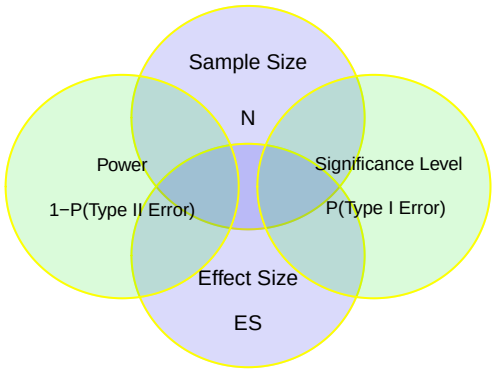


Figure 10.5: The four related quantities in the design of experiments; given three, the fourth can be calculated. Github–Local

[iii] Email discussion with the author confirmed that the data had not been updated since 2010.
[iv] The study was experimental because it did not meet all the requirements for an official SERT run.

have a large impact), while in other cases only a large effect-size is of interest, e.g., when the population is small a large effect-size may be needed to have a large impact.[v]

Smaller effect-sizes are likely to be more costly to detect because more measurements are needed to isolate small effect-sizes compared to larger ones.

Figure 10.6 shows how percentage differences in the presence of a condition in a population can have a dramatic effect on the false positive rate (in red), for the same statistical power and p-value.

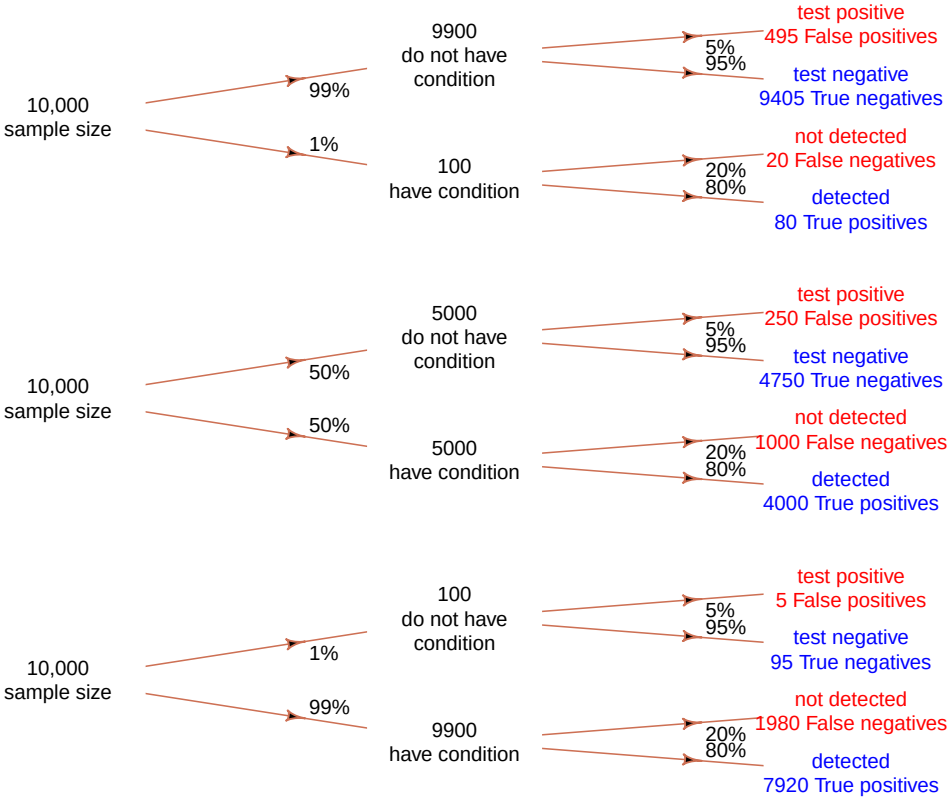


Figure 10.6: Examples of the impact of population prevalence, statistical power and p-value on number of false positives and false negatives. Github–Local

Methods for calculating effect-size depend on the kind of analysis being performed on the sample,[543] and include the following:

- correlation, e.g., the Pearson correlation coefficient, is a measure of effect-size,
- combining information on the mean and standard deviation of two samples into a single value. For instance, Cohen's d is one measure used when samples have similar standard deviations, and is given by: $d = \frac{\mu_1 - \mu_2}{\sigma_{pooled}}$. There are a variety of effect-size calculations associated with Cohen's name.

  Figure 10.7 illustrates how differences in mean and standard deviation, of two distributions, result in a given Cohen's d,

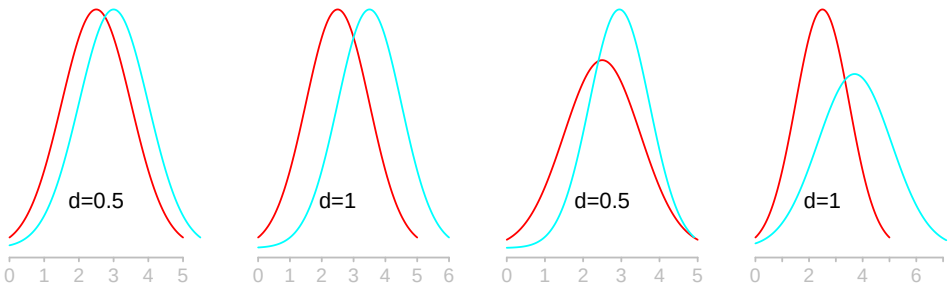


Figure 10.7: Visualization of Cohen's d for two normal distributions having different means and the same standard deviation (two left), and different mean and standard deviations (two right). Github–Local

- odds ratio (i.e., $odds = \frac{p}{1-p}$), that is, the ratio of the odds of an event occurring in one sample divided by the ratio of the same event occurring in the other sample (perhaps a control group).

[v] Statistical books[378] and papers sometimes concern themselves with questions of where to draw the lines that delimit large/medium/small effect-sizes, an approach that might be applicable when researchers are more interested in publishing papers than making useful discoveries.

10.2.2 Sampling error

If the reader agrees that sampling error is an important issue, this section can be skipped. Otherwise, read on and be frightened into agreeing.

The Central Limit theorem is a statement about the mean value of samples drawn from a population. If the population has a finite variance (power laws with an exponent between zero and two have an infinite variance), then the distribution of sample means converges to a Normal distribution as the sample size, N, increases (it does not matter what distribution the population has, it is the distribution of sample means that converges to the Normal).

How quickly does the distribution of sample means converge? The Berry-Esseen theorem gives the best known estimate of convergence of the distribution of the mean of independent, identically distributed, variables to a Normal distribution:

$$|F_n(x) - \Phi(x)| \leq \frac{0.34(\rho + 0.43\sigma^3)}{\sigma^3\sqrt{N}}$$

where: F_n is the cumulative distribution function of the means, Φ the cumulative distribution function of a Normal distribution, ρ the third moment of x (and less than infinity), σ the standard deviation and N the sample size.

The only parameter available for influencing the error is the number of measurements; the error is proportional to: $\frac{1}{\sqrt{N}}$, e.g., to halve the error in the sample mean, the sample size needs to increase by a factor of four.

Figure 10.8 shows the distribution of mean values for samples drawn from three different distributions (using two sample sizes); the vertical lines are 95% confidence bounds.[vi]

A study by Chen, Chen, Guo, Temam, Wu and Hu[344] measured the performance of programs in the SPEC CPU2006 benchmark using 1,000 sets of input data for each program. As an exercise in sampling let's assume we only have access to three of a possible 1,000 input datasets, what range of execution times might we expect to see from processing just three datasets?

Figure 10.9 was obtained by randomly sampling three items from the population of 1,000 and repeating the process 100 times. The red cross is the sample mean, and the vertical brown lines each sample's standard deviation; the blue line is the mean for the population of 1,000 input sets and the green lines the bounds of this population's standard deviation.

Figure 10.10 shows the distribution of sample means for sample sizes of 3 and 12 items. As expected, the larger samples show less variation in the mean value.

Sources of noise (i.e., random variability) in a sample include the following:

- measurement error caused by imperfect tools used to make measurements, which can include coding mistakes and the definition of what is being measured, e.g., lines of code,[1704]
- demographic variability, e.g., measurements of particular kinds of programs, or developers working in a single location or for one company,
- environmental variability is the sea in which developers swim, or have swum in the past, e.g., company culture or habits acquired from early teachers.

Figure 10.11 shows the number of commits to glibc[704] for each day of the week, separated out by year. The plot in the top left shows daily totals over all years. The combined plot suggests that most commits occur near the middle of the week, with the number falling off towards the beginning and end of the week. However, the yearly plots rarely show anything like this pattern; is any interpretation of the pattern of commits in the combined plot a just-so story?

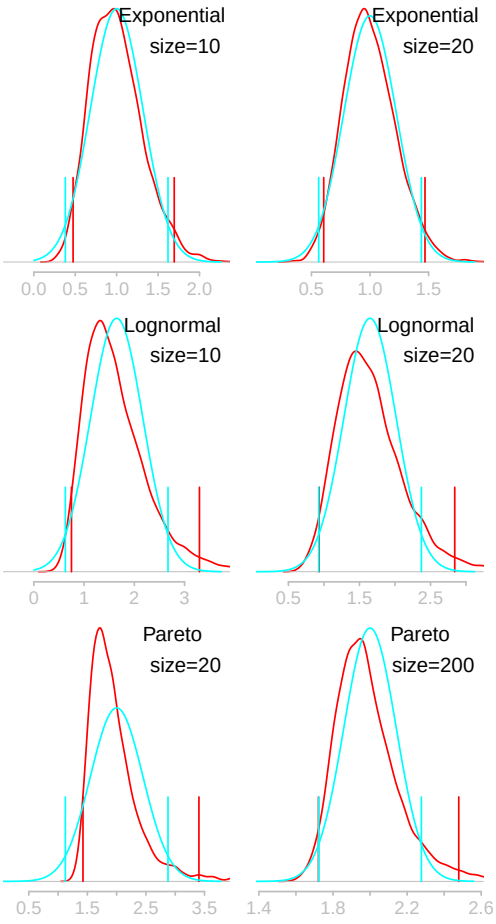


Figure 10.8: Distribution of 4,000 sample means, for two sample sizes, drawn from exponential (upper), lognormal (center) and Pareto (lower) distributions, vertical lines are 95% confidence bounds. The blue curve is the Normal distribution, predicted by theory. Github–Local

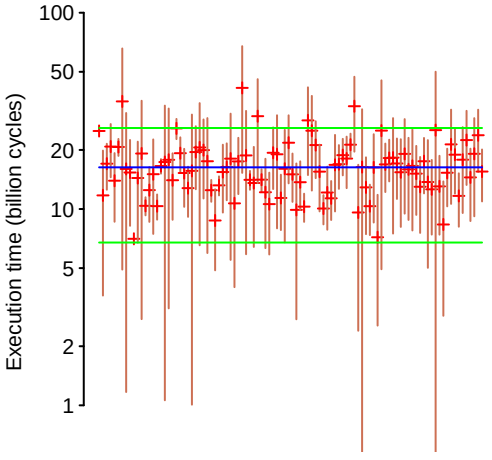


Figure 10.9: Mean (red) and standard deviation (brown line for each sample; not symmetrical because of log scaling) of samples of 3 items drawn from a population of 1,000 items (whose mean shown by blue line and standard deviation by green lines). Data kindly provided by Chen.[344] Github–Local

10.2.3 Statistical power

If an effect exists, and an experiment is performed to measure it, what is the likelihood that the effect will be detected? The numeric answer to this question is known as the *statistical power*, of the experiment. The *power* of a statistical test is its ability to detect a difference when one actually exists in the data. Failing to detect an effect, when one exists, is known as making a *Type II error*, or more commonly as a false negative (β is

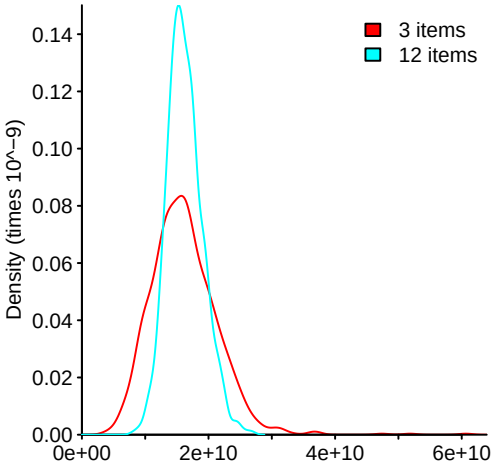


Figure 10.10: Density plot of mean of samples containing 3 or 12 items randomly selected from a data set of 1,000 items; process repeated 1,000 times for each sample size. Data kindly provided by Chen.[344] Github–Local

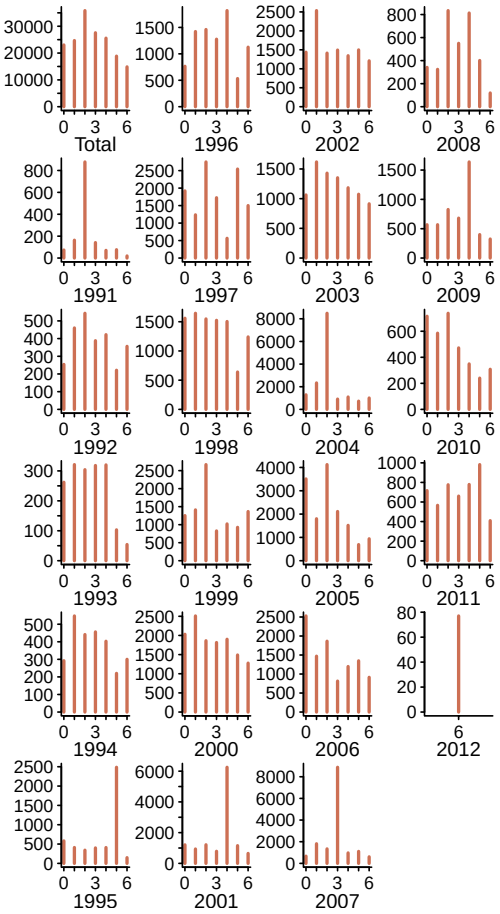


Figure 10.11: Number of commits to glibc for each day of the week, for the years from 1991 to 2012. Data from González-Barahona et al.[704] Github–Local

commonly used to denote the Type II error rate). Techniques for reducing Type II errors include:

- being willing to accept a larger *Type I error*, or more commonly as a true negative (α is commonly used to denote the Type I error rate),
- sampling from a population thought to have a higher probability of containing the sought after characteristics. For instance, Vasa[1883] excluded releases with less than 30 changed classes in a study of class change dynamics. If a subset of a population is selected to maximise detection rate, care must be taken to ensure that any statement of statistical power refers to the subset population, not the larger population from which it was subsetted,
- increasing the number of measurements made, i.e., sample size.

Figure 10.12 is an example showing the distribution of measurements in two populations: X (red) and Y (green), e.g., the time taken to execute all possible programs, with all possible input, on two different computers. The upper and middle plot only differ in mean value, while the middle and lower plot only differ in standard deviation. The false positive rate, α, is shaded in red, and the false negative rate, β, in green. The two rates are connected in that increasing one decreases the other, and vice versa.

When there is a large overlap between samples (middle plot), most of the measurements in either sample have values that suggest they could have drawn from the other sample. In the upper plot, the difference in the sample means makes it more likely that measurements from Y will have values that appear to have been drawn from a different distribution, than samples from X.

The area of the unknown distribution excluding β (i.e., $1-\beta$), is known as the power of the test.

A power of 80% is often quoted[378] as being an acceptable lower limit of a test having a high power, just like 5% is often quoted as an acceptable significance level in many contexts.

If there is a need to estimate whether an effect exists (e.g., one computer is faster than another, or a new algorithm uses less memory), before an experiment is run the question to ask is whether a difference (if it exists) is likely to be detected using the available resources, e.g., time and effort needed to obtain a measurement sample. A statistical power calculation shows the tradeoffs that can be made between sample size and probability of detecting an effect (assuming information on population mean, standard deviation and estimated differences between two or more samples).

The `pwr` package supports power analysis calculations for a variety of standard statistical tests. The functions are passed values for three sample characteristics and return the value of the fourth; see fig 10.5.

A study by Syed, Robinson and Williams[1805] investigated variations in the number of intermittent failures experienced, when using the Firefox browser, at different processor speeds, system memory and hard disc sizes. A total of 11 known coding mistakes, causing intermittent failure (four of these did not produce fault experiences) and nine different hardware configurations were selected. The conditions expected to cause each mistake to result in a fault being experienced were created, and Firefox was executed 10 times with each hardware configuration. Table 10.1 shows the number of each fault experienced with each hardware configuration.

This experiment failed to detect a connection between hardware configuration differences and faults experiences. What is the likelihood that if a connection existed, this experiment would have detected it. Alternatively, how large would the connection need to be for this experiment to detect it?

