A call for professional software engineering to be used in scientific research (and epidemic modelling in particular)

This is a preprint. It has not been peer reviewed.

PLEASE EMAIL COMMENTS TO harold@thimbleby.net ASAP

Harold Thimbleby PhD FRSM HonFRSA FRCP(Edin) HonFRCP
See Change Fellow in Digital Health
Swansea University, Wales, SA2 8PP
harold@thimbleby.net
07525 191956

This version at 10:50 am BST on July 27, 2020

Access current version at
http://www.harold.thimbleby.net/reliable-models.pdf

A call for professional software engineering to be used in scientific research (and epidemic modelling in particular)

Harold Thimbleby
See Change Fellow in Digital Health
Swansea University, Wales
Email: harold@thimbleby.net

Abstract

Background: Computer models inform our response to COVID-19. While models are routinely criticised for their assumptions, the algorithms and the quality of code implementing them avoid scrutiny and, hence, cannot be justified.

Problem: Assumptions in programs are hard to scrutinise as they are rarely explicit. In addition, both algorithms and code have bugs, effectively unintentional and unknown assumptions with unwanted effects. Code is fallible. Any model interpretation that relies on code is therefore fallible.

Solutions: Code can be made *much* more reliable using software engineering good practice. Three specific solutions are presented. First, professional software engineers can help. Secondly, Software Engineering Boards (supplementing and analogous to Ethics or Institutional Review Boards) must be instigated and used. Thirdly, code must be considered an intrinsic part of any model, and therefore must be formally reviewed by competent software engineers.

"Criticism is the mother of methodology."
Robert P Abelson
Statistics as Principled Argument, 1995

Introduction

Let us start with an analogy. It is uncontentious to report statistics fully and carefully in the scientific literature. For example, a statistical claim might be written-up as follows:

> Random intercept linear mixed model suggesting significant time by intervention-arm interaction effect. Bonferroni adjusted estimated mean difference between intervention-arms at 8-weeks 2.52 (95% CI $0.78, 4.27, p = 0.0009$). Between group effect size $d = 0.55$ (95% CI $0.32, 0.77$).[1]

All the numbers here are likely to have been generated by computer, but a standard form — confidence intervals, p levels, and so on — is used to present them so the claims can be seen to be complete, and are easy to interpret and to scrutinise.

Computers are now very widely used, not just to calculate statistics from data, but to run the models that generate the data that is analysed. Strangely, the reliance on computer code to generate data (or graphs) is rarely adequately justified, and hence presenting results, such as the statistics above, in conventional 'rigorous' ways may be deceptive. As a matter of fact, the results claimed are no more reliable than the code that was used to generate them.

Just as failure to properly articulate statistics undermines the value of scientific claims, failure to properly document and explain computer code undermines its scientific value, as well as undermining the scientific value of the models and the results it generates. Indeed, as few papers use code that is as well-understood and as well-researched as standard statistical procedures (such as Bonferroni corrections), the scientific problems with poorly reported code are potentially much worse.

Code, and results from code, need to be presented in a way that ground belief in claims derived from using them. Specifically, code must be developed in a sufficiently rational and rigorous way that is able to support clear presentation and scrutiny (developing justifiably reliable code is the domain of software engineering, which will be discussed further below). We would not believe a statistical result that was obtained from an *ad hoc* analysis with some new method; instead, we demand statistics that is recognisable, even traditional, so we are assured we understand what the scientist has done, and more importantly, we understand how the scientist got the results they did — or how they *should* have.

Focusing on epidemic modelling, this paper shows that accepted levels of code quality and discussion in scientific research urgently need addressing.

To make critical claims, to inform policy making or to underpin further research, models need to be run under varying assumptions.[2] Being able to understand (at least in principle) the exact code used in implementing a model is critical to having confidence in the results that rely on it. Unfortunately, code is rarely seen as a research contribution in its own right: it is rarely published in any useful form or discussed in sufficient detail, as this paper will show.

Code needs to be carefully documented and explained because all code has tacit assumptions, bugs and cybersecurity vulnerabilities[3] that, if not articulated, can affect results in unknown ways that may undermine the claims. The problem is analogous to the problem of failing to elaborate statistical claims properly: failure to do so suggests that the claims may have unknown limitations or even downright flaws.

Even good quality code has, on average, a defect every 100 lines — and that rate is only achieved by experienced industrial software developers.[4] World-class software can attain

maybe 1 bug per 1,000 lines of code. Code developed for experimental research purposes will have higher rates of bugs than professional industrial software, because the code is less well-defined and evolves as the researchers gain new insights into their models. It is therefore inevitable that typical modelling code has many bugs. Such bugs undermine confidence in model results.

State of the art

A review of epidemic modelling[5] says, "we use the words 'computational modelling' loosely," and then, curiously, the review discusses exclusively mathematical modelling, implying there is no conscious role for code. Yet without code, models could not be run.