Analyzing the statistical power of an experiment involving a difference in proportions (i.e., failures before and after) requires an estimate of effect size (calculated from the proportion of failures before and after a change of hardware specification), the number of runs (10 in this case), and the desired p-value; e.g., 0.05. In this study, there were multiple changes of hardware specification and to keep things simple this analysis calculates the power for one change.

Does a hardware change cause more or fewer faults to be experienced? Without a theory providing a believable rationale for more/less, it has to be assumed that either could occur. In other words, a two-sided test is required.

[vi] There will be fluctuations in the values drawn to create each sample.

Mhz-Mb-Gb	124750	380417	410075	396863	494116	264562	332330
667- 128- 2.5	4	10	6	5	2	3	5
667- 256-10	4	8	8	6	4	3	8
667-1000- 2.5	4	7	3	4	3	1	8
1000- 128-10	3	10	3	6	0	1	1
1000- 256- 2.5	3	9	0	6	0	1	2
1000-1000-10	2	9	4	5	0	0	1
2000- 128- 2.5	0	10	5	6	0	0	0
2000- 256-10	2	8	5	7	0	0	0
2000-1000-10	1	7	3	5	0	0	0

Table 10.1: Number of times, out of 10 execution, a known (numbered) coding mistake resulted in a detectable failure of Firefox running on a given hardware configuration (cpu speed-memory-disk size). Data from Syed et al.[1805]

In the following code: `h` is the effect size (the `ES.h` function, from the `pwr` package, calculates this from the estimated proportion of runs that failed before/after the hardware change), `n` is the number of runs, `sig.level` is the p-value significance level and `power` the statistical power. The argument that is not specified (it is not necessary to specify `NULL`, this is the default value), or is given a `NULL` value, is returned by the call.

The default value of the `alternative` parameter is `"two.sided"`.

```
library("pwr")

pwr.2p.test(h=ES.h(before, before+diff), n=num_runs, sig.level=0.05, power=NULL)
pwr.2p.test(h=ES.h(before, before+diff), n=NULL,     sig.level=0.05, power=0.8)
```

Figure 10.13, upper plot, shows the power achieved (y-axis), if a given difference in faults experienced does occur (x-axis), the before proportions 0.05, 0.25 and 0.5 are plotted; the power is plotted for 10 and 50 runs.

The probability of a difference being detected from 10 runs is below 0.5 (i.e., less than 50% chance of detecting a difference at a p-value of 0.05 or better), unless a change of hardware has a large impact on the proportion of faults experienced.

Figure 10.13, lower plot, shows the number of runs needed (y-axis), to have an 80% chance of detecting a given difference (x-axis) in proportion of faults experienced; the before proportions 0.05, 0.25 and 0.5 are plotted, at a significance of 0.05.

This lower plot can be used to find how much difference needs to be experienced for an experiment using 10 runs (per possible fault experience) to be likely to detect it. The failure of this experiment to detect any hardware configuration impact on number of known faults experienced, provides evidence that if any difference does exist, its impact is to add less than 50% or so to the proportion of intermittent fault experiences.

If the `pwr` package does not contain a function that calculates the power of the statistical test being considered, a Monte Carlo simulation can be used to perform a power calculation for the test being considered. The algorithm simulates an experiment, by obtaining samples from the population(s) that are thought to exist and performing the analysis on each sample, counting each success/failure to detect a difference.

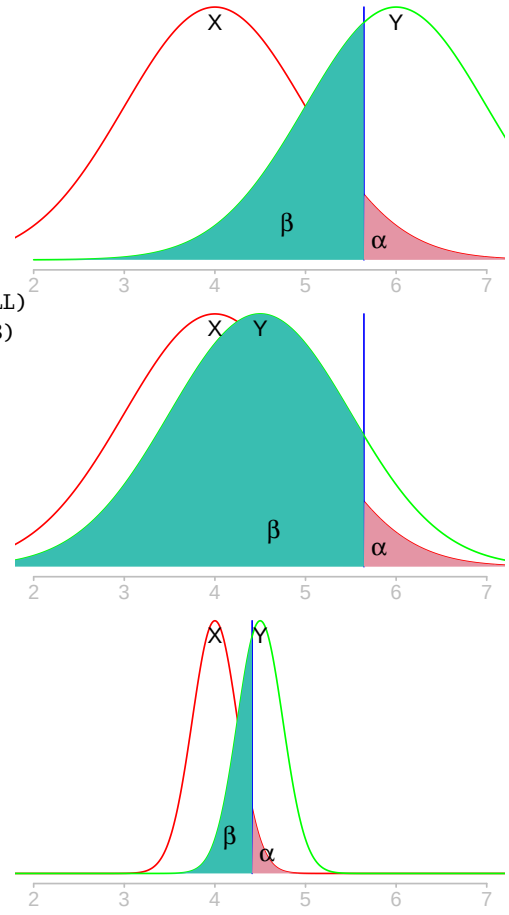


Figure 10.12: The impact of differences in mean and standard deviation on the overlap between two populations (α: probability of making a false positive error, and β: probability of making a false negative error). Github–Local

The following code creates two populations and then compares two samples drawn from these populations. The user written function `some_test_statistic` compares two samples and returns the probability that an analysis of two samples will produce a given value; Github–statistics/boot-power.R contains an example that checks for a difference in mean value between samples drawn from two populations, see figure 10.14:

```
boot_power=function(pop_1, pop_2, sample_size, test_stat, alpha=0.05)
{
num_samples=5000 # Number of times to run the 'experiment'.
results=sapply(1:num_samples, function(X)
                {
                sample_1=sample(pop_1, size=sample_size, replace=TRUE)
                sample_2=sample(pop_2, size=sample_size, replace=TRUE)
                return(test_stat(sample_1, sample_2, alpha))
                })

return(sum(results<alpha)/num_samples) # fraction detected
}
```

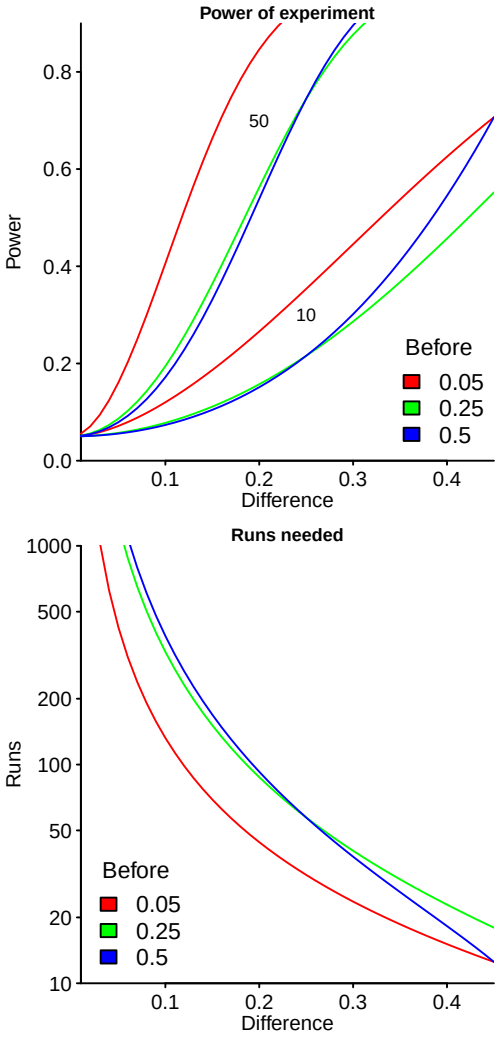


Figure 10.13: Power analysis (50 and 10 runs at various p-values) of detecting a difference between two runs having a binomial distribution (runs needed to achieve power=0.8 at various p-values). Github–Local

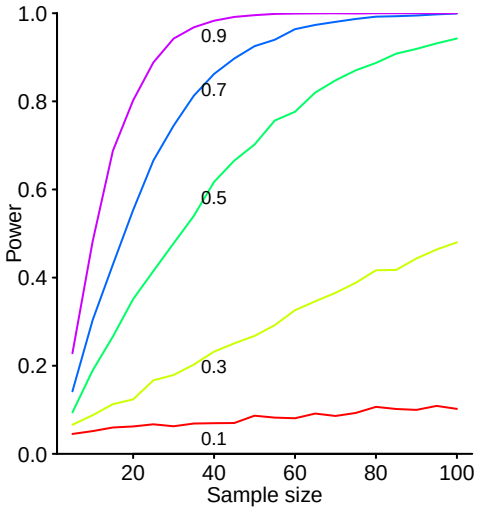


Figure 10.14: The statistical power of detecting that a difference exists between the mean values of samples of various sizes drawn from two populations; actual mean difference between samples adjacent to colored line. Github–Local

```
# Create two slightly different populations (which happen to be Normal here).
population_1=rnorm(100000, mean=0, sd=1)
population_2=rnorm(100000, mean=0+0.5, sd=1)

expected_sample_size=20 # The expected size of the sample to be collected
boot_power=function(population_1, population_2, expected_sample_size,
                            some_test_statistic, alpha=0.05)
```

Figure 10.14 shows the results of a Monte Carlo simulation that tests for a difference in the mean of two samples of various sizes, each drawn from a different population; see Github–statistics/response-power.R for the values calculated using an analytic solution, applicable for populations having a Normal distribution.

Obtaining good enough accuracy from a power analysis requires a good approximation of the likely characteristics of the sample obtained by an experiment. This information about the sample might be extracted from the results of related studies, a preliminary study or theory of the processes involved.

10.3 Describing a sample

A list of values can overwhelm readers with too much detail and techniques for compressing many values into a few, often just one value, are available.[vii] The few compressed values are known as *descriptive statistics*, and the following are some common sample descriptions:

- a point estimate of a central value and its variability, e.g., mean and standard deviation,
- an equation fitted to the sample data according to some condition, e.g., minimising mean squared error,
- quartiles, a cluster of measurements based on where values are relative to other values in the sample, e.g., a box-and-whiskers plot such as fig 8.20.

The mean and standard deviation are the two most commonly used descriptive statistics. It is incorrect to think that two distributions having the same mean and standard deviation will be very similar; see figure 10.15.

10.3.1 A central location

Perhaps the most widely used, single value summary of a sample, derives from the idea of a *middle* or *central* location.

- the mean, is perhaps the most commonly used central location; obtained by adding together the values, in a sample, and dividing by the number of values,
- the *median* is obtained by sorting the N values into numerical order and selecting the value of the $\frac{N+1}{2}$th element (if N is even the average of the middle two values is used),
- the *mode* is the value most likely to be sampled (R's `mode` function is unrelated to the statistical algorithm of that name, it returns the type or storage mode of an object). The `modeest` package contains functions for estimating various kinds of mode.

For symmetric distributions the values of the mean, median and mode are equal, while for asymmetric distributions, the three values can be very different.

When sample values are drawn from a unimodal distribution, the difference between the median and mean is less than or equal to $\sqrt{0.6}\sigma$, and for other non-unimodal distributions less than σ.

The difference between the median and mode is less than or equal to $\sqrt{3}\sigma$.

Unless the sample distribution is symmetric, it is not possible to sum multiple modes, e.g., cost estimates. For nonsymmetric distributions, adding underestimates the true value, e.g., for a Gamma distribution the mean is $k\theta$ and the mode is $(k-1)\theta$, where k and θ describe the Gamma distribution.

Figure 10.16 shows the distribution of execution times of the 1,000 input data sets from Chen et al.[344] If we are interested in an estimate of the execution time of a randomly

[vii] Plotting is a technique that can make use of all the values, and is the major focus of chapter 8.

chosen input data set, the median value, the point that equally divides the number of input data sets is the obvious choice. If we are interested in an estimate of the execution time most likely to be encountered, the value of the mode is the obvious choice.

Some distributions have such fat tails that the mean is infinite, e.g., the Cauchy distribution. In practice, the regularity with which very large values occur results in the mean value of a sample jumping around erratically, as new measurements are made. A distribution that does not have a finite mean may still have a median; the median is not affected by extreme values in the way the mean is, and any extreme values that do appear in a sample do not prevent the median converging to a fixed value.

The median absolute deviation is based around using the median as a robust estimation of variance; supported by the `mad` function.

The well-known algorithms for calculating the mean and standard deviation of a sample require that each value be independent of the others. When a sequence of values is serially correlated, i.e., the value of a measurement is related to the value of one or more immediately previous measurements, the calculated mean and standard deviation is biased. In the case of the mean, the uncertainty in its value grows for positive correlation, and decreases for negative correlation. Figure 10.17, upper plot, shows the fraction of this change for various sample sizes; it is based on an AR(1) model, where each value correlates with the immediately preceding value by an amount given in the legends on the right of the plot; see section 11.10 for a discussion of AR models. A positive correlation causes the ratio of the sample standard deviation, relative to the population standard deviation, to be underestimated, while a negative correlation causes it to be overestimated; figure 10.17, lower plot, shows the fraction of this change.

The `sandwich` package supports the calculation of various error measures that are caused by serial correlation, e.g., the `lrvar` function calculates the error in the long term mean of a series.

Circular data: Some measurements are made using a circular scale, with values that increase and wrap around from the maximum value to start again at the minimum value, e.g., angles take on values between 0 and 360.

The mean, if it exists, has a direction (or angle) and a length; figure 11.81 shows a calculated mean of values drawn from a circular distribution; see section 11.12.

Compositional data: The individual components of a sample of compositional data (i.e., data whose components always sum to a fixed value, such as percentages summing to 100%) are correlated, and the mean of each component cannot be calculated independently of the other components. The `mean` function in the `compositions` package calculates the mean of compositional data; see section 10.3.6.

Several methods of calculating the variance and standard deviation of compositional data have been proposed. The `compositions` package supports the `mvar` function, which calculates what is known as the *total variance* (or *generalized variance*), and the `msd` function which calculates the *metric standard deviation* (both return single values). The variation matrix includes information about the relationship between every pair of components, and is returned by the `variation` function; see Github–statistics/composite-variation.R for the variance calculation of the values plotted in figure 5.31.

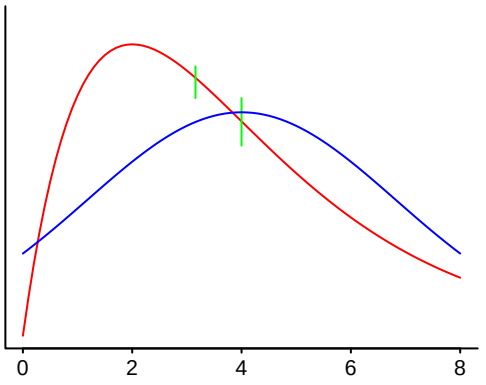


Figure 10.15: A Normal distribution with mean=4 and variance=8 and a Chi-squared distribution with four degrees of freedom having the same mean and variance (the vertical lines are at the distributions' median value). Github–Local

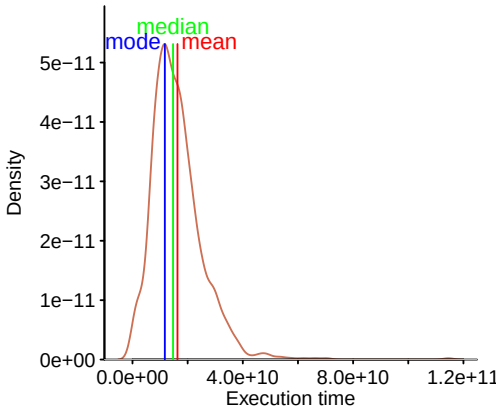


Figure 10.16: Density plot of execution time of 1,000 input data sets, with lines marking the mean, median and mode. Data kindly supplied by Chen.[344] Github–Local

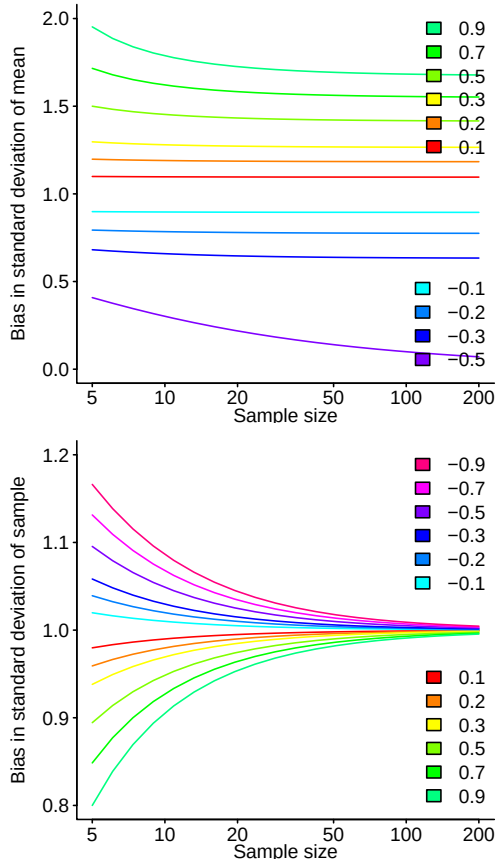


Figure 10.17: Impact of serial correlation, AR(1) in this example, on the calculated mean (upper) and standard deviation (lower) of a sample (the legends specify the amount of serial correlation). Github–Local

10.3.2 Sensitivity of central location algorithms

Samples sometimes contain values that are noticeably different from the other values (e.g., much smaller or larger; which may or may not be the result of noise); the terms *outlier* or *influential observation* are used for such values. The percentage of sample values needed to cause a statistical estimator to produce an arbitrarily large (positive or negative) value is known as the *breakdown point*.