A systematic review[6] of published COVID models for individual diagnosis and prognosis in clinical care, including apps and online tools, noted the common failure to follow standard TRIPOD guidelines.[7] (TRIPOD guidelines ignore code completely, let alone its quality.) The review ignored the mapping from models to their implementation, yet if code is unreliable, the model *cannot* be reliably used, and cannot be reliably interpreted. It should be noted that flowcharts and other implementations of models, which the review did consider, are programs intended for direct human use. They too must be designed at least as carefully as computer programs, for exactly the same reasons: it is hard to program reliably.

A high-profile 2020 COVID-19 model[8,9] uses a modified 2005 computer program[10,11] for H5N1 in Thailand, which did not model air travel or other factors required for later western models. The 2020 model forms part of a series of papers[9–11] none of which provide details of the code. A co-author disclosed[12] that the code is thousands of lines long and is undocumented C code. As Ferguson, the code author, said in an interview, "For me the code is not a mess, but it's all in my head, completely undocumented. Nobody would be able to use it . . . "[13]

This is problematic, especially as the code would have required many non-trivial modifications to update it for COVID-19 with different assumptions; moreover, the code would have had to have been updated very rapidly. If this C code had been made available for review, the reviewers would not have known how to evaluate it without documentation. It is, in fact, hard to imagine how a large undocumented program could have been repeatedly modified over fifteen years without becoming incoherent. If code is undocumented, there would be an understandable temptation to modify it arbitrarily to get desired results; worse, it is methodologically impossible to distinguish legitimate attempts at debugging from merely fudging the results. In contrast, if code is properly documented, the documentation defines the original intentions (including formally using mathematics), and therefore any modifications need to be properly justified and explained — or the theory revised.

The programming language C is not a dependable language; to develop reliable code in C requires professional tools and skills. Moreover, C code is not portable, which limits making it available for other scientists to use safely (C gets different results on different compliers, libraries, and hardware).

Because of its importance, a project started to document this COVID model.[14] Documenting the code now may describe what it does, including bugs, but it is unlikely to explain what it should have done. If nothing else, as the code is documented, bugs will be found, which will then be fixed (refactoring), and so the belatedly-documented code will not be the code that was used in the published models. It is well-known that documenting code helps improve it, so it is surprising to find an undocumented model being used in the first place. The revised code has been published, and it has been heavily criticised, supporting the concerns expressed in the present paper.[15]

Some papers[17] publish models in pseudo-code, a simplified form of programming. Pseudo-code looks deceptively like real code that might be copied to reproduce it, but as pseudo-code introduces invisible simplifications, using it is arguably even worse than not publishing code at all. Pseudo-code, properly used, can give a helpful impression of the overall approach of an algorithm, certainly, but pseudo-code alone is not a surrogate for code. It is not precise enough to help anyone scrutinise a model; copying pseudo-code introduces avoidable bugs. An extensive criticism of pseudo-code, and suggestions for reliable publication of code can be found elsewhere.[18]

Only if there is access to the *actual* code and data (in the specific version that was used for preparing the paper) does anyone know what researchers have done, but merely making code available (for instance, providing it in Supplementary Information with papers, putting it in repositories, or using open source) is not sufficient.

Some COVID-19 papers[e.g.,19,20] do make "documented" code available, but provide no more than superficial comments: this is *not* documentation as properly understood. Such comments do not explain code, explain contracts, nor explain algorithms. Some papers[e.g.,19] make unfinished, incomplete code available.

These problems appear to be representative: of 30 articles listed on *Nature Digital Medicine's* top open access web page on 14 July 2020, pqpers were selected for review if and only if their title implied use of a computer model; 7 (58%, N=12) provide no code at all (though some may provide code on request), and 5 (42%, N=12) provide code but only with trivial comments that do not add useful information to the code.

One paper in the sample does provide trivial documentation for their code, but does not provide any access to it. One paper provides a reference to a GitHub repository that is empty, and is therefore considered not to provide any code (a plausible inference is that no version control or repository was used during preparation of the paper and the authors are planning to migrate to GitHub, but did not do so in time for publication).

None of the sample provide any code with adequate documentation, so none can be considered to provide code that is open to proper review or wider scientific scrutiny — all the code (when provided) is too complex to understand without adequate documentation. (Sample data is provided in this paper's Supplementary Information.)

If full code is made available, it may be technically "reproducible," but the scientific point is to be able to understand and challenge, potentially refute, the findings; to do that, much more is required than merely being able to run the code.[21,22]

Number of papers relying on code	32	100%
Have some or all code accessible	12	37%
Use code repository, e.g., GitHub (1 was empty)	11	34%
Helpful comments	5	16%
Trivial comments	8	25%
No or uninsightful comments (e.g., copyright)	19	59%
Some or all code on request	8	25%
No code available	12	37%
Use data repository, e.g., Dryad	8	25%

Table 1: Summary of exploratory survey.

Even if a computer can run it, badly-written code (as found in *all* the research reviewed in this paper) is inscrutable, even if its original programmers think otherwise. Only if there is access to *adequate* documentation can anyone know what the researchers *intended* to do. Without all three (code, data, adequate documentation), there are dangers that a paper simplifies or exaggerates the results reported, and that bugs and errors in the code or data, perhaps unnoticed by the paper's authors, have affected the results they report.[18]

Making poor code (including pseudo-code) available without proper documentation and without discussing its limitations is unethical: it encourages others to reproduce and build on poor work.