The breakdown point for the mean is proportional to $\frac{1}{N}$, i.e., no matter how many observations are made, it only takes one extreme value to produce a completely spurious result for the mean; the mean has the smallest breakdown point it is possible to have.

At the other end of the scale, the median has a breakdown point of 0.5 (i.e., half of the measurements can have extreme value without affecting the result value) and for this reason the median is often recommended, over the mean, when measurements values are known to be very noisy. However, the median cannot be recommended for universal use because there are situations where it does not perform as well as the mean. For

instance, when values are drawn from a discrete distribution whose mean is roughly halfway between measurable points, and the sample includes duplicate values, then most samples will have a median value slightly larger/smaller than the actual mean, i.e., the median is not evenly distributed across possible values in the way the mean is likely to be distributed; see figure 10.18.

The probability of an outlier occurring depends on the reliability of the measurement process and the characteristics of the population being sampled. The following two techniques are robust in the presence of extreme values in a sample:

- *trimmed mean* removes a percentage of the largest and smallest values, before calculating the mean of the remaining values (it has been found that 20% is a good value for general use). The `mean` function includes a `trim` argument for specifying the percentage to be trimmed,
- *winsorized mean* replaces rather than remove values. The values of the lowest X% are replaced with the lowest value that is just not within the specified percentage, and the values of the highest X% are replaced with the highest value just not within this percentage; the Winsorized mean is calculated using the updated list of values. The `psych` package contains functions that calculate various quantities using the Winsorized mean.

The trimmed and winsorized means may produce biased results when applied to samples drawn from a population having an asymmetric distribution.

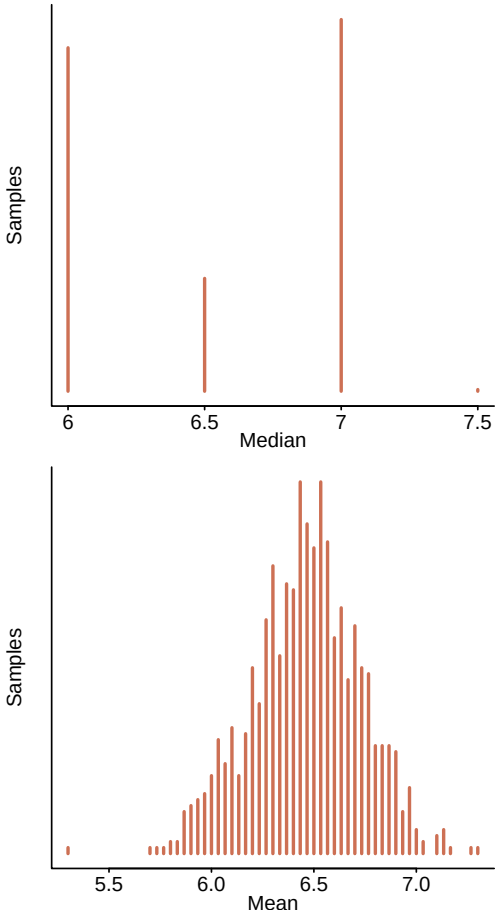


Figure 10.18: Number of sample median (upper) and mean (lower) values for 1,000 samples drawn from a binomial distribution. Github–Local

10.3.3 Geometric mean

The *geometric mean* of N values is:

$$Mean_g = \left(\prod_{i=1}^{N} X_i\right)^{\frac{1}{N}}$$

For instance, the geometric mean of 10, 100, 1000 is $(10 \times 100 \times 1000)^{\frac{1}{3}} \rightarrow 100$.

The geometric mean is preferred to the arithmetic mean when ranking ratios or normalised data (which is a kind of ratio), because it gives consistent results.

When one or more values, X_i, is negative or zero, calculating a geometric mean is a more complicated process.[761]

Consider the (invented) benchmark performance of the three systems in table 10.2. Treating a as the base performance, what is the relative performance improvement of b and c?

If the arithmetic mean is used, the performance ranking of b and c, relative to a, depends on whether the calculation used is a ratio of their means, or the mean of their ratios. The fourth column lists the mean of the values in the second and third column of the corresponding row, and the fifth column lists the ratio of these mean values (relative to a). The individual benchmark ratios for a and b are: $\frac{2}{1}$ and $\frac{105}{100}$, and for a and c: $\frac{3}{1}$ and $\frac{103}{100}$. The mean of these ratios is listed in the sixth column. Comparing columns five and six shows that the ranking of b and c depends on the method of calculating the ratios; also see table 13.1.

system	integer	float	arithmetic mean	ratio of means	mean of ratios	geometric mean
a	1	100	50.5			10
b	2	105	53.5	$\frac{53.5}{50.5} \rightarrow 1.0594$	`mean(2/1+105/100) -> 3.05`	14.49
c	3	103	53	$\frac{53}{50.5} \rightarrow 1.0495$	`mean(3/1+103/100) -> 4.03`	17.58

Table 10.2: Example integer/float benchmark performance measurements of three systems and various methods of calculating relative performance. The relative performance of b and c depends on which mean is used.

If the geometric mean is used, the relative order of the final ratio is not order dependent.

Sometimes the arithmetic and geometric means produce the same benchmark rankings, e.g., a benchmark[560] of eight Intel IA32 processors used the arithmetic mean of ratios

to compare results, the results from using the geometric means was not large enough to affect the relative ranking of processors for a given performance characteristic; see Github–benchmark/powervperfasplos2011.R.

The Geometric mean might be used when values cover several orders of magnitude, e.g., a geometric or logarithmic series (such as: 2, 4, 8, 16, 32, 64)

Methods for calculating the geometric mean include the expression `exp(mean(log(x)))`, and the `geometric.mean` function in the `psych` package.

10.3.4 Harmonic mean

The *harmonic mean* is used to find the "average" of a list of ratios or proportions; it is defined as:

$$Mean_h = \frac{N}{\sum_{i=1}^{N} \frac{1}{X_i}}$$

for instance, the harmonic mean of 1, 2, 3, 4, 5 is: $\frac{5}{\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}} \rightarrow 2.189781$

When there are two values the formula becomes:

$Mean_h = 2\frac{x \cdot y}{x+y}$, which has the same form as the F_1 score, or F-measure, used in information retrieval to combine the precision and recall:

$$F_1 = (1+\beta^2)\frac{precision \cdot recall}{\beta^2 precision + recall}$$

Two methods of calculating the harmonic mean are: `1/mean(1/x)`, and the `harmonic.mean` function in the `psych` package.

10.3.5 Contaminated distributions

A common assumption that often goes unquestioned, is that a sample, or the error present in the measurements it contains, is best described by a Normal distribution. Textbooks are filled with techniques that only exhibit the cited desirable attributes, when the Normality assumption holds. There is also the lure of analytic solutions to a problem, which again may only apply when the Normality assumption holds.

Even when sample values appear to be drawn from a Normal distribution, a small percentage of contaminated values can have a dramatic effect on the value returned by a statistical algorithm.

The Contaminated Normal distribution is a mixture of values drawn from two Normal distributions, both having the same mean, but with 10% of the values drawn from a distribution whose standard deviation is five times greater than the other. Figure 10.19 shows the kernel density of two samples, one containing 10,000 values drawn from a Normal distribution and the other containing 10,000 values from a Contaminated Normal distribution; visually they seem very similar; see Github–statistics/contam-norm.R to learn the color used to plot each sample.

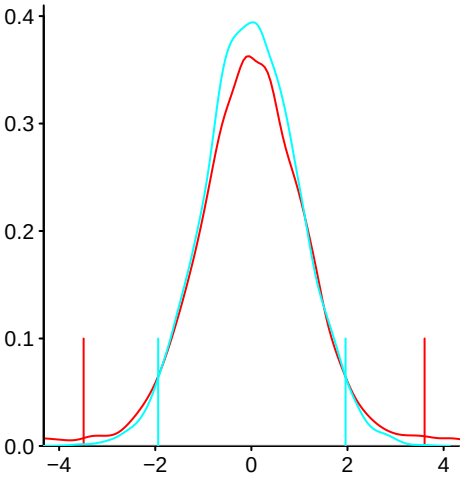


Figure 10.19: Density plot of two samples; samples either drawn from a Normal distribution or a Contaminated Normal distribution (i.e., values drawn from two normal distributions, with 10% of values drawn from a distribution having a standard deviation five times greater than the other); the lines bounding the 95% quartile identify the color used for each plot. Github–Local

This Contaminated Normal distribution has a standard deviation that is more than three times greater than the Normal distribution from which 90% of the values are drawn. This illustrates that a Normal distribution contamination by just 10% of values from another distribution can appear to be Normal, but have very different descriptive statistics.

A number of tests are available for estimating whether sample values have been drawn from a Normal distribution. The Shapiro-Wilk test (the `shapiro.test` function), the Kolmogorov-Smirnov Test (the `ks.test` function)[viii], and the Anderson-Darling test are common encountered. A comparison of four normality tests[1565] found the Shapiro-Wilk test to be the most powerful normality test; see fig 9.10.

When a data set contains very few values, even the Shapiro-Wilk test may fail to determine (e.g., p-value < 0.05) that sample values are not drawn from a Normal distribution. In the case of the Contaminated Normal distribution, samples containing only 10 values are considered to have a Normal distribution in around 30% of cases (i.e., p-value > 0.05), with the percentage dropping to 10% for samples containing 20 values.

Wilcox[1958] provides an analysis of potential problems that outliers, skewed distributions, and fat tails can cause.

[viii] Both included in R's base system.

10.3.6 Compositional data

Compositional data is an aggregate of components, each contributing a portion of the total, i.e., ideally summing to 1 or 100%. The requirement of a fixed total creates a correlation between the components, i.e., if one of component increases, one or more of the others has to decrease correspondingly. Failure to take this correlation between variables into account can analysis results having surprising characteristics, e.g., being unrealistic.

The theory needed to underpin techniques for handling compositional data is all very new, and many issues are still unresolved.

The `compositions` package supports the analysis of compositional data, and offers four approaches to the analysis of data (based on the geometries of the sample space). The compositional mapping functions are:

- `aplus`: the total amount matters, but amounts are compared relatively, e.g., the difference between 1 and 2 is treated the same as the difference between 100 and 200,
- `rplus`: the total amount matters, and amounts are compared absolutely, e.g., the difference between 1 and 2 is treated the same as the difference between 100 and 101,
- `acomp`: the total amount is constant, but amounts are compared relatively, e.g., the difference between 1 and 2 is treated the same as the difference between 100 and 200,
- `rcomp`: the total amount is constant, and amounts are compared absolutely, e.g., the difference between 1 and 2 is treated the same as the difference between 100 and 101.

Figure 5.31 shows the proportion of development time spent in the design, coding and testing phases of 39 applications. Which compositional mapping function is appropriate for this data? The measurements are hours spent in each phase, and from a project duration perspective a time difference of 1-hour is an absolute difference; see Github–statistics/composite-variation.R. When the percentage of total time spent in each phase is of interest, the `rcomp` function applies; when the absolute time is of interest, the `rplus` function applies.

```
library("compositions")

percent_phase=rcomp(est, parts=c("Design_Phase", "Code_Phase", "Test_Phase"))
hours_phase=rplus(est, parts=c("Design_Phase", "Code_Phase", "Test_Phase"))

mean(percent_phase)
mean(hours_phase)
```

The `dist` function can be used to calculate a distance between two compositional values. One use for a distance value is using the bootstrap to estimate the likelihood of a given difference between two mean values; see Github–projects/composite-mean-diff.R and section 10.5.2.

10.3.7 Meta-Analysis

Meta-analysis is the process of combining quantitative evidence from multiple studies to create more accurate estimates of the characteristics studied.

If descriptive statistics of each sample is the only information available, the mean and standard deviation can be pooled (creating a weighted single, combined value). The calculation is as follows (it assumes that each sample is independent of other samples; at the time of writing, no built-in functions are provided in R's base system):

```
pooled_mean=function(df)
{
return(sum(df$s_n*df$s_mean)/sum(df$s_mean))
}

pooled_sd=function(df)
{
return(sqrt(sum(df$s_sd^2*(df$s_n-1))/sum(df$s_n-1)))
}

studies=data.frame(s_n=c(5, 10, 20),
                   s_mean=c(30, 31, 32),
                   s_sd=c(5, 4, 3))
```

```
pooled_mean(studies)
pooled_sd(studies)
```

Medical and social science experiments often measure one or more characteristics of a system before/after an event, e.g., a drug or social program. Various meta-analysis techniques have been created to deal with this kind of before/after study; the `meta` package contains support for this analysis. In software engineering, replicating studies of this kind is not (yet) a common occurrence.

A study by Sabherwal, Jeyaraj and Chowa[1622] performed a meta-analysis of studies of the determinants of success of information systems projects, based on 612 findings from 121 studies published between 1980 and 2004.

The *file drawer problem* is the situation where the results from a study fail to reach the level of statistical significance needed for the work to be accepted for publication, i.e., a meta-analysis may be biased because the published results do not include studies with poor statistical significance (these results are sitting a file draws).[595]

A study by Bem[170] investigated *premonition*, i.e., a persons' ability to predict future events. In nine experiments subjects were asked to guess which of several stimuli would be randomly selected, after their response has been recorded. Experiment 1 involved 50 men and 50 women, who saw a screen containing two images of a curtain and were asked to select one of the curtain images. After a subject selected one curtain image, a picture of either a brick wall or of something else was revealed; the something else picture was either explicitly erotic or neutral. Each subject completed 36 trials. The sequencing of pictures, and their left/right position was randomly selected.

The results found that 53% of subjects selected the curtain image revealing an erotic image at a rate greater than chance (i.e., 50%); subject success rate for the neutral image was 47% (no significant subject sex difference was found). While bootstrap test shows that neither of these percentages occur less than 5% of the time (see Github–statistics/FeelingFuture.R), the binomial test used by the paper's author found a statistically significant difference (the statistical analysis performed was as good as, or better, than that seen in most software engineering papers).

One solution to the file drawn problem is to preregister studies. Here, before collecting any data, researchers submit a description of the study and the data analysis techniques they plan to use; this information is kept confidential until the study is completed. Preregistration reduces the ability of researchers to engage in data dredging. One study[1062] found significant differences in 12 of the 15 meta-analysis studies analysed, compared using only published papers and then including preregistered studies.

10.4 Statistical error

The outputs from applying a statistical technique generally includes probabilities, and it is the responsibility of the person doing the analysis to decide the cut-off probability below/above which an event is considered to have/have not occurred.

The two kinds of statistical error that can be made are:

- treating a hypothesis as true when it is actually false; the statistical term is making a *Type I* error, but *false positive* is more commonly used, and expressed in mathematics: $P(\textit{Type I error}) = P(\textit{Reject } H_0 | H_0 \textit{ true})$,
- treating a hypothesis as false when it is actually true; the statistical term is making a *Type II* error, but *false negative* is more commonly used, and expressed in mathematics: $P(\textit{Type II error}) = P(\textit{Do not reject } H_0 | H_A \textit{ true})$, where H_A is an alternative hypothesis.

		Decision made	
		Reject H	Fail to reject H
Actual	H true	Type I error	Correct
	H false	Correct	Type II error

Table 10.3: The four states available in hypothesis testing and their outcomes.

The practical consequences of a statistical error depend on who is affected by the outcome of the decision made. For instance, consider the consequences of a manager's decision

on whether to invest more time and money testing the reliability of a software system. An incorrect decision can result in more than losing the original investment (e.g., losing market share to a competitor); the bearer of any loss depends on the actual situation and the decision made, as table 10.4 illustrates:

		Decision made	
		Finish testing	Do more testing
Actual	More testing needed	Customer loss	Ok
	Testing is sufficient	Ok	Vendor loss

Table 10.4: Finish/do more testing decision and outcome based on who incurs any loss.

10.4.1 Hypothesis testing

A hypothesis is an unverified explanation of why something is the way it is. Hypothesis testing is the process of collecting and evaluating evidence that may, or may not, be consistent with the hypothesis, i.e., positive and negative testing.[ix] Once enough evidence consistent with the hypothesis has been collected, people may feel confident enough to start referring to it as a theory or law.[453]

The most commonly used statistical hypothesis testing technique is based on what is known as the *null hypothesis*,[x] which works as follows:

- a hypothesis, H, having testable prediction(s) is stated,
- an experiment to test the prediction(s) is performed, producing data D,
- assuming the hypothesis is true, the probability of obtaining the data produced by the experiment is calculated. The calculation made is: $P(D|H)$; that is the probability of obtaining the data D, assuming that the hypothesis H is true.

  If the calculated probability is less than or equal to some prespecified value, the hypothesis is rejected, otherwise it is said that *the null hypothesis has not been rejected*, i.e., the result of the experiment is not conclusive evidence that the null hypothesis is true.

Expressed in code, the null hypothesis testing algorithm is as follows:

```
void null_hypothesis_test(void *result_data, float p_value)
{
// H is set by reality, only accessed by running experiments
if (probability_of_seeing_data_when_H_true(result_data) < p_value ||
    !H)
   printf("Willing to assume that H is false\n");
else
   printf("H might be true\n");
}

null_hypothesis_test(run_experiment(), 0.05);
```

A test statistic is said to be *statistically significant*, when it allows the null hypothesis to be rejected. The phrase "statistically significant" is often shortened to just "significant", a word whose common usage meaning is very different from its statistical one; this shortened usage is likely to be misconstrued when the audience is unaware that the statistical definition is being used, and treating the word as-if it is being used in its everyday meaning sense.

Statistical significance does not mean the pattern found by the analysis has any practical significance, i.e., the magnitude of the pattern detected may be so small as to make it useless for practical applications.

Running one experiment that produces a (statistically) surprisingly high/low p-value is a step in the process of increasing peoples' confidence that a hypothesis is true/false.

Replication of the results (i.e., repeating the experiment and obtaining similar measurements) provides evidence that the first experiment was not a chance effect; another boost

[ix] Gigerenzer[677] discusses how people make decisions in an uncertain environment.

[x] As the market leader in hypothesis testing techniques, over many decades, this technique attracts regular criticism.[379] The criticism is invariably founded on widespread misuse of the null hypothesis ritual; misuse is the fate of all widely used techniques.

in confidence. Replication by others, who independently set up and run an experiment, is the ideal replication (it reduces the possibility that unknown effects specific to a person or group influenced the outcome); an even larger boost in confidence.

There is a great deal of confusion surrounding how the results from a null hypothesis test should be interpreted. Studies have found[679] that people (incorrectly) think that one or more of the following statements apply:

- *Replication fallacy*: The level of significance measures the confidence that the results of an experiment would be repeatable under the conditions described. This is equivalent to saying: $P(D|H) == 1 - P(D)$, and would apply if the hypothesis was indeed true,
- the significance level represents the probability of the null hypothesis being true. This is equivalent to saying: $P(D|H) == P(H|D)$.

The Bayesian approach to hypothesis testing is growing in popularity and works as follows:

- two hypotheses, H_1 and H_2, having testable prediction(s) are stated (the second hypothesis may just be that H_1 is false),
- a non-zero probability is stated for the hypotheses being true, $P(H_1)$ and $P(H_2)$, known as the *prior* probabilities,
- an experiment to test the prediction(s) is performed (producing data D),
- the previously estimated probabilities, that H_1 and H_2 are true, is updated. The calculation uses Bayes theorem, which for H_1 is:

$$P(H_1|D) = \frac{P(H_1)P(D|H_1)}{P(H_1)P(D|H_1) + P(H_2)P(D|H_2)}$$

The updated prior probability, on the basis of the experimental data, is known as the *posterior probability* of the hypothesis being true.

10.4.2 p-value

In a randomized experiment, the *p-value* is the probability that random variation alone produces a test statistic as extreme, or more extreme, than the one observed.

The p-value for each coefficient of a fitted regression model (the subject of chapter 11) is a test of the hypothesis that the coefficient is zero, i.e., there is no association. When the actual value of a coefficient is close to zero, the reported p-value may be spurious. One solution is to rotate the axes, which will have the effect of increasing the value of the coefficient and removing this artefact from the p-value calculation (for this data).

In a commercial environment, the choice of p-value should be treated as an input parameter to a risk assessment comparing the costs and benefits of all envisioned possibilities.

In many social sciences, the probability of the null hypothesis being rejected is required to be less than 0.05 (i.e., 5%, or slightly less than 2σ),[xi] for a result to be considered worth publishing, while in civil engineering, a paper describing a new building technique that created structures having a 1-in-20 chance of collapsing would not be considered acceptable. High energy physics requires a p-value below $5\sigma \rightarrow 5.7 \cdot 10^{-7}$, for the discovery of a new particle to be accepted.

As sample size increases, p-values will always become smaller. For instance, if some aspect of flipping a coin very slightly favours heads, given enough coin flips a sufficiently small p-value, for the hypothesis that the coin is not a fair one, will be obtained. There is no procedure for adjusting p-values for hypothesis testing using very large amounts of data.

When lots of measurement data, covering many variables, is available it is possible to go on a fishing expedition, looking for relationships between variables.[1592] The probability of finding one significant result, when comparing n pairs of variables, using a p-value of 0.05, is $1 - (1 - 0.05)^n$ (which is 0.4, when $n = 10$). When multiple comparisons are made, the base p-value needs to be adjusted to take account of the increased probability of noise being treated as a signal.

Perhaps the most common technique is the *Bonferroni correction*, which divides the base p-value by the number of tests performed. In the above example, the base p-value would

[xi] Journals with high impact factors can be more choosy, and some specify a p-value of 0.01.

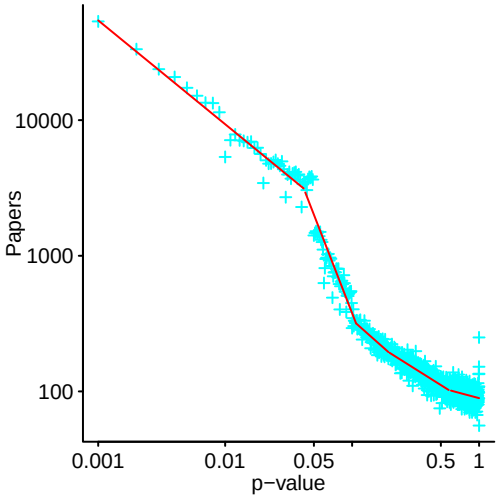


Figure 10.20: Number of papers reporting a p-value equal to a given value; lines are a fitted segmented regression model (four segments were specified). Data from Head et al.[795] Github–Local

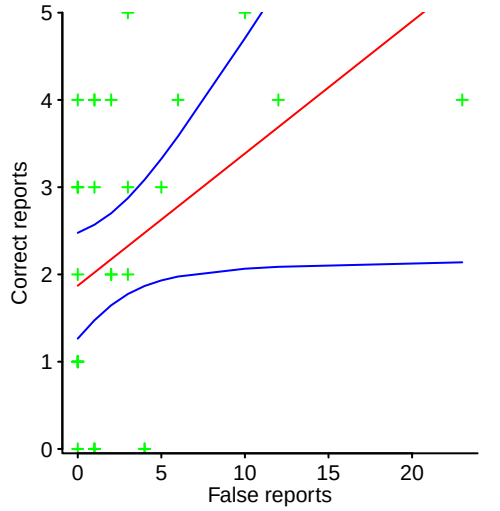


Figure 10.21: Regression model (red line; pvalue=0.02) fitted to the number of correct/false security code review reports made by 30 professionals; blue lines are 95% confidence intervals. Data from Edmundson et al.[531] Github–Local

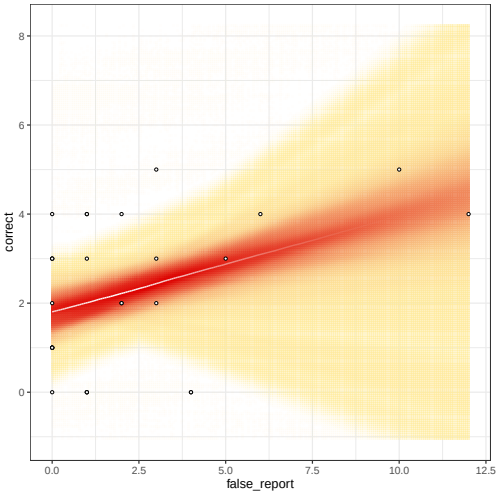


Figure 10.22: Bootstrapped regression lines fitted to random samples of the number of correct/false security code review reports made by 30 professionals. Data from Edmundson et al.[531] Github–Local

be adjusted from 0.05 to $\frac{0.05}{10} \rightarrow 0.005$, to account for the possibility of each of the ten tests matching.

The `p.adjust` function supports p-value adjustment using a variety of different techniques.

Researchers know their work only has a chance of being accepted for publication, if the reported results have p-values below a journal's cut-off value. Given the use of published paper counts as a measure of academic performance, there is an incentive for researchers to run many slightly different experiments[2005] to find a combination that produces a sufficiently low p-value, that the work can be written up and submitted for publication[951] (a process known as *p-hacking*[1216]).[xii] One consequence of only publishing papers containing studies achieving a minimum p-value, is that many results are likely to be false (while a theoretical analysis suggests most are false,[894] an empirical analysis suggests around 14% of false positives for medical research[910]).

A study by Head, Holman, Lanfear, Kahn and Jennions[795] investigated the distribution of p-values appearing in the results section of Open Access papers in the PubMed database. Figure 10.20 shows the number of papers reporting a p-value equal to a given value, with fitted segmented regression model (four segments were specified, but the segment boundaries were selected by the fitting process).

10.4.3 Confidence intervals

Many statistical techniques return a single number, a point value. What makes this number so special, would a value close to this number be almost as good an answer? If an extra measurement was added to the sample, how likely is it that the original number would dramatically change; what if one measurement were excluded from the sample, how much would that change the answer?

A *confidence interval* is an upper and lower bound on the numeric point value(s) returned by statistical technique. A common choice is the 95% confidence bound, the default value used by many R packages.

Numeric confidence intervals can be mapped into visual form by adding them to a plot. Figure 10.21 illustrates how confidence intervals provide an easier to digest insight into the uncertainty of a fitted regression model, compared to the single number that is the p-value. The red line shows a fitted regression model, whose predictor has a p-value of 0.02; the 95% confidence intervals in blue, showing how wide a range of lines could be said to fit the sample almost as well, i.e., any straight line bounded by the blue lines.

A confidence interval is a random variable, it depends on the sample drawn. If many 95% confidence intervals are obtained (one from each of many samples), the true fitted model is expected to be included in this set of intervals 95% of the time (it is a common mistake to think that the confidence interval of one sample has this property). The probability that the next sample will be within the 95% confidence interval of the current sample, for a Normal distribution, is 84% or around 5 out of 6.[420]

A closed form formula for calculating confidence intervals is only known for a few cases, e.g., the mean of samples drawn from a Normal distribution; for a Binomial distribution a variety of different approximations have been proposed.[1490]

Built-in support for calculating confidence intervals, in R packages, is sporadic. Monte Carlo simulation can be used to calculate a confidence interval from the sample, e.g., the bootstrap. This approach has the advantage that it is not necessary to assume that sample values are drawn from any particular distribution. Figure 10.22 was created by fitting many models, via bootstrapping, and using color to indicate density of fitted regression lines.

10.4.4 The bootstrap

The bootstrap is a general technique for answering questions about uncertainties in the estimate of a statistic calculated from a sample, e.g., calculating a confidence interval or standard error.[822] Bootstrap techniques operate on a sample drawn from a population, and cannot extract information about the population that is not contained in the sample,

[xii] Which commercial company would not be willing to add warts to their software to keep an important customer happy?

e.g., if the population contains reds and greens, and a sample only contains reds, then the bootstrap will not provide any information about the greens.

The term *bootstrapping* denotes the process by which a computer starts itself from an off-state. In statistics, it is used to denote a process where new samples are created from an existing sample; the term *resampling* is sometimes used.

The bootstrap procedure often starts by assuming there is no difference, in some characteristic, between samples; it then calculates the likelihood of two samples having the characteristic they are measured to have. The assumption of no difference requires that the items in both samples be *exchangeable*. Deciding which items, if any, in a sample are exchangeable is a crucial aspect of using the bootstrap to answer questions about samples.

Individual time series measurements contain serial correlations. The block bootstrap is one technique for applying bootstrap techniques to time series data. The `tsboot` function, in the `boot` package, supports the bootstrapping of time series data.

Estimating the confidence interval for the mean value of a sample is a good example of the basic bootstrap algorithm; the steps involved are as follows:

- create a sample by randomly drawing items from the original sample. Usually the items are selected with replacement, i.e., an item can be selected multiple times. When items are selected without replacement (i.e., can only be selected once), the term *jacknife* is used,
- calculate the mean value of the created sample,
- iterate the create/calculate cycle, say, 5,000 times,
- analyze the 5,000 mean values, to obtain the lowest and highest 2.5%. The 95% confidence interval for the mean of the original sample is calculated from this lowest/highest band (several algorithms, giving slightly different answers, are available).

The `boot` package supports common bootstrap operations, including the `boot.ci` function for obtaining a confidence interval from a bootstrap sample.

The distribution of the sample from which the bootstrap algorithm draws values is known as the *empirical distribution*.

The *bootstrap distribution* contains m^n possible samples, when sampling with replacement from m possible items to create samples containing n items; when the order of items does not matter, there are $\binom{2m-1}{n}$ possible samples (a much smaller number).

The same bootstrap procedure can be applied to obtain confidence intervals on a wide range of metrics. Figure 10.23 shows confidence intervals for the kernel density plotted in figure 8.14, and was produced by the `sm.density` function, in the `sm` package, using the following code:

```
library("sm")

res_sample=sample(cint$Result, size=1000) # generate 1000 samples

sm.density(res_sample, h=4, col=point_col, display="se", rugplot=FALSE,
        ylim=c(0, 0.03),
        xlab="SPECint Result", ylab="Density\n")
```

The importance of using the appropriate sample size, when using the bootstrap, is illustrated by the analysis of the data from a study by Davis, Moyer, Kazerouni and Lee,[444] which investigated the use of regular expressions in eight languages; the sample size varied between languages. The regex library provided by each language supports different matching functionality, and to handle this the researchers mapped regexs found in each language's source code to a common representation. This mapping makes it possible to assumes that regexs in their common representation form are interchangeable.

Table 10.5 shows, for each language, the mean length of regular expressions, sample size, and the bootstrap probability that the mean observed is less than the bootstrapped means. While Rust has the longest regex mean length, its sample size is relatively small, and the bootstrap finds that it is not possible to rule out that possibility that the mean length observed is not unusual, i.e., 8.8% of generated samples had a mean greater than 39.9. Javascript and Java have, respectively, the second and third longest mean lengths, and their larger sample sizes reduces the uncertainty in the expected mean length; a mean regex length as large as the ones seen is very unlikely to be encountered, i.e., none appeared in the generated samples.

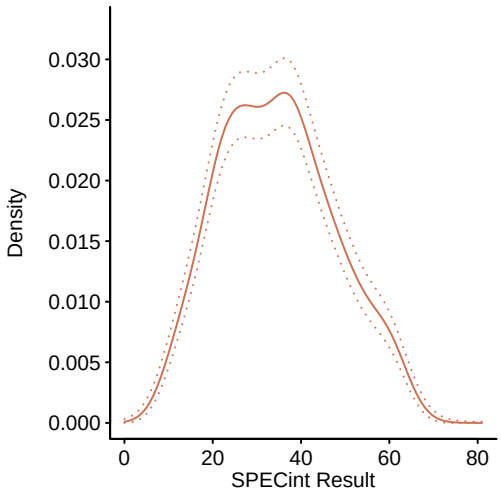


Figure 10.23: Kernel density plot, with 95% confidence interval, of the number of computers having the same SPECint result. Data from SPEC.[1742] Github–Local

Language	mean	sample_size	Probability
rust	39.9	2005.0	8.2
go	30.2	21882.0	99.8
python	32.4	43486.0	79.5
php	27.7	43809.0	100.0
perl	23.7	141393.0	100.0
javascript	38.8	149479.0	0.0
ruby	33.7	151898.0	16.3
java	37.6	165859.0	0.0

Table 10.5: Mean length of sample of regular expressions in languages and bootstrapped probability of occurrence. Data from Davis et al.[444] Github–Local

10.4.5 Permutation tests

For small sample sizes, many computers are fast enough for it to be practical to calculate a statistic (e.g., the mean) for all possible permutations of items in a sample. This kind of test is known as a *permutation test*. Permutation tests do not have any preconditions on the distribution of the sample, other than it be representative of the population, and they return an exact answer.