Looking beyond pandemic modelling

The problems of unreliable code are not limited to COVID-19 modelling, which, understandably, has perhaps been rushed into publication. For instance, a 2009 paper reporting a model of H5N1 pandemic mitigation strategies[16] provides no details of its code. Its supplementary information, which might have provided code, no longer exists.

A small sample of papers covering a broad range of science were selected from current online editions (as of July 2020) of *Nature Digital Medicine*, *Royal Society Open Science* and *Lancet Digital*, based on the paper's title implying that a program has been used in the published research. Commentary, correspondence and editorials were excluded. The sample misses many papers that do use code, but the criterion selects papers where the wording of the title indicates that code is considered a component of the scientific contribution. Indeed, all sampled papers used code in their research.

This is not a representative sample of scientific research in general, but it is a selection of recent papers that have been accepted for publication after rigorous peer review in high-profile, highly competitive leading peer-reviewed journals. The sample arguably represents what a broader community currently considers to be best practice. The data is summarised in Table 1.

One paper claimed to have accessible code, but there was no code on the nominated

server. Only one paper had more than one version of code (on GitHub), suggesting that it is rare to use version control systems. The evidence implies that manually developed code is uploaded to a repository before submitting the paper to just "go through the motions," and that published code is not maintained.

8 papers used the Dryad Digital Repository or similar (Figshare etc) to provide structured access to their data. Unlike GitHub, Dryad has scientifically-informed guidelines on handling data, and all papers that used Dryad provided more than just their raw data — they provided a little, sometimes substantial, documentation for their data. The only paper providing documentation for their code used Dryad to publish and document their code.

It should be noted that almost all of the available code contains "magic numbers" — that is, data masquerading as code. This common practice ensures that the published data is rarely all of the relevant data, and emphasises the need for repositories like Dryad to require the inclusion of code.

Evidently, effective access to code is not just an afterthought, it rarely happens. Providing structured repositories that provide suggestions for and encourage good practice (such as Dryad), and requiring their use, would be a lever to improve the quality and value of code in published papers.

The Supplementary Material provides a summary of the analysis, and citations for all papers in the sample.

Bugs, code and programming

Critiques of data and model assumptions are very common.[6,23] Yet data and program are equivalent; indeed, all the code examined for this paper includes "magic numbers" — data and assumptions written into program code. It is curious, then, that program code is not critiqued, particularly as typical code has less certain provenance than data, which is generally well-documented.

Bugs can be understood as discrepancies between what code is intended to do and what it actually does. Many bugs cause erroneous results, but bugs may be "fail safe" by causing a program to crash so no incorrect result can be delivered. Better, contracts and assertions are essential defensive programming technique that block compilation or block execution with incorrect results; they turn bugs into safe termination (see Supplementary Information). None of the code examined for this paper includes either.

If code is not documented it cannot be clear what it is intended to do, so it is not even possible to detect and eliminate bugs. Indeed, even with good documentation, *intentional bugs* will remain, that is, code that correctly implements the wrong things. For instance, in numerical modelling, using an inappropriate method can introduce errors that are not "bugs" in the sense of incorrectly implementing what was wanted (e.g., ill-conditioning), but are bugs in the sense of producing incorrect results — that is, what was wanted was naïve. Similarly, misuse of random numbers (e.g., using standard libraries without testing them) is a common

cause of bugs.[24]

Following the Dunning-Kruger Effect,[25] unqualified programmers over-estimate their programming skills because they do not have the skills to recognise their lack of knowledge. Dunning and Kruger say,

> "People usually choose what they think is the most reasonable and optimal option [. . .] The failure to recognise that one has performed poorly will instead leave one to assume that one has performed well; as a result, the incompetent will tend to grossly overestimate their skills and abilities. [. . .] Not only do these people reach erroneous conclusions and make unfortunate choices, but their incompetence robs them of the metacognitive ability to realise it."

Unlike many skills (skating, brain surgery, . . .) programming is one where errors can go unnoticed for long periods of time. The worse programmers are, the more trivial bugs they tend to make. Trivial bugs are easy to find so, ironically, being a poorprogrammer *increases* one's self-assessment because debugging seems very productive. It is easy for poor programmers and their associates to believe they are better than they actually are.

It sounds harsh to call programmers incompetent, but challenged with the complexity of programs and the complexity of the domains programs are applied in, we are all incompetent and succumb to the limitations of our cognitive resources, creating blindspots in our thinking.[26] We *all* make mistakes we are unaware of, and for all sorts of reasons. We therefore generally have a higher opinion of our own competence than is justified.

Computers themselves are also a part of the problem. Naïvely modifying a program can make it more complex, and hence less scrutable. In theory, programs can be written so that it is not possible to determine what they do or how they do it (whether deliberate obfuscation or accidentally), except by running them, if indeed it is possible to exactly reproduce the necessary context to do so.[27] The point is, introducing bugs has to be avoided in the first place.

The Call to Action, next, together with this paper's Supplementary Information provides solutions.

Call to action

Computer programs are the laboratories of modern scientists, and should be used with a comparable level of care that virologists use in their laboratories — lab books and all[21] — and for the same reasons: computer bugs accidentally cultured in one laboratory can infect research and policy worldwide. Given the urgency of rigorously understanding COVID-19, any epidemic for that matter, it is essential that epidemiologists engage professional software engineers to help develop reliable laboratory methodologies. For instance, code lab books can be generated and signed easily.

Software used for informing public health policy, medical research or other medical applications is *critical software*. Professional critical software development, as used in aviation

and the nuclear power industry, is (put briefly) based on *correct by construction*:[28] effectively, design it right first time, supported by numerous rigorous techniques to manage error. (See this paper's Supplementary Information.) Not coincidentally, these are *exactly* the right methods to ensure code is both dependable and scrutable. Conversely, not following these practices undermines the rigour of the science.

An analogous situation arises in ethics. There are some sorts of research that are ethically unacceptable, but few people have the ethical expertise to make sound ethical judgements — particularly when it comes to assessing their own work. Misuse of data, exploiting vulnerable people, and not obtaining consent are typical problems. National funders, and others, therefore require Ethics Boards to formally review ethical quality. Medical journals will not publish research that has not undergone ethical review.

Analogously, and supplementing Ethics Boards, Software Engineering Boards would authorise as well as provide advice to guide the implementation of high-quality programming. Just as journals require conflicts of interest statements, data availability statements, and ethics board clearance, we must move to epidemic modelling papers — and in due course, all scientific papers — being required to include Software Engineering Board clearance statements as appropriate. Interestingly, Software Engineers have a code of ethics that applies to *their* work in epidemic modelling.[29]

There need to be many Software Engineering Boards (SEBs) to ensure convenient access and oversight, certainly at least one per university. Active, senior, professors of software engineering should be on these SEBs; this is not a job for people who are not qualified in the area or not actively connected with the true state of the art. There are many high-quality software companies (especially those in safety-critical areas like aviation and nuclear power) who would be willing to help.

Open Source generally improves the quality of software. SEBs will take account of the fact that open source software enables contributors to be located anywhere, typically without a formal contractual relationship with the leading researchers. Where appropriate, then, SEBs might require *local* version control, unit testing, static analysis and other quality control methods for which the lead researcher remains responsible. See the Supplementary Information for further suggestions.

A potential argument against SEBs is that they may become onerous, onerous to run and onerous to comply with their requirements. A more mature view is that SEBs need their processes to be adaptable and proportionate. If software being developed is of low risk, then less stringent engineering is required than if the software could cause frequent and critical outcomes, say in their impact on public health policy for a nation. Hence SEBs processes are likely to follow a risk analysis, perhaps starting with a simple checklist. There are standard ways to do this, such as following IEC 61508:2010[30,31] or similar. Following a risk analysis, the Board would focus scrutiny where it is beneficial without obstructing routine science.

A professional organisation, such as the UK Royal Academy of Engineering ideally working in collaboration with other national international bodies such as IFIP, should be asked to develop and support a framework for SEBs. SEBs could be quickly established to provide

direct access to mature software engineering expertise for both researchers and for journals seeking competent peer reviewers. In addition, particularly during a pandemic, SEBs would provide direct access to their expertise for Governments and public policy organisations. Given the urgency, this paper recommends that *ad hoc* SEBs should be established for this purpose.

Further issues are discussed in the Supplementary Information.

Conclusions

The challenge to epidemic modelling, then, is to manage software development to reduce the impact of bugs and poor programming practices. Epidemic models should be explicit on their methodologies, limitations and weaknesses,[2] and software methodologies should not be ignored.

Unfortunately, while programming is easy, and is often taken for granted, programming *well* is very difficult.[26] Programming well is essential for modelling epidemics. Simply put, without quality code and data, research is not open to scrutiny, let alone proper review. Many would argue that availability of code and data ensure research is reproducible, but that is a very weak criterion: both need to be documented and clear enough to be *challengeable*.[18,22,32] Current epidemiological coding practice is an alarming basis for public policy.

We must prioritise getting appropriate professional software engineering skills and resources to bear on the COVID-19 pandemic without delay. While this paper's Supplementary Information summarises relevant software engineering practice, Software Engineering Boards are a constructive and practical way to support and improve computer-based epidemic modelling.