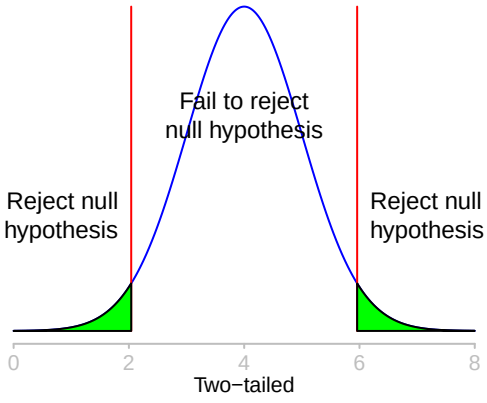
Some techniques designed for manual implementation (e.g., Student's t-test) are approximations to the exact answer returned by a permutation test.

The `coin` package contains infrastructure for creating permutation tests and functions that perform common tasks (the names of these functions are derived from the names of the tests designed for manual implementation, e.g., `spearman_test` and `wilcox_test`).

The following permutation test calculates the likelihood that the professional experience of the two samples of subjects appearing in figure 8.3 have different mean values:

```
library("coin")

# The default is alternative="two.sided",
# an option not currently listed in the Arguments section.
oneway_test(experience ~ as.factor(language), data=Perl_PHP, distribution="exact")
```



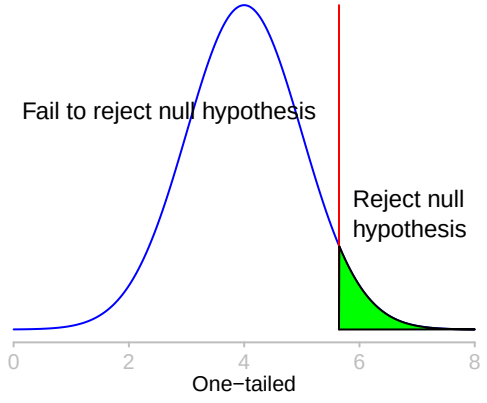


Figure 10.24: One and two-sided significance testing. Github–Local

10.5 Comparing samples

The need to compare measurements, obtained from running experiments, kick-started the development of statistics. The wide range of experimental designs (e.g., one/two/k samples, parametric/non-parametric and between/within subject), along with the need for practical manual solutions, resulted in the evolution of techniques designed to do a good job of handling each specific kind of comparison. This book assumes a computer is available to do the number crunching, and uses either regression (covered in chapter 11), or the bootstrap.[xiii]

Samples may be compared to check whether they are the same/different, in some sense, or by specifically testing whether one sample is greater than, or less than, the other:

- in a *two-sided* test (also known as a *two-tailed* or *non-directional* test) the samples are checked for being the same or different, where an increase or decrease in some attribute is considered a difference. Figure 10.24, upper plot, the percentage on each side is half the chosen p-value,
- in a *one-sided* test (also known as a *one-tailed* or *directional* test) the samples are checked for only one case, either an increase or a decrease in the measured attribute. Figure 10.24, lower plot, the percentage on the one side is the chosen p-value.

A commonly encountered null hypothesis, when comparing two samples, is that there is no difference between them. In many practical situations a difference is expected, or hoped, to exist, otherwise no effort would have been invested in obtaining the data needed to perform the analysis.

[xiii] Other books tend to primarily cover the manual techniques: such as the t-test, which is a special case of multiple regression using an explanatory variable indicating group membership, and the Wilcoxon-Mann-Whitney test, which is essentially proportional odds ordinal logistic regression.

Experiments are often performed because a difference in one direction is of commercial interest. However, expecting or wanting a result that shows a difference in one direction is not sufficient justification for using a one-sided statistical test.

A one-sided test should only be used when the direction is already known, or when an effect in the non-predicted direction would be ignored. If an effect in a particular direction is expected, but an effect in the opposite direction would not be ignored (i.e., would be considered significant) a two-sided test should be used.

Some of the kinds of sample comparisons commonly made include:

- a level of confidence that sample values have been drawn from the same/different distribution (discussed in section 9.2.1),
- the difference, d_m, in the mean of two samples,
- the difference, d_v, in the variance of two samples,
- the correlation, C, between values, paired from two samples.

Correlated measurements: Many data analysis techniques assume that each measurement is independent of other measurements in the sample.

An experiment that measures the same subject before and after the intervention (i.e., a within-subjects design; a between-subjects design involves comparing different subjects) involves correlated data. One technique for handling this kind of correlated data is mixed-effects models, discussed in section 11.6.

Time dependent measurements may be correlated, with later measurements affected by earlier events that are not part of the benchmark (say). A correlation between successive measurements, where none should exist, either needs to be removed or taken into account during analysis. The Durban Watson test can be used to check for a correlation between successive measurements within each run. The `durbinWatsonTest` function, in the `car` package implements this test; see the discussion associated with figure 11.22.

Time series analysis deals with sequentially correlated data, see section 11.10.

10.5.1 Building regression models

Using regression modeling to analyse data may appear to be over-kill (it is used to analyse many of the datasets appearing in this book). When a computer is available to do the work, it makes sense to use the most powerful analysis techniques available that has the fewest preconditions; learning to apply the appropriate, less powerful, technique, often with stronger preconditions, is a waste time (unless you don't have access to a computer).

Techniques designed for manual implementation, such as Pearson correlation, Spearman correlation, t-test, Wilcoxon signed-rank test, etc., are all special cases of regression; for examples of the correspondence with regression, see Github–statistics/manual-tests.R. Manual implementation techniques for comparing two or more samples have been made obsolete by the bootstrap (covered in section 10.4.4), when a computer is available.

Regression provides a simple unified framework for dealing with many data analysis problems; it is possible to start with a simple model, and progressively add more features.

A study by Potanin, Damitio and Noble[1513] refactored the Java Development Kit collection so that it nolonger made use of incoming aliases, e.g., following the owner-as-dominator or owner-as-accessor encapsulation discipline. The DaCapo benchmark,[207] which contains 14 separate programs, was used to compare the performance of the original and refactored versions. The programs were each run 30 times, with measurements made during each of the last five iterations; this process was repeated five times, generating 25 measurements for each program for a total of 350 measurements.

The researchers claimed that their changes to the aliasing properties of the original code did not degrade performance. If the claim is true, the explanatory variable kind-of-refactoring, will have a trivial impact on the quality of the fitted regression model. The simplest model possible is based on the program name, and explains 99.9% of the variance (in this case the intercept is an unnecessary degree of freedom):

```
prog_mod=glm(performance ~ progname-1, data=dacapo_bench)
```

The fitted equation contains just the mean value of the runtime of each separate program, for all programs in the sample. The `summary` function lists the details of the fitted model as: Github–Local

```
Call:
glm(formula = performance ~ progname - 1, data = dacapo)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-4174.6   -205.0     -9.0    116.6   3946.5

Coefficients:
                  Estimate Std. Error  t value Pr(>|t|)
prognameavrora     22881.32      48.03  476.439  < 2e-16 ***
prognamebatik       2519.87      48.03   52.469  < 2e-16 ***
prognameeclipse    53660.53      48.03 1117.330  < 2e-16 ***
prognamefop          395.89      48.03    8.243 2.92e-16 ***
prognameh2         24100.39      48.03  501.823  < 2e-16 ***
prognamejython     15808.13      48.03  329.160  < 2e-16 ***
prognameluindex      708.00      48.03   14.742  < 2e-16 ***
prognamelusearch    7239.52      48.03  150.743  < 2e-16 ***
prognamepmd         4017.61      48.03   83.656  < 2e-16 ***
prognamesunflow    22788.81      48.03  474.513  < 2e-16 ***
prognametomcat      7672.11      48.03  159.750  < 2e-16 ***
prognametradebeans 27987.82      48.03  582.768  < 2e-16 ***
prognametradesoap  64888.58      48.03 1351.122  < 2e-16 ***
prognamexalan      26381.35      48.03  549.318  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 345970)

    Null deviance: 1.5873e+12  on 2100  degrees of freedom
Residual deviance: 7.2169e+08  on 2086  degrees of freedom
AIC: 32759

Number of Fisher Scoring iterations: 2
```

Adding kind-of-refactoring as an explanatory variable (see Github–regression/dacapo_progname.R for details), finds that it is not significant on its own, but an interaction exists between it and a few programs (primarily sunflow). The model:

```
prog_refact_mod=glm(performance ~ progname+progname:refact_kind,
                                                    data=dacapo_bench)
```

explains 99.92% of the variance. There are 12 program/refactoring interactions with p-values less than 0.05 (out of 84 possible interactions), with most of these changing the estimated mean performance by around 1% and one making 8% difference (i.e., sunflow); see Github–regression/dacapo_progname_refact.R.

Building a regression model has enabled us to confirm that, apart from a few, small, interactions the various refactorings of JDK did not change the DaCapo benchmark performance.

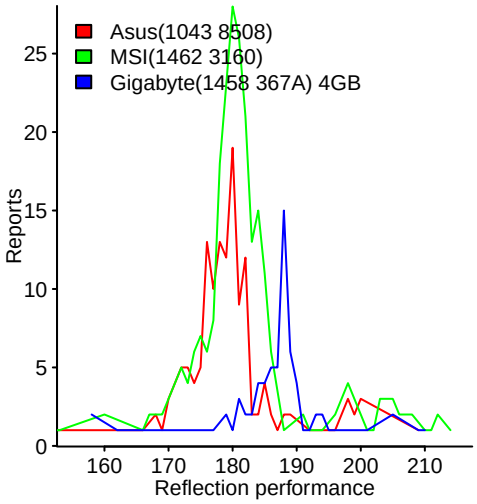


Figure 10.25: Number of Reflection benchmark results achieving a given score, reported for GTX 970 cards from three third-party manufacturers. Data extracted from UserBenchmark.com. Github–Local

10.5.2 Comparing sample means

Comparing two samples, to check for a difference in their mean values, is perhaps the most common statistical test performed. The bootstrap is a general purpose technique for answering sample comparison questions; see section 10.4.4.

The nVidia GTX 970 is a popular graphics card, with many variations on the reference design being sold (during August 2016 there were 51 variants included in the 64,392 results for this card in the UserBenchmark.com database). Figure 10.25 shows the number of Reflection benchmark results reported for GTX 970 cards, from three third-party manufacturers.

The mean score of these Asus, MSI and Gigabyte cards are 176.2, 179 and 186.8 respectively. Are these differences more likely to be the result of random variation or by some real hardware/software difference?

The bootstrap can be used to answer this question, as follows.

Assume there is no difference in the mean performance of, say, MSI and Gigabyte on the Reflection benchmark. In this case the benchmark results (255 from MSI and 73 from Gigabyte) can be merged to form a sample of 328 results. Using this combined empirical sample perform the following:

- randomly select, with replacement, 328 items from the empirical sample,
- divide this new sample into two subsamples, randomly selecting one to contain 255 items and the other 73 items,
- find the mean of the two subsamples, subtract the two mean values and record the result,
- repeat this process R times,
- count how many of these bootstrapped differences in the mean are greater than the differences in the means of the two cards; no assumption is made about the direction of the difference, i.e., this is a two-sided test.

The following code uses the `boot` function, from the `boot` package, to implement the above algorithm, with the user provided function (`mean_diff` in this case) that is called for each randomly generated sample (see Github–group-compare/UserBenchmark_compare.R):

```
library("boot")

mean_diff=function(res, indices)
{
t=res[indices]
return(mean(t[1:num_MSI])-mean(t[(num_MSI+1):total_reps]))
}

MSI_refl=MSI_1462_3160$Reflection
Giga_refl=Gigabyte_1458_367A$Reflection

num_MSI=length(MSI_refl)     # Size of each sample
num_Giga=length(Giga_refl)
total_reps=num_MSI+num_Giga  # Total sample size

GTX_boot=boot(c(MSI_refl, Giga_refl), mean_diff, R = 4999) # bootstrap

refl_mean_diff=mean(MSI_refl)-mean(Giga_refl) # Difference in sample means
# Two-sided test
length(GTX_boot$t[abs(GTX_boot$t) >= abs(refl_mean_diff)]) # == E
```

The argument `R` specifies the number of resamples, with `boot` returning the result of calling `mean_diff` for each resample.

The likelihood of encountering a difference in mean values, as large as that seen in the MSI and Gigabyte performance (i.e., the p-value), is given by the equation: $\frac{E+1}{R+1}$, where: E is the number of cases where the bootstrap sample had a larger mean difference. The result varies around: $\frac{34+1}{4999+1} \rightarrow 0.007$ (the MSI/Asus the value is: $\frac{840+1}{4999+1} \rightarrow 0.17$).

If there were no difference in performance, a difference in mean value as large as that seen for MSI/Gigabyte is expected to occur 0.7% of the time. Based on a 5% cut-off, we can claim this percentage is so small that there is likely to be a real difference in performance. A mean difference at least as large as the MSI/Asus mean difference, is likely to occur 17% of the time, when there was no real difference in performance; a large enough percentage to infer that there is unlikely to be any difference in performance.

If a difference is thought likely to exist, the next question is the likely size of the difference, and the confidence intervals on this value. A bootstrap procedure can be used to answer these questions.

Once two samples are considered to be different, items within each sample can only be treated as exchangeable with other items within the corresponding sample. The two subsample now have to be selected from their respective empirical samples, as in the following code (see Github–group-compare/UserBenchmark_mdiff.R):

```
library("boot")

mean_diff=function(res, indices)
{
t=res[indices, ]
```

```
return(mean(t$refl[t$vendor == "Gigabyte"])- mean(t$refl[t$vendor == "MSI"]))
}

# Need to identify vendor used for each measurement.
MSI_refl=data.frame(vendor="MSI", refl=MSI_1462_3160$Reflection)
Giga_refl=data.frame(vendor="Gigabyte", refl=Gigabyte_1458_367A$Reflection)

MSI_Giga=rbind(MSI_refl, Giga_refl)

# Pass combined dataframe and specify identifying column
GTX_boot=boot(MSI_Giga, mean_diff, R = 4999, strata=MSI_Giga$vendor)
```

The `boot.ci` function calculates confidence intervals from the values returned by `boot` (in this case, the difference in mean values): Github–Local

```
> boot.ci(GTX_boot)
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 4999 bootstrap replicates

CALL :
boot.ci(boot.out = GTX_boot)

Intervals :
Level      Normal              Basic
95%   ( 4.439, 11.168 )   ( 4.378, 11.095 )

Level     Percentile            BCa
95%   ( 4.534, 11.251 )   ( 4.908, 11.739 )
Calculations and Intervals on Original Scale
> mean(GTX_boot$t)
[1] 7.825658
> sd(GTX_boot$t)
[1] 1.716761
```

Deciding if, and when, items in a sample are exchangeable can be non-trivial and requires an understanding of the problem domain.

A study by Gandomani, Wei and Binhamid[648] investigated the accuracy of software cost estimates, made using both expert judgement and Planning Poker, on 15 projects in one company, and both expert judgement and Wideband Delphi in 17 projects in another company; table 10.6 shows a subset.

Is there a difference in the estimates made using expert judgement and either of the other two techniques?

Project	Expert judgement	Planning Poker	Difference
P1	41	40	1
P4	60	56	4
P7	33	45	-12
P12	18	20	-2

Table 10.6: Effort estimates made using expert judgement and Planning Poker for several projects. Data from Gandomani et al.[648]

Each estimate is specific to one project, and it makes no sense to include estimates from other projects in the random selection process; estimates from different projects are not exchangeable. Possible ways of handing this include:

- treating each project as being exchangeable; the resampling could occur at the project level with both estimates for each selected project being used. This makes no sense, from the business perspective.
- randomly selecting from the set of estimates for each project. This possibility is not ruled out, from the business perspective, even though there are only two estimates for each project.

The following code randomly samples estimates for each project (see Github–group-compare/16.R):

```
mean_diff=function()
{
```

```
s_ind=rnorm(len_est_2) # random numbers centered on zero
# Randomly assign estimates to each group
expert=c(est_2$expert[s_ind < 0], est_2$planning.poker[s_ind >= 0])
# Sampling with replacement, so two sets of random numbers needed
s_ind=rnorm(len_est_2) # random numbers centered on zero
poker=c(est_2$expert[s_ind < 0], est_2$planning.poker[s_ind >= 0])
# The code for sampling without replacement
# poker=c(est_2$expert[s_ind >= 0], est_2$planning.poker[s_ind < 0])
return(mean(expert)-mean(poker))
}

est_mean_diff=abs(mean(est_2$expert)-mean(est_2$planning.poker))
len_est_2=nrow(est_2)

t=replicate(4999, mean_diff()) # Run the bootstrap

# What percentage of means are as large as the experiment?
100*length(which(abs(t) > est_mean_diff))/(1+length(t))
```

The p-value for a two-sided test between Expert and Planning Poker is 0.02, which suggests there is a difference, but does not provide any information about the direction of difference; see Github–group-compare/16.R.