Future work should extend improving software standards in other areas, particularly in healthcare.[26,33]

References

[1] Richards, D., Enrique, A., Eilert, N. *et al*. "A pragmatic randomized waitlist-controlled effectiveness and cost-effectiveness trial of digital interventions for depression and anxiety" *Nature Digital Medicine* 2020; textbf3(85). DOI: 10.1038/s41746-020-0293-8

[2] Whitty, C. J. M., "What makes an academic paper useful for health policy?" *BMC Medicine* 2015; **13**:301. DOI: 10.1186/s12916-015-0544-8

[3] Shneiderman, B., "Opinion: The dangers of faulty, biased, or malicious algorithms requires independent oversight," *Proceedings National Academy of Sciences* 2016; **113**(48)13538–13540. DOI: 10.1073/pnas.1618211113

[4] Ladkin, P. B., Littlewood, B., Thimbleby, H. & Thomas, M., "The Law Commission presumption concerning the dependability of computer evidence," *Digital Evidence and*

Electronic Signature Law Review 2020; **17**. DOI: 10.14296/deeslr.v17i0.5143 (NB use http://journals.sas.ac.uk/deeslr/article/view/5143 until the DOI is resolved.)

[5] Heesterbeek, H., Anderson, R. M., Andreasen, V. *et al*, "Modeling infectious disease dynamics in the complex landscape of global health," *Science* 2015; **347**(6227):aaa4339. DOI: 10.1126/science.aaa4339

[6] Wynants, L., van Calster, B., Bonten, M. M. J. *et al*, "Prediction models for diagnosis and prognosis of covid-19 infection: systematic review and critical appraisal," *BMJ* 2020; **369**:m1328. DOI: 10.1136/bmj.m1328

[7] Moons, K. G., Altman, D. G., Reitsma, J. B. *et al*, "Transparent Reporting of a multivariable prediction model for Individual Prognosis or Diagnosis (TRIPOD): explanation and elaboration," *Annals of Internal Medicine* 2015; **162**(1):W1–73. DOI: 10.7326/M14-0698

[8] Adam, D., "Modelling the pandemic: The simulations driving the world's response to COVID-19," *Nature* 2020; **580**:316–318. DOI: 10.1038/d41586-020-01003-6

[9] Ferguson, N. M., Laydon, D., Nedjati-Gilani, G. *et al*, "Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand," Downloaded from http://www.imperial.ac.uk/media/imperial-college/medicine/sph/ide/gida-fellowships/Imperial-College-COVID19-NPI-modelling-16-03-2020.pdf, 16 March 2020.

[10] Ferguson, N. M., Cummings, D. A. T., Fraser, C., Cajka. J. C., Cooley, P. C. & Burke, D. S., "Strategies for mitigating an influenza pandemic," *Nature* 2005; **437**:209–214. DOI: 10.1038/nature04017

[11] Ferguson, N. M., Cummings, D. A. T., Fraser, C. *et al*, "Strategies for mitigating an influenza pandemic," *Nature* 2006; **442**:448–452. DOI: 10.1038/nature04795

[12] Ferguson, N., http://twitter.com/neil_ferguson/status/1241835454707699713, 22 March 2020.

[13] Leake, J., "Neil Ferguson interview: No 10's infection guru recruits game developers to build coronavirus pandemic model," *The Sunday Times*, 29 March 2020. http://www.thetimes.co.uk/article/neil-ferguson-interview-no-10s-infection-guru-recruits-game-develope

[14] Ferguson, N., http://twitter.com/neil_ferguson/status/1241835456947519492, 22 March 2020.

[15] Richards, D. & Boudnik, K., "Neil Ferguson's Imperial model could be the most devastating software mistake of all time," *The Telegraph*, 16 May 2020, http://www.telegraph.co.uk/technology/2020/05/16/neil-fergusons-imperial-model-could-devastating-software-mistake

[16] Sander, B., Nizam, A., Garrison, L. P. Jr *et al*, "Economic evaluation of influenza pandemic mitigation strategies in the us using a stochastic microsimulation transmission model," *Value Health* 2009; **12**(2):226–233. DOI: 10.1111/j.1524-4733.2008.00437.x

[17] Zlojutro, A., Rey, D. & Gardner, L., "A decision-support framework to optimize border control for global outbreak mitigation," *Nature Scientific Reports* 2019; **9**:2216. DOI: 10.1038/s41598-019-38665-w

[18] Thimbleby, H. & Williams, D., "A tool for publishing reproducible algorithms & A reproducible, elegant algorithm for sequential experiments," *Science of Computer Programming* 2018; **156**:45–67. DOI: 10.1016/j.scico.2017.12.010 Also on GitHub: http://GitHub.com/haroldthimbleby/relit

[19] Kissler, S. M., Tedijanto, C., Goldstein, E. *et al*, "Projecting the transmission dynamics of SARS-CoV-2 through the postpandemic period," *Science* 2020; DOI: 10.1126/science.abb5793

[20] Verity, R., Okell, L. C., Dorigatti, I. *et al*, "Estimates of the severity of coronavirus disease 2019: a model-based analysis," *Lancet* 2020; DOI: 10.1016/S1473-3099(20)30243-7

[21] Schnell, S., "Ten Simple Rules for a Computational Biologist's Laboratory Notebook," *PLoS Comput Biology* 2015; **11**(9):e1004385. DOI: 10.1371/journal.pcbi.1004385

[22] Popper, K. R., *Conjectures and Refutations: The Growth of Scientific Knowledge*, 2nd edition, Routledge, 2002.