A study by Jørgensen and Carelius[950] asked companies to bid on a software development project.[xiv] In the first round of bidding 17 companies were given a one-page description of user needs and asked to supply a non-binding bid; in the second round the original 17 companies plus an additional 18 companies (who had not participated in the first round) were given an 11-page specification (developed based on feedback from the first round) and asked to submit firm-price bids.

What difference, if any, did participating in the first round make to the second bids, submitted by the initial 17 companies (call them the A companies) and how did these bids compare to those submitted by the second sample of 18 companies bidding for the first time (call them the B companies)?

Figure 10.26 shows density plots of the submitted bids. The mean values were: kr183,051[xv] for initial bid from A companies, kr277,730 for final bid from A companies and kr166,131 for single bid from B companies.

Are the items in each sample (the companies asked to submit a bid) exchangeable? Small companies have lower operating costs than large companies; it is unrealistic to consider bids from small/large companies to be exchangeable. The size of companies involved in bidding were classified as small (five or fewer developers), medium (between 6 and 49 developers) and large (50 or more developers).

The call to `boot` has to include information on how the data is stratified, i.e., split into different levels. The argument `strata` is used to pass a vector of integer values specifying the strata membership of the values present in the first argument. Everything else stays the same, with `boot` treating members of each strata as exchangeable when generating, new samples (see Github–group-compare/compare-bid.R):

```
bid_boot=boot(comp_bid$Bid, mean_diff, R = 4999,
                              strata=as.factor(comp_bid$CompSize))
```

The p-value, for the hypothesis that the mean values are the same, is: $\frac{52+1}{4999+1} \to 0.01$, i.e., a difference this large is (statistically) surprising.

Jørgensen and Carelius proposed the hypothesis that the main factor controlling the size of the bids was the information contained in the project specification. I think this is rather idealistic, more practical considerations are discussed in section 5.2.

The intervals of a time series are, by their very nature, not exchangeable. Bootstrapping a time series requires its own distinct algorithm; the `tsboot` function handles the details.

Permutation tests: When the two samples contain only a few items, it is practical to generate and test all possible item permutations.

A study by Grant and Sackman[727] measured the time taken for subjects to write a program using either an online or offline computer interface (this experiment was run during the

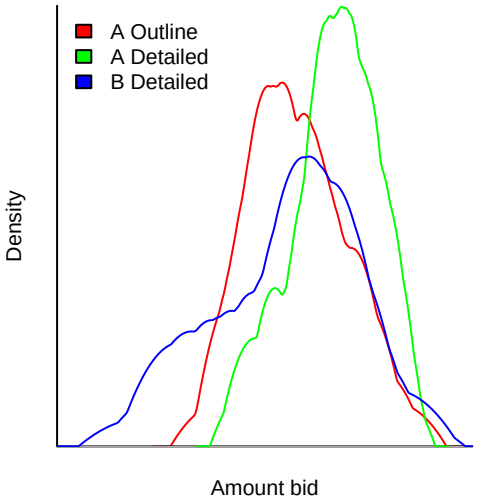


Figure 10.26: Density plots of project bids submitted by companies before/after seeing a requirements document. Data from Jørgensen et al.[950] Github–Local

[xiv] Four of the companies that submitted a bid were selected to independently implement the project.
[xv] The exchange rate was approximately 10 Norwegian Krone to one Euro.

1960s mainframe era). Given 12 subjects, split into two groups of six, how likely is the difference in mean time between the online/offline use cases?

This question is about the population of people who took part in the experiment, not a wider population. For this population there are `choose(12, 6)` ==924 possible subject combinations. The following is an excerpt of an implementation of a two-sided test (see Github–group-compare/GS-perm-diff.R):

```
subj_time=c(online$time, offline$time)  # Combine samples
subj_mean_diff=mean(online$time)-mean(offline$time)

# Exact permutation test
subj_nums =seq(1:total_subj)
# Generate all possible subject combinations
subj_perms=combn(subj_nums, subj_online)

mean_diff = function(x)
{
# Difference in mean of one combination of subjects
mean(subj_time[x]) - mean(subj_time[!(subj_nums %in% x)])
}

# Indexing by column iterates through every permutation
perm_res=apply(subj_perms, 2, mean_diff)

# p-value of two-sided test
sum(abs(perm_res) >= abs(subj_mean_diff)) / length(perm_res)
```

For the Algebra program, 272 of the possible groups, of subject combinations, had a difference in mean time greater than, or equal, to that of the empirical sample. Because all possibilities have been calculated, the p-value is exact: $\frac{272}{924} \rightarrow 0.2934$.

The `coin` package provides this kind of exact calculation for many of the traditional group comparison tests, e.g., the `wilcoxsign_test` function is the permutation test equivalent of the `wilcox.test` function (in the base library).

The bootstrap techniques used to answer questions about differences in the mean of two samples, can be generalised to a wide variety of comparison tests. A new comparison test can be implemented by replacing the `mean_diff` function used in the earlier examples (the requirement of exchangeability remains an integral requirement).

10.5.3 Comparing standard deviation

A study by Jørgensen and Moløkken[954] asked 19 professional developers to estimate the effort required to implement a task, along with an uncertainty estimate, i.e., minimum and maximum about the most likely value. Nine of these developers were explicitly instructed to compare the current task with similar projects they had worked on (they were also given a table that asked them to assess similarity within various percentage bands).

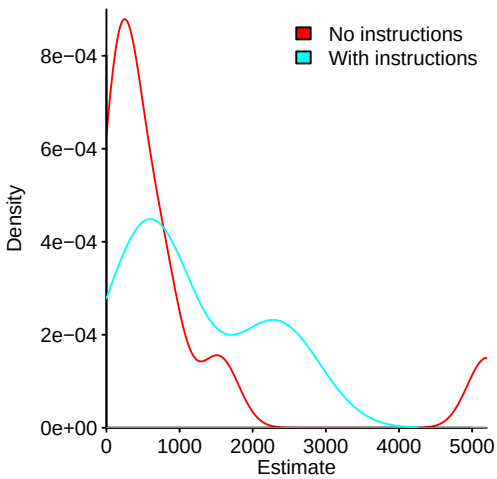


Figure 10.27: Density plot of task implementation estimates: with no instructions (red) and with instruction on what to do (blue). Data from Jørgensen el al.[954] Github–Local

The visual appearance of the density plots, in figure 10.27, suggests that there is a difference in the standard deviation of the estimates in the two samples. A bootstrap test, of the difference in the standard deviations of the two samples, can be implemented by replacing the `mean_diff` function used in the previous section, by the function `sd_diff` as follows (see Github–group-compare/simula_04sd.R):

```
sd_diff=function(est, indices)
{
t=est[indices]
return(sd(t[1:num_A_est])-sd(t[(num_A_est+1):total_est]))
}
```

The p-value, for the hypothesis that the standard deviations are the same, is: $\frac{2170+1}{4999+1} \rightarrow 0.43$, i.e., the difference is not that (statistically) surprising.

The `ansari_test` function, in the `coin` package, [xvi] performs an *Ansari-Bradley Test* (a two-sample permutation test for a difference in variance); see Github–group-compare/simula_04_var.R.

[xvi] The `ansari.test` function is included in R's base system.

10.5.4 Correlation

Correlation is a measure of linear association between variables, e.g., the extent to which one variable always increases/decreases when another variable increases/decreases. The range of correlation values is -1 (the variables change together, but in opposite directions) to 1 (the variables always change together), with zero denoting no correlation.

Correlation is related to regression, except that: it treats all variables equally (i.e., there are no response or explanatory variables), the correlation value is dimensionless and correlation is a linear relationship (i.e., there need not be any correlation between variables having a non-linear relationship, e.g., in the $y = x^2$ relationship, y can be predicted x, but there is zero correlation between them).

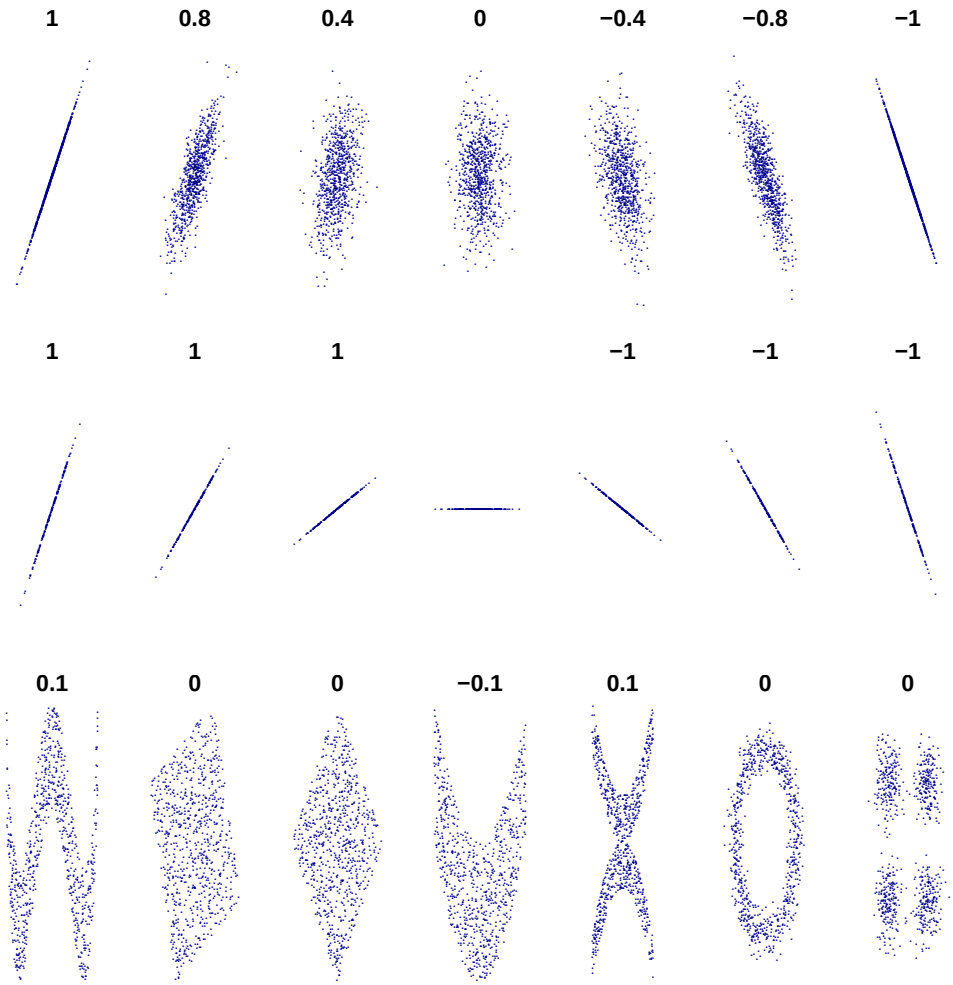


Figure 10.28: Examples of correlation between samples of two value pairs, plotted on x- and y-axis. Github–Local

Three commonly encountered correlation metrics are:

- *Pearson product-moment correlation coefficient*, (also known as Pearson's R or Pearson's r), which applies to continuous variables,
- Spearman's rho, ρ (a lowercase Greek letter), is identical to Pearson's coefficient except the correlation is calculated from the ranked values, i.e., the sorted order (which makes it immune to extreme values),
- Kendall's tau, τ (a lowercase Greek letter), is like Spearman's rho in that it is based on ranked values, but the calculation is based on the number of items sharing the same rank, i.e., relative difference in rank is not included in the calculation; Spearman's rho does include relative differences.

The `cor.test` function, included in the base system, supports all three coefficients and provides confidence interval.

Dichotomous variables: When the result of a measurement has one of two values, the standard techniques for calculating correlation, which require that most if not all values be unique, cannot be used. It is possible to recast the problem in terms of probabilities, which means that the approach taken for every problem could be different.

The following is an example of one approach to a particular binary problem involving binary measurements.

One technique for having high reliability access files, is to host the files on two or more websites; if one site cannot be accessed, the file could be obtained from another site. The naive analysis suggests, that, if the average reliability of the websites is 95%, then the reliability of two paired sites would be 99.75%. However, this assumes the unavailability of each website is independent of its paired site.

A study by Bakkaloglu, Wylie, Wang and Ganger[123] had a client program read a file from over 120 websites every 10-minutes, between September 2001 and April 2002. They recorded whether the file was successfully accessed or not.

Most websites were available most of the time. Bakkaloglu et al proposed various techniques for calculating correlated failures, based on the probability that site X is unavailable when site Y is unavailable, i.e., $P(X unavailable|Y unavailable)$. The following example takes the mean value over all pairs of sites:

$$= mean(P(X unavailable|Y unavailable))$$

$$= mean\left(\frac{P(X\&Y unavailable)}{P(Y unavailable)}\right)$$

The following calculates the average unavailability probability for one site paired with every other site; see Github–probability/reliability/web-avail.R:

```
given=web_down[ , ind]
others=web_down[ , -ind]

both_down=(others & given)

av_prob=mean(colSums(both_down)/sum(given))
```

Averaged over all pairs of sites the probability of one site being unavailable, when its pair is also unavailable, is 0.3 (at the 10-minute measurement point). Given that all accesses originated from the same client, it is not surprising that this probability is much higher than the average probability of one site being unavailable (0.1); all accesses start off going through the same internet infrastructure and problems in this infrastructure will affect access to all sites.

10.5.5 Contingency tables

Count data with categorical explanatory variables has a natural visual representation, as a table of numbers; these tables are known as *contingency tables*. Table 10.7 shows a count of items in the sample having both the listed row and column attributes.

Contingency tables are a technique for reducing lots of data to a compressed visual form. Reasons for compressing data to this form include: wanting to hide information (i.e., readers have to think about what is being presented), not knowing how to make the best use of available information, i.e., the compressed form throws away potentially useful information. Analysis of the uncompressed data is likely to reveal more about it, than an analysis of the simplified form.

Sometimes the only available data is present in a contingency table.

A study by Nightingale, Douceur and Orgovan[1381] investigated the characteristics of hardware failures over a very large number of consumer PCs. Table 10.7 shows a contingency table containing the available data, i.e., the number of system crashes believed to have been caused by hardware problems involving the system DRAM or CPU.

	DRAM failure	**no DRAM failure**
CPU failure	5	2,091
no CPU failure	250	971,191

Table 10.7: Number of system crashes of consumer PCs traced to CPU or DRAM failures. Data from Nightingale et al.[1381]

The traditional, manual friendly, technique for analyzing this kind of data is the chi-squared test (χ is the Greek letter), which provides a yes/no answer.[xvii]

[xvii] The `chisq.test` function is part of the base system; if your readership demands a chi-squared test, the `chisq_test` function in the `coin` package can be used to bootstrap confidence intervals.

A regression model can be fitted to this data (even though there is not a lot of it), extracting more information than the chi-squared test. Github–Local

```
Call:
glm(formula = failures ~ CPU * DRAM, family = poisson, data = PC_crash)

Deviance Residuals:
[1]  0  0  0  0

Coefficients:
                  Estimate Std. Error   z value Pr(>|z|)
(Intercept)      13.786278   0.001015 13586.243  < 2e-16 ***
CPUTRUE          -6.140881   0.021892  -280.505  < 2e-16 ***
DRAMTRUE         -8.264818   0.063254  -130.661  < 2e-16 ***
CPUTRUE:DRAMTRUE  2.228858   0.452194     4.929 8.27e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance:  2.6646e+06  on 3  degrees of freedom
Residual deviance: -7.7825e-11  on 0  degrees of freedom
AIC: 43.948

Number of Fisher Scoring iterations: 3
```

The information in the fitted model that is not immediately obvious from the numbers in the table, is that the crash rate is higher when both the CPU and DRAM fail (although, in this case, an obvious conclusion).

The regression approach makes it trivial to handle more rows and columns, as well as non-straight line fits. As the number of values in the table increases, it becomes more difficult to visually extract any patterns that may be present; figure 10.29 shows how a heatmap might be one way of highlighting details.

There are a wide variety of techniques for comparing multiple contingency tables. Note: different pair comparison algorithms can give very different results.[1812]

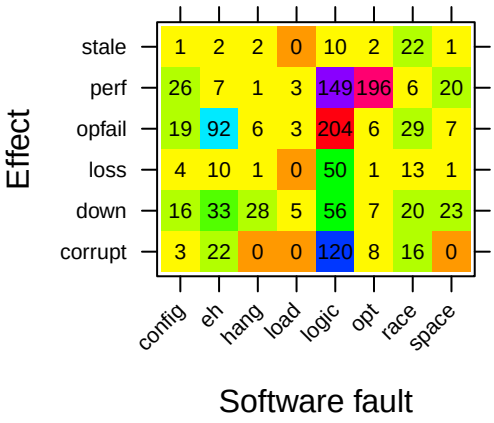


Figure 10.29: Number of software faults having a given consequence, based on an analysis of faults in Cassandra. Data from Gunawi et al.[755] Github–Local

10.5.6 ANOVA

Readers are likely to encounter the acronym ANOVA (*Analysis of variance*), an analysis technique that developed independently of linear regression and having its own specialized terminology. This technique was designed for manual implementation.

Functionally ANOVA and least squares are both special cases of the general linear model (ANOVA is a special case of multiple linear regression with orthogonal, categorical predictors; ANCOVA adds covariates to mix). A one-way analysis of variance can be thought of as a regression model having a single categorical predictor, that has at least two (usually more) categories.

Treating the various kinds of ANOVA models as special cases of the family of regression models, makes it possible to use the more flexible options available in regression modeling, e.g., easier handling of unequal group sizes, adjusting for covariates and methods for checking models.

The `anova` function generates ANOVA style output, when passed a model built using `glm` and some other regression model building functions; the `Anova` function in the `car` package supports more functionality.

One-way ANOVA focuses on testing for differences among a group of means; it evaluates the hypothesis that $\alpha_i = 0$ in the following equation:

$$Y_i = \mu + \alpha_i + \varepsilon_i$$

where: μ is the group mean, α_i is the effect of the response variable on the i'th group and ε_i is the corresponding error.

Chapter 11

Regression modeling

11.1 Introduction

Regression modeling is the default hammer used in this book to create the output from data analysis of software engineering data; figure 11.1, gives a high level overview of the various kinds of hammers available in the regression modeling toolkit. Concentrating on a single, general technique, removes the need for developers to remember how to select from, and use, many special purpose techniques (which in many cases only return a subset of the information produced by regression modeling).

The arrow lines connect related regression techniques, based on the characteristics of the data they are designed to handle; the techniques highlighted in red are the common use cases for their respective data characteristics.

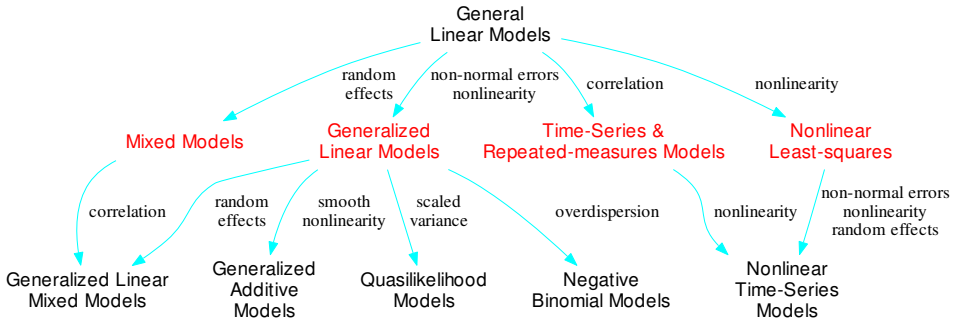


Figure 11.1: Relationship between data characteristics (edge labels) and applicable techniques (node labels) for building regression models.

Regression modeling is powerful enough to fit almost any data to within any selected error bounds, which means overfitting is an ever present danger;[i] model validation (e.g., how well a model might fit new data, or an estimation of the benefit obtained from including each coefficient in a model) is an important self-correcting step.

As always, it is necessary to remember the adage: "All models are wrong, but some are useful."

The main reasons for building a regression model are:

- understanding: structuring the explanatory variable(s) in an equation that can be used to interpret the impact they have on the response variable, i.e., build an understanding of the processes that influence the response variable to behaves the way it does,
- prediction: that is predicting the value of the response variable, for values of the explanatory variables that have not been measured.

The focus of interpretive modeling is understanding why, which creates a willingness to trade-off prediction accuracy for model simplicity, while the focus of predictive modeling is accuracy of prediction, which creates a willingness to trade-off understanding of behavior for greater accuracy.

[i] It is possible to fit an expression containing a single parameter to any data, to any desired degree of accuracy.[229]

Understanding is the primary focus for the model building in this book; builders of computing systems are generally interested in controlling what is happening and control requires understanding; predicting is a fall back position. Model building for prediction is often easier than building for understanding, once readers master building for understanding they will not find it difficult to switch to a predictive focus.

Regression models contain a *response variable*, one or more *explanatory variables*[ii], and some form of error term.

The *response variable* is modeled as some combination of *explanatory variables* and an additive or multiplicative error term (the error term associated with each explanatory variable represents behavior not accounted for by the explanatory variable; different kinds of regression model make different assumptions about the characteristics of the error).

It is always possible to concoct a model that fits some data to within any error tolerance, i.e., the amount of variation in the measurements used, that the model does not explain. It is very important to always ask how well a model is likely to fit all the data likely to be encountered, not just the data used to build it.

If a sample contains many variables, then it is sometimes possible to build a model only using a few of these variables, that has an impressive fit to the chosen response variable. A study by Zeller, Zimmermann and Bird[2005] built a fault prediction model whose performance was comparable to the best available at the time. The model used four explanatory variables to predict the probability of a fault report being associated with the source code contained a file; the explanatory variables were the percentage occurrence of each of the characters `IROP` in each file. The model was *discovered* by checking how good a job every possible character did at predicting fault probability, and picking those that gave the best fit.

11.2 Linear regression

The simplest form of regression model is linear regression, where the *response variable* is modeled as a linear combination of *explanatory variables* and an additive error (the error terms are assumed to be independent and identically distributed; ε denotes the total error). The equation is:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \varepsilon \tag{11.1}$$

Note that the term *linear* refers to the coefficients of the model, i.e., β, not the form taken by the explanatory variables, which may have a non-linear form, as in:

$$y = \alpha + \beta x^2 + \varepsilon$$

or:

$$y = \alpha + \beta \log(x) + \varepsilon$$

A linear model is perhaps the most commonly used regression model, reasons for this include:

- many real world problems exhibit linear behavior, or a good enough approximation to it for practical purposes, over their input range,
- they are much easier to fit manually than more sophisticated models, and until recently software to build other kinds of models was not widely available,
- they can generally be built with minimal input from the user (apart from having to decide which column of data to use as the response variable).

The `glm` function[iii] builds a linear model, and the common use case has two arguments, a formula expressing a relationship between variables (response variable on the left and explanatory variable(s) on the right), and an object containing the data (this object is required to contain columns whose names match the identifiers appearing in the formula).

[ii] Books that focus on the predictive aspect of models, use the term *prediction variable* or just *predictor*, while those that focus on running experiments use terms such as *control variables* or just the *controls*.

[iii] Many books start by discussing the `lm` function, rather than `glm`, because the mathematics that underpins it is easier to learn, another reason for this is herd mentality, it's what everybody else does; if you dear reader want to learn this mathematics I recommend taking this approach. As its name implies the Generalised Linear Method has a wider range of applicability and its use here is in line with the aim of teaching one technique that can be used everywhere. Also, the mathematics behind `glm` makes fewer assumptions about the sample characteristics, e.g., it does not require that the variance in the error be constant (which `lm` does).

The formula has the form of an equation, with the = symbol replaced by ~ (pronounced *is distributed according to*) and the coefficients α and β are implicitly present, i.e., they do not need to be explicitly specified in the code.

The following code uses `glm` to build a model showing the relationship between the number of lines of source code (*sloc*) in FreeBSD, and the number of days elapsed since the project started (in 1993):

```
BSD_mod=glm(sloc ~ Number_days, data=bsd_info)
```

The fitted equation is:

$$E[sloc] = \alpha + \beta \times Number\_days$$

where: $E[sloc]$ is the expected value of *sloc* (the error term is discussed below).

Figure 11.2 shows the measured data points, and a straight line based in the coefficients contained in the object returned by `glm`.

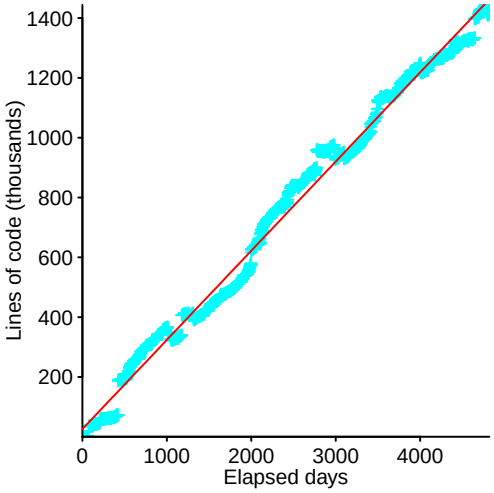


Figure 11.2: Total lines of source code in FreeBSD by days elapsed since the project started (in 1993). Data from Herraiz.[817] Github–Local

The `summary` function takes the object returned by `glm` and prints details about the fitted model;[iv] the following is for the model fitted to the FreeBSD data: Github–Local

```
Call:
glm(formula = sloc ~ Number_days, data = kind_bsd)

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-82990  -32136   -3609   35389   87324

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.139e+05  1.171e+03   97.24   <2e-16 ***
Number_days 3.937e+02  4.205e-01  936.33   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 1657283104)

    Null deviance: 1.4610e+15  on 4826  degrees of freedom
Residual deviance: 7.9964e+12  on 4825  degrees of freedom
AIC: 116172

Number of Fisher Scoring iterations: 2
```

The table following the `Coefficients:` header, in the `summary` output, lists the fitted values for α and β (Intercept and `Number_days` respectively), the standard error in these estimates (`Std.Error`) and the probability that, if the true value of the coefficient was zero, the estimated value would have occurred by chance (in the `Pr(>|t|)` column).

The values listed in the `summary` output can be plugged into the model formula to give the following fitted equation:

$$sloc = 1.139 \cdot 10^5 + 3.937 \cdot 10^2 Number\_days \tag{11.2}$$

The fit between the model and the data is not perfect, and the following are the two forms of uncertainty, or variation, present in the model:

1. Uncertainty in the values of the model coefficients. The values listed in the `Std. Error` column denote one standard deviation, which when added to the model gives the following:
$$sloc = (1.139 \cdot 10^5 \pm 1.171 \cdot 10^3) + (3.937 \cdot 10^2 \pm 4.205 \cdot 10^{-1}) Number\_days \tag{11.3}$$

2. Uncertainty caused by the inability of the explanatory variable used in the model to explain everything. This uncertainty is the ε appearing in equation 11.1; the term *residual* is used to denote this quantity. In the general case it is unlikely that ε will have a fixed value over the range of values supported by a model and `glm` does not return any value(s).

   In figure 11.2 the variations in the unexplained error, ε, appear to be small. The `aov` function can be used to obtain a single fixed value; it returns 40,710 as the residual standard error. The equation, including this estimate of the residual is:

[iv] Only a few digits of the estimated values are printed by default.

$$sloc = 1.139 \cdot 10^5 + 3.937 \cdot 10^2 Number\_days \pm 4.071 \cdot 10^4$$

In other words, the difference between measured values and values calculated using this fitted model are predicted to have a standard error of $4.071 \cdot 10^4$.

The object returned by the call to `glm` can be used to make predictions, and these can be overlaid on the output from an earlier call to `plot`, as follows:

```
BSD_pred=predict(BSD_mod)  # uses fitted model and measured values
lines(BSD_pred, col="red") # x-axis starts at 1 and increment
```

The `predict/lines` approach follows this book's aim of using techniques that work for the general case. Plotting a fitted straight line is such a common operation that there is a function for doing just that, e.g., `abline(reg=BSD_mod, col="red")`, but this does not always work when the axis have been scaled in some way and is of no use for fitted models that are more complicated than a straight line.

Before being carried away with the high degree of agreement between this model and the data, it is important to remember that the model has a number of characteristics that do not reflect reality, including:

- source code does not spontaneously grow of its own accord, and the only justification for treating *number of days* as an explanatory variable is that the resulting model provides potentially interesting insight into the rate of growth of these software systems.
- when it started the BSD project contained zero lines of code, but this model has an Intercept of $1.39 \cdot 10^5$,
- the model shows the number of lines increasing forever, at a constant rate, whereas at some point in the future growth must slow down and eventually stop,
- it says nothing about large amounts of code being added/removed over very short periods (known to exist because of visible breaks in the connectedness of plotted values).

While the model has various disconnects with reality, it does provide strong evidence that growth has been remarkable constant over a long period. Unless there are seismic changes within the FreeBSD development world, the constant rate of code growth would be expected to continue to hold for a non-trivial number of days into the future.

Fitting a model to the data marks the start of the next stage of analysis; creating viable explanations for the processes that could have produced the behavior found. Some factors and processes that might be involved in driving FreeBSD's essentially constant rate of growth over 20 years include:

- developers working on the system have continually discovered new functionality to add,
  - if there has always been functionality to add, why haven't more developers become involved, increasing the rate of growth until there is less to do?
  - to what extent is the continual stream of new hardware devices responsible for driving growth?
- what are the bottlenecks that have prevented increases in growth rate, when the resources have been available?
  - has growth rate remained constant because the developers working on the systems have remained constant?
  - is there a buffer of code waiting to be released, whose growing and shrinking hides an internal growth rate that is much more variable than the externally visible rate?

The questions answered by the analysis of one set of measurements invariable raises more questions, whose answers require more data.

The call to `summary`, passing the value returned by `glm`, is an example of function overloading in action. The value returned by `glm` has class `glm`, which, when passed as an argument to `summary`, results in `summary.glm` being called; a call to `predict` results in `predict.glm` being called (function overloading is the most common use of object-oriented constructs in R programs; the use of a period in the function name is a naming convention followed by the implementers and not something that changes the behavior of the R compiler).

Some readers of data analysis may find a visual presentation of the coefficients of a fitted model, along with their standard error, easier to process. The `sjPlot` package offers a variety of options for plotting of fitted model information.

11.2.1 Scattered measurement values

In the FreeBSD analysis, the measurements ran together in a way that created a visually recognizable line. The common case is not always so accommodating, and often when many samples are plotted a scattering of visually disjoint points appears; viewed as a whole a general trend may emerge.

A study by Kampstra and Verhoef[969] investigated the estimated cost and duration of 73 large Dutch federal IT projects.[v] Figure 11.3 shows that few measurement points are close to the (red) fitted line returned by `glm`; the variability of measured values is much larger than that for the FreeBSD data. While numeric estimates of the uncertainty present in the fitted model are readily available, interpreting these numeric values requires a degree of effort and some experience. A confidence interval provides an easy to interpret visual representation of the uncertainty in a fitted model.

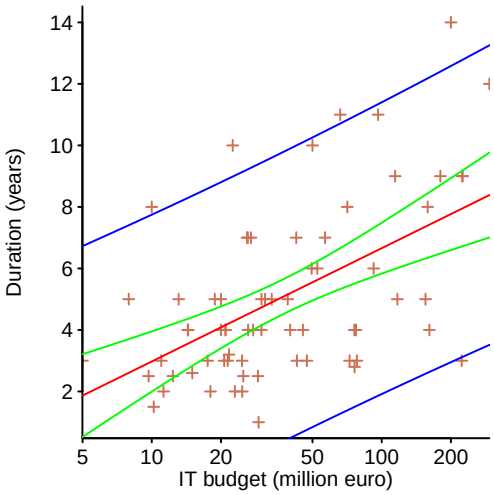


Figure 11.3: Estimated cost and duration of 73 large Dutch federal IT projects, along with fitted model and 95% confidence intervals (green for the bounds of the fitted line and blue for the bounds of any new measurements). Data from Kampstra et al.[969] Github–Local

The kind of uncertainty, in the fitted model, of interest will depend on whether the model is built to gain understanding or make predictions:

- when understanding is the priority, the confidence interval of interest involves the estimated model coefficients:
  - a call to `predict` with the argument `se.fit=TRUE`, returns the standard error for each fitted value. Multiplying `se.fit` by `qnorm`,[vi] converts the returned value to a 95% confidence interval (in this case, 2.5% above and below the fit; the two `qnorm` values differ only in sign because the Normal distribution is symmetrical), i.e., there is a 95% expectation that the actual model fits within the interval enclosed by these lower/upper bounds. `qnorm(0.975)==1.96` and the literal value is often used (sometimes the value 2 is treated as a sufficiently close approximation).[vii]

    ```
    fed_pred=predict(fed_mod, newdata=list(log.IT=1:7, log.IT_sqr=(1:7)^2),
                                                    se.fit=TRUE)
    lines(fed_pred$fit, col="green")     # fitted line
    # CI above and below
    lines(fed_pred$fit+qnorm(0.975)*fed_pred$se.fit, col="green")
    lines(fed_pred$fit+qnorm(0.025)*fed_pred$se.fit, col="green")
    ```

  - the `confint` function in the `MASS` package, or the `boot.ci` function in the `boot` package, can be used to obtain a point estimate of the confidence interval of the fitted model coefficients.
- when prediction is the priority, the interval is known as the *prediction interval*; the bounds between which newly measured values are expected to appear. Two sources of uncertainty are added to calculate the prediction interval: uncertainty in the model coefficients (i.e., the confidence interval) plus the variance in the data not explained by the fitted model; the calculation is (the `predict` function can perform this calculation for a few types of fitted models):

    ```
    # print.aov also calculates it from residuals returned by glm...
    MSE=sum(fed_mod$residuals^2)/(length(fed_mod$residuals)-2)
    # Variances, but not sd, can be added
    pred_se=sqrt(fed_pred$se.fit^2+MSE)
    lines(fed_pred$fit+1.96*pred_se, col="blue")
    lines(fed_pred$fit-1.96*pred_se, col="blue")
    ```

When measurement values, and an associated fitted regression line, are plotted, it is easy to visually fixate on the line and forget about the associated uncertainties. Including a confidence band as part of a plot provides a vivid visual reminder of the uncertainty in the fit.