[23] Sayburn, A., "Covid-19: Experts question analysis suggesting half UK population has been infected," *BMJ* 2020; **368**:m1216. DOI: 10.1136/bmj.m1216

[24] Knuth, D. E., *The Art of Computer Programming*, **2** (Seminumerical algorithms), 3rd ed, Addison-Wesley, 1998.

[25] Kruger, J. & Dunning, D., "Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments," *Journal of Personality and Social Psychology* 1999; **77**(6):1121–1134. DOI: 10.1037/0022-3514.77.6.1121

[26] Thimbleby, H., *Fix IT: How to solve the problems of digital healthcare*, Oxford University Press, 2020.

[27] Thimbleby, H., Anderson, S. O. & Cairns, P., "A Framework for Modelling Trojans and Computer Virus Infection," *Computer Journal* 1999; **41**(17):444–458. DOI: 10.1093/comjnl/41.7.444

[28] Woodcock, J. C. P., Larsen, P. G., Bicarregui, J. C. & Fitzgerald, J. S., "Formal methods: Practice and experience," *ACM Computing Surveys* 2009; **41**(4):19. DOI: 10.1145/1592434.1592436

[29] *ACM Code of Ethics and Professional Conduct*, `http://www.acm.org/code-of-ethics`, accessed 23 April 2020.

[30] Redmill, F., "Understanding the Use, Misuse and Abuse of Safety Integrity Levels," revised, *Lessons in System Safety*, Eighth Safety-critical Systems Symposium, Redmill, F. & Anderson, T., eds., 2000. `http://homepages.cs.ncl.ac.uk/felix.redmill/publications/1%20SILs.pdf`

[31] International Electrotechnical Commission (IEC), *IEC 61508:2010 CMV Commented version*, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2010. `http://webstore.iec.ch/publication/22273`

[32] Benureau, F. C. Y. & Rougier, N. P., "Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific," *Frontiers in Neuroinformatics* 2017; **11**:69. DOI: 10.3389/fninf.2017.00069

[33] Zhang, Y., Masci, P., Jones, P. & Thimbleby, H., "User Interface Software Errors in Medical Devices," *Biomedical Instrumentation & Technology* 2019; **53**(3):182–194. DOI: 10.2345/0899-8205-53.3.182

Acknowledgements

Ross Anderson, Nicholas Beale, Ann Blandford, Paul Cairns, Rod Chapman, José Corréa de Sà, Paul Curzon, Jeremy Gibbons, Richard Harvey, Will Hawkins, Ben Hocking, Daniel Jackson, Peter Ladkin, Bev Littlewood, Paolo Masci, Stephen Mason, Robert Nachbar, Martin Newby, Patrick Oladimeji, Claudia Pagliari, Simon Robinson, Jonathan Rowanhill, John Rushby, Prue Thimbleby, Will Thimbleby, Martyn Thomas, and Ben Wilson provided very helpful comments.

This work was jointly supported by See Change (M&RA-P), Scotland (an anonymous funder), by the Engineering and Physical Sciences Research Council [grant number EP/M022722/1], and was supported by the Royal Academy of Engineering through the Engineering X Pandemic Preparedness programme. The funders had no involvement in the research or in the writing of this paper.

Author contributions

Harold Thimbleby is the sole author.

Competing interests

The author declares no competing interests.

Additional information

Supplementary Information is available for this paper.

An earlier version of this paper was submitted to the UK Parliamentary Science and Technology Select Committee's inquiry into UK Science, Research and Technology Capability and Influence in Global Disease Outbreaks, under reference LAS905222, 7 April, 2020.

Supplementary Information

A call for professional software engineering to be used in scientific research (and epidemic modelling in particular)

Harold Thimbleby
harold@thimbleby.net

Further issues for Software Engineering Boards

Software Engineering Boards, henceforth SEBs, will be used to ensure that critical code, including epidemic modelling, is of high standard, to provide assurance for scientific papers, Government public health and other policies, etc, that the code used is of appropriate quality for its intended uses.

Further details of the SEB proposals are in the main paper. Here we raise some issues, potential limitations and possible responses that can be addressed over time:

1. Until there are national qualifications, nobody — certainly nobody without professional training in software — really knows just how bad (or good) they are at software engineering.

2. Although SEBs may start with a checklist approach, like Ethics Boards generally do, it cannot be assumed that people approaching SEBs know enough about software engineering to perform adequate software assessments when there is any risk (as there is in public policy, medical apps, and so on). SEBs may also provide mentoring and training.

3. Unlike Ethics Boards, which provide hands-off oversight, SEBs should provide professional advice, perhaps providing training or actually helping hands-on develop appropriately reliable software. During a pandemic Boards would be very willing to do this, but in the long run it is not sustainable as voluntary labour, so all research, particularly medical research, should include support for professional software engineering.

4. Ethics Boards typically require researchers to fill in forms and provide details, which is a feasible approach as researchers know if they are doing experiments on children, for instance, so the forms are relatively easy to fill in (if often quite tedious). On the other hand, few healthcare and medical researchers understand software and programming, so they are *not* able to fill in useful software forms on their own. SEBs need to know how well engineered the software really is, not how good its developers *think* it is. As typical programs are enormous, SEBs are either going to need resources to evaluated programs, or they to need to supervise independent bodies that can do it for them.