Plotting values does not always reveal an obvious pattern in the distribution of points. The absence of a visual pattern may be because no relationship exists between the response and explanatory variables, or because the noise in the data is much greater than the signal, i.e., a relationship that exists is swamped by noise present in the measurements.

How much random scattering of measurement values has to exist before a fitted regression model can be said to be not worth bothering about?

[v] They discovered there was a lot of uncertainty in the estimates given.

[vi] This calculation assumes that the measurement error has a Normal distribution, the default assumption made by `glm` when building a model.

[vii] For small sample sizes a call to `qt` may be more accurate.

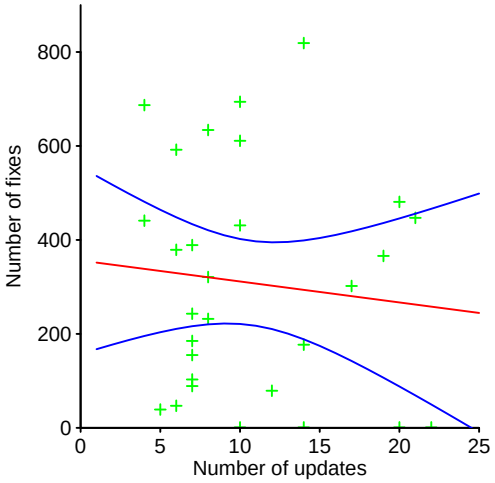


Figure 11.4: Number of updates and fixes in each Linux release between version 2.6.11 and 3.2. Data from Corbet et al.[402] Github–Local

The `glm` function, and many other model building functions available in R, is capable of fitting models to data points that are randomly distributed. For instance, Figure 11.4 shows the number of updates and fixes made in various Linux versions released between early 2011 and 2012. The standard error of the fitted line shows that its slope could have a positive or negative value.

The output from `summary` shows how poor the fit actually is;the `Pr(>|t|)` column lists the p-value for the hypothesis that the coefficient in the corresponding row is zero, i.e., that no relationship was found to exist for that component of the model. Github–Local

```
Call:
glm(formula = Fixes ~ Total.Updates, data = cleaned)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-310.60  -223.67     0.48   184.51   525.26

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)    356.233    101.522   3.509   0.0016 **
Total.Updates   -4.464      8.478  -0.526   0.6029
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 60685.71)

    Null deviance: 1655335  on 28  degrees of freedom
Residual deviance: 1638514  on 27  degrees of freedom
AIC: 405.62

Number of Fisher Scoring iterations: 2
```

11.2.2 Discrete measurement values

Regression models are not limited to fitting continuous numeric explanatory variables, variables having nominal values (i.e., discrete) can also be included in a fitted model.

A study by Cook and Zilles[396] investigated the impact of compiler optimization flags on the ability of software to continue to operate correctly, when subject to random bit-flips, i.e., simulating random hardware errors; 100 evenly distributed points in the program were chosen and 100 instructions from each of those points were used as fault injection points, giving a total of 10,000 individual tests run, for each of 12 programs from the SPEC2000 integer benchmark compiled using gcc version 4.0.2 (using optimization options: 00, 02 and 03) and the DEC C compiler (called *osf*).

The fitted model has percentage of correct benchmark program execution as the response variable[viii] and optimization level as the explanatory variable; the call to `glm` is unchanged:

```
bitflip_mod=glm(pass.masked ~ opt_level, data=bitflip)
```

The `summary` output of the fitted model is: Github–Local

```
Call:
glm(formula = pass.masked ~ opt_level, data = bitflip)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-12.6689   -2.8454   -0.3478    4.4017    8.1100

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    28.589      1.825  15.665  < 2e-16 ***
opt_level02     9.161      2.581   3.550  0.00112 **
opt_level03     7.429      2.581   2.878  0.00677 **
opt_levelosf   11.642      2.414   4.822 2.74e-05 ***
```

[viii] Percentage correct is always between 0 and 100%; technically correct techniques for handling response variables having a lower and upper bound are discussed in section 11.3.6.

```
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 29.97578)

    Null deviance: 1785.8  on 38  degrees of freedom
Residual deviance: 1049.2  on 35  degrees of freedom
AIC: 249.07

Number of Fisher Scoring iterations: 2
```

Plugging the model coefficients into the regression equation we get:

$pass.masked = 28.6 + 9.2 \times D_{O2} + 7.4 \times D_{O3} + 11.6 \times D_{osf}$

where: D_i, known as a *dummy variable* or *indicator variable*, take one of two values:

$$D_i = \begin{cases} 1 & \text{optimization flag used} \\ 0 & \text{optimization flag not used} \end{cases}$$

The value for optimization 00 is implicit in the equation, it occurs when all other optimizations are not specified, i.e., its value is that of the intercept.

The standard error in the 02 and 03 compiler options is sufficiently large for their respective confidence bounds to have significant overlap; suggesting that these two options have a similar impact on the behavior of the response variable.

11.2.3 Uncertainty only exists in the response variable

Many algorithms used to fit regression models attempt to minimise the difference between the measured points and a specified equation. For instance, least-squares minimises the sum of squares of the distance along one axis between each data point and the fitted equation;[ix] alternative minimization criteria are discussed later, e.g., giving greater weight to positive error than negative error.

An important, and often overlooked, detail, is that many regression techniques assume that the values of the explanatory variable(s) contain no uncertainty (i.e., measurements are exact), with all uncertainty, ε, occurring in the response variable; see equation 11.2.

A consequence of assuming uncertainty only exists in the response variable, is that the equation produced by fitting a model that specifies, say, X as the explanatory variable and Y the response variable will not be algebraicly consistent with a model that assumes Y is the explanatory variable and X the response variable. That is, algebraicly transforming the first equation produces an equation whose coefficients are different from the second.

A study by Kroah-Hartman[1046] investigated the number of commits made between the release of a version of Linux and the immediately previous version, and the number of developers who contributed code to that release, for the 67 major kernel releases between versions 2.6.0 and 4.6.

In the upper plot of figure 11.5, the number of developers is treated as the explanatory variable (x-axis), and number of commits as the response variable (y-axis), with the fitted regression line in red and dashed lines showing the difference between measurement and fitted model. In the lower plot the explanatory/response roles played by the two variables, when fitting the regression model, is switched; to simplify comparison the axis denote the same variables in both plots, with the blue line denoting the newly fitted model, and dashed lines showing the difference between measurement and model (now on the x-axis response variable; the line fitted in the upper plot is also plotted for comparison, still in red).

In the first case the fitted equation is:

$$commits = -237 \pm 523 + (8.7 \pm 0.44) mathitNumber\_devs \tag{11.4}$$

transforming this equation we get:

$$\begin{aligned} Number\_devs &= \frac{237 + commits}{8.7} \\ &= 27 + 0.11 commits \end{aligned} \tag{11.5}$$

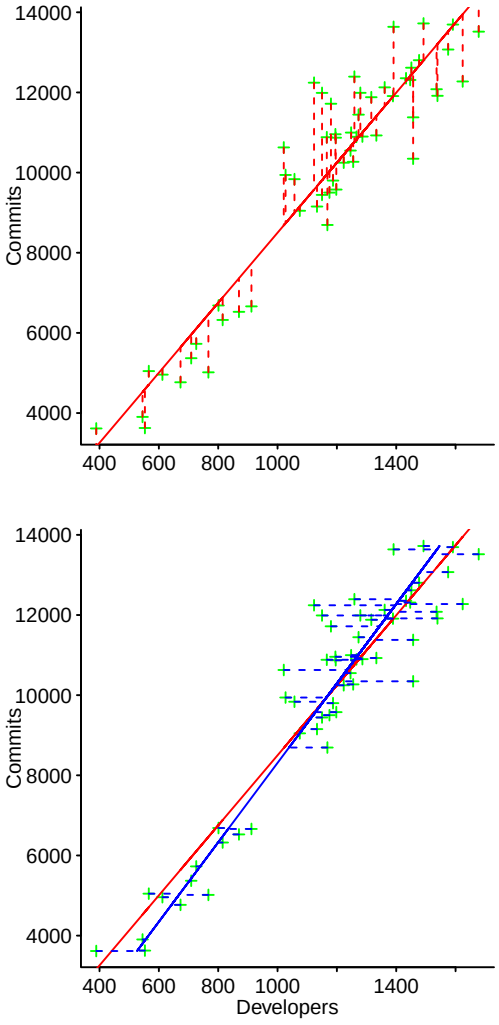


Figure 11.5: Number of commits made, and the number of contributing developers for Linux versions 2.6.0 to 3.12. The blue line in the right plot is the regression model fitted by switching the x/y values. Data from Kroah-Hartman.[1046] Github–Local

[ix] Minimising the sum of squares in the error has historically been popular because it is a case that can be analysed analytically.

However, when a model is fitted by switching the roles of the two variables, in the formula passed to `glm`, the model returned is described by the following equation:

$$Number\_devs = 162 \pm 52 + (0.10 \pm 0.005) commits$$

which differs from equation 11.5, obtained by transforming equation 11.4.

There is another difference between the two fitted models, the second model is a better fit to the data. Somebody who is only interested in the quality of fit may be tempted to select the second model, purely for this reason.

What is the procedure for deciding which measurement variables play the role of response and explanatory variable, e.g., should number of developers be considered an explanatory or response variable?

An important attribute of explanatory variable(s) is that their value is controlled by the person making the measurement. For instance, the model building process used to create figure 11.2 has number of days as the explanatory variable; this choice was completely controlled by the person making the measurements.

The Kroah-Hartman commit measurements are based on the day of release of a version of the Linux kernel, a date that is outside the control of the measurement process. In fact both measurements have the characteristics of a response variables, that is, the value they have, was not selected by the person making the measurement. Both the possibility of variation in Linux version release dates and variation in number of commits made by developers are sources of uncertainty, both variables need to be treated as containing measurement error.

Building a regression model using explanatory variables containing measurement error can result in models containing biased and inconsistent values, as well as inflating the Type I error rate.[269, 1683]

There are a variety of regression modeling techniques that can take into account error in the explanatory variable. These techniques are sometimes known as *model II* linear regression techniques (model I being the case where there is no uncertainty in the explanatory variables), and also as *errors-in-variable models*, *total least-squares* or *latent variable models*; methods used go by names such as *major axis*, *standard major axis* and *ranged major axis*.

If all the variables used to build a model contain some amount of error, then it is necessary to decide how much error each variable contributes to the total error in the model. Some model II techniques are not scale invariant, that is, they are only applicable if both axes are dimensionless or denote the same units, otherwise rescaling one axis (e.g., converting from kilometers to miles) will change its relative contribution. If each axis denotes a different unit, it does not make sense to use a model building technique that attempts to minimise some measure of combined uncertainty.

SIMEX (SIMulation-EXtrapolation) is a technique for handling uncertainty in explanatory variables that works in conjunction with a range of regression modeling techniques. The SIMEX approach does not suffer from many of the theoretical problems that other techniques suffer from, but requires that the model builder provide an estimate of the likely error in the explanatory variable(s). The `simex` package implements this functionality, and supports a wide variety of regression models built by functions from various packages.

Continuing with the Linux developer/commit count example, to build a regression model using SIMEX, we need an estimate of the uncertainty in the number of developers contributing at least one commit to any given release. The `simex` function taking a model built using `glm` (and by other regression model building functions) and an estimate of the uncertainty in one or more of the explanatory variables, and returns an updated model that has been adjusted to take this uncertainty into account.

The following is a rough and ready approach to estimating the uncertainty in the Kernel attributes, measured by Kroah-Hartman:

- the release date of a new version of Linux is assumed to have an uncertainty of ±14 days about the actual release date.[x]
- the possible variation in the unique contributor count for any release is assumed to be uniformly distributed in the range: measured contributor count plus/minus number of developers contributing their first commit in the last 14 days.

[x] Pointers to a more reliable, empirically derived, value are welcome.

- based on these assumptions, a standard deviation of 41 is obtained for the number of unique developers making at least one commit, averaged over all versions; see Github–regression/clean/dev-commit.R.

Integrating this estimate of the standard deviation in the explanatory variable into a regression model is a two-step process:

- first build a regression model using `glm` in the usual way, but with the optional named parameter `x` set to `TRUE` (`y` also needs to be `TRUE`, but this is its default value and so the assignment below is redundant),
- pass the model returned by `glm` to `simex`, along with the name of the explanatory variable and its estimated standard deviation.

In code, the implementation is:

```
yx_line = glm(commits ~ developers, x=TRUE, y=TRUE)

sim_mod=simex(yx_line, SIMEXvariable="developers", measurement.error=41)
```

Compare equation 11.4 with the following equation, derived from the model returned by `simex` (see Github–regression/dc-simex.R):[xi]

$$commits = -387 \pm 453 + (8.9 \pm 0.4) Number\_devs$$

The error in individual explanatory variable measurements can be specified by assigning a vector to `measurement.error` (the argument `asymptotic=FALSE` is also required); see fig 11.35.

How reliable is a fitted model that ignores any uncertainty/error in explanatory variable measurements? The only way to answer this question is to build a model that takes this error into account and compare it with one that does not. The difference between the two ways of structuring fitted models can sometimes be much larger than that in figure 11.5.

The question of whether economies of scale exist for software development might be answered by analysing project effort/size data. Figure 11.6 shows lines for two fitted regression models, one with Effort as the explanatory variable, the other with Size as the explanatory variable (from a study by Jørgensen, Indahl and Sjøberg[953]). If economies of scale exist, the slope of the effort/size line will be less than one (diseconomies of scale produce a slope greater than one). In this case, one slope is less than one and the other greater than one. The models fitted by switching response/explanatory variables are outside each other's 95% confidence intervals (there is no reason to expect them to be inside).

Many measurement values treated as explanatory variables in this book were not under the control of the person who measured them. For instance, lines of code, number of files and reported problems measured at a given point in time are all response variables. To reduce your author's workload, most model fitting in this book does not make any adjustments for errors in the explanatory variables.

There are techniques, and R packages, for building complete models starting from the data, rather than refitting an existing regression model. For those wanting to a build model from scratch, the `lmodel2` package provides functions that implement many of the available methods.

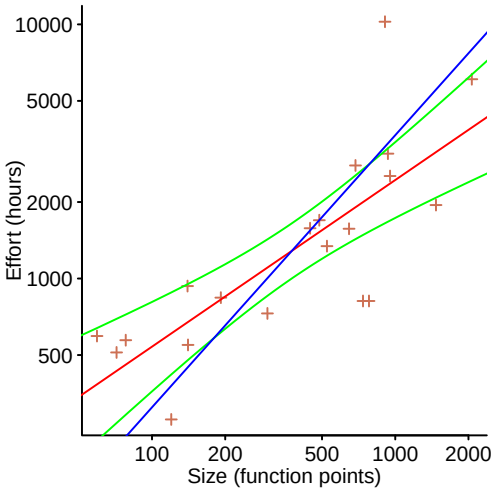


Figure 11.6: Effort/Size of various projects and regression lines fitted using Effort as the response variable (red, with green 95% confidence intervals) and Size as the response variable (blue). Data from Jørgensen et al.[953] Github–Local

11.2.4 Modeling data that curves

A model based on a straight line is a wonderful thing to behold, it is simple to explain and often aligns with people's expectations (many real world problems are well fitted by a straight line). However, life is complicated and throws curved data at us.

Having encountered an operating system having constant lines of code growth over many years, it is tempting to draw a conclusion about the growth rate of other operating systems. However, the way in which the data points curve around the fitted line in the upper plot of figure 11.7, suggests that some of the processes driving the growth of the Linux kernel are different from those driving FreeBSD; perhaps a quadratic or exponential equation would be a better fit (these possibilities were chosen because they are two commonly occurring forms for upwardly curving data).

[xi] Readers might like to experiment with the value of `measurement.error`, to see the impact on the model coefficients.