5. Like some Ethics Boards, SEBs might become, or be perceived as becoming, onerous and heavy handed. It is essential that SEBs have (and perhaps are chaired by) experienced, professional software engineers to avoid this problem.

6. When code is taken seriously, concerns may be raised on programmers' contributions to research, intellectual property rights, and co-authoring.[34] Software engineering is a hard, creative discipline, and getting epidemiological models to work is generally a significant challenge, on a par with the setting up and exploring the mathematical models. Often the software engineers will be solving entirely new problems and contributing to the research. How this is handled needs exploring. How software engineers are appropriately credited and cited for their contributions also needs exploring.

7. SEBs require policies on membership, transparency, and accountability.

8. There should be a clear separation between the SEB members' activities as part of the Board, and their other activities, including professional advice, code development, or training (which is likely to be in demand from the same people who require formal approvals from the SEBs).

9. Professional Engineering Bodies have a central role to play in professionalism, ranging from education and accreditation to providing professional structures for SEBs. For example, should and if so how should the programming skills taught to computational scientists (epidemiologists, computational biologists, economists, computational chemists, . . .) be accredited?

10. In the main paper, SEBs are viewed as a constructive contribution to good science, specifically helping improve the quality of epidemiological modelling. More generally, SEBs will have wider roles, for instance in overseeing software subject to medical device regulation.[26]

There are other ideas to help make SEBs work, but it is clear they are part of the solution and we must not let perfection be the enemy of the good. SEBs don't need to be perfect on day one, but they do need to get going in some shape or form to start making their vital contribution.

Software engineering practices for pandemic modelling

This Supplementary Information provides more explanations and justification for the following standard software engineering practices that support reliable modelling, reliable research, and, most generally, reliable science.

The reader is referred to standard textbooks for more information.[35,36] The book *Why Programs Fail*[37] is a very good practical guide to developing better code, and may be found more accessible. Humphrey[38] outlines a thorough discipline for anyone wanting to become a good programmer. Improvement is such an important activity, Humphrey has also published a book to persuade managers of the benefits.[39]

Software Engineering includes the following, which are discussed at more length below:

- (***0***) define requirements: how reliable does code have to be at what cost?
- (***1***) formal methods and tool use,
- (***2***) defensive programming,
- (***3***) dependable programming languages,
- (***4***) version control and open source,
- (***5***) rigorous testing,
- (***6***) good documentation and record keeping,
- (***7***) usability,
- (***8***) reusing quality-assured solutions,
- (***9***) simplicity,
- (***10***) explicit compliance with standards,
- (***11***) and more factors, such as security ...
- (***12***) effective teamwork.

(*0*) Without defining requirements, not enough skilled effort will be put into designing and implementing reliable software — or excess effort will be wasted.

It is not always necessary to program well if the code to be produced is for fun, experimenting, or for demonstrations. On the other hand, if code is intended for life-critical applications, then it is worth putting more engineering effort into it. The first step of software engineering, then, is to assess the requirements, specifically the reliability requirements of the code that is going to be produced.

In practice, requirements and expectations change. Early experimental code, developed informally, may well be built on later to support models intended to inform public policy, for instance. Unfortunately, prototypes may impress project leaders who then want to rush into

production software because, it seems, "it obviously works." Fortunately, best practice software engineering can be adopted at any stage, particularly by using *reverse engineering*. In reverse engineering, one carefully works out (generally partly automatically) what has already been implemented. This specification, carefully reviewed, is then used as the basis for a more rigorous software engineering process that implements a more reliable version of the system.

(*1*) Without formal methods, there is no rigorous and checked specification of a program, so nobody — including its developers — will know exactly what it is supposed to do.

Formal methods require sophisticated knowledge of logic,[28] as well as practical knowledge of using appropriate formal methods tools (Alloy, HOL, PVS, SPARK, VDM, Z, and many others). Using the right tools is essential for reliable programming, because the tools do quickly and reliably what, done by hand, would be slow and error-prone. Standard tools cover verification, static analysis of code, version control, documentation, and so on — this paper explains why some of these activities are essential for reliable programming below. Crucially, tools are designed to catch common human errors that we are all prone to.

Formal methods have the huge advantage that they "think differently" and therefore help uncover design problems and bugs that can be found in no other way. Because formal methods are logical, mathematical theories (safety properties, etc) can be expressed and checked (often automatically), which provides a very high degree of fault tolerance (e.g., redundancy), and hence good reasons to believe the quality of the final code — that it does what it is supposed to do. Unfortunately, because formal methods are mathematical, few programmers have experience using them.

(*2*) Without defensive programming, any errors — in data, code, hardware, or in use — will go unnoticed and be uncorrected.

Defensive programming is based on a range of methods, including error checking, independent calculation (using multiple implementations written by independent programmers), assertions, regression testing, etc. Notoriously, what are often unconsciously dismissed as trivial concerns frequently lead to the hardest to diagnose errors, such as buggy handling of "well-known, trivial" things like numbers.[40] The great advantage of defensive programming is that it detects, and may be able to recover from, bugs that have been missed earlier in the development process (such as typos in the code). Defensive programming requires professional training to be used effectively, for example it is not widely known that some choices of programming language make defensive programming unnecessarily hard.[41]

A special case of defensive programming appropriate for pandemic modelling is mixing methods. Do not rely on one programming method, but mix methods (e.g., different numerical methods) to use and compare multiple approaches to the modelling.

(*3*) Using inappropriate programming languages undermines reliability.

Many popular languages are popular because they are easy to use, which is not the same as being reliable to use. The fewer constraints a language imposes, the easier it *seems* to be to program in, but the lack of constraints means the language cannot provide the checks stricter languages do. C, for instance, which is one of the languages widely used for modelling,[12,42] is not a good choice for a reliable programming language — it has many intrinsic weaknesses that are well-known to professionals, but which frequently trap inexperienced programmers. (This is not the place for a review[41] but Excel is even worse in this regard.) In particular, C is not a portable language (unless extreme care is taken), which means models will work differently on different types of computer. SPARK Ada is one example of a much more appropriate programming language to use. SPARK Ada also has the advantage that most Ada programmers are better qualified than most C programmers.

(*4*) Version control and open source organises and helps software development.

It is appreciated that the models may change and be adapted as new data and insights become available. Changing models makes it even harder to ensure that they are correct, and thus emphasises the relevance of the core message this paper: we have to find ways to make computer models more reliable, inspectable, and verifiable. Version control keeps a record of what code was used when, and enables reconstruction of earlier versions of code that has been used. Version control is supported by many tools (such as Git, Subversion, etc).

If version control is not used, one has no idea what the current program actually is. Version control is essential for *reproducibility*:[32] it enables efforts to duplicate work to start with the exact version that was used in any published paper, provided that the published paper discloses the version and a URL for the relevant repository. Note that version control should also be used for data and web site data used by code, otherwise the results reported are not replicable.

If results cannot be reproduced, has anything reliable been contributed? When a modelling paper presents results from a model, it is important to reproduce those results without using the same code. Better still, research should be reproduced without sharing libraries or APIs (for example, results from a model using R might be reproduced using Mathematica). Reproducing the same results relying on the same codebase tells you little. The more independent reproductions of results the greater the evidence for belief in the implications.

Clearly, with the transformations a program from avian flu in Thailand[10] to COVID-19 in the United States and in Great Britain[9] taking place over many years, version control would have been very helpful to keep proper track of the changes. Note that professional version control repositories also provide secure off-site back up, ensuring the long-term access to the code and documentation — this would avoid loss of supplementary information problems, as occurred in.[16]

Most version control systems would, in addition, enable open source methods so the code could be shared — and reviewed — by a wider community. Open source is not a panacea, however; it raises many trade-offs. Particularly for world-wide concerns like pandemic modelling, it increases diversity in the software developers, and fosters a diverse scientific collaboration. Open source can raise people's standards — some countries[43,44] are using Excel models to manage COVID-19, and open source projects properly implemented would help these people enormously.

Open source raises important licensing and management questions to ensure the quality of contributions. A salutary open source case is NPM, where lawyers from a company called Kik triggered Azer Koçulu, that is, a *single* programmer, to remove all his code from a repository. This caused problems to many thousands of JavaScript programmers worldwide who could no longer compile anything — ironically, including Kik itself.[45]

Critically in the case of epidemic modelling, open source democratises the model development and interpretation, and enables properly-informed public debate. Note that many (if not most) successful open source projects have had a closed team of highly dedicated and well-paid developers.

(*5*) Without testing, there is no evidence that a program works under real conditions.

In poorly-run software development it is very easy to miss bugs, because the flawed thinking that inserted bugs in the code is going to be the same flawed thinking with the same misconceptions that tries to detect them. Rigorous testing includes methods like fault injection. Here, the idea is that if testing finds no bugs, that may be because the testing is not rigorous enough rather than that the program actually has no bugs. Fault injection inserts random bugs, and then testing gives statistical insights into the number of bugs in a program (depending on how many deliberate bugs it successfully finds). There are many other important testing methods.[35,36]

(*6*) Without documentation and record keeping, nobody — least of all the programmer — knows what code is supposed to do or how to get it to do it.

Documentation covers internal documentation (how code works), developer (how to include it in other programs), configuration (how to configure and compile the code in different environments), external documentation (how the code is used), and help (documentation available while using the program).

For scientific purposes, perhaps the most important form is internal documentation: how to understand how the code works. This is different from developer documentation, which is how to use the code in other programs. For example, code for solving a differential equation needs explaining — what method does it use, what assumptions does it have? In contrast, the developer documentation for differentiation would say things like it solves ordinary differential equations with parameters e for the function f with the independent variable x in

the interval $[u, v]$, or whatever, but *how* it solves equations is of little interest to the developer who just needs to use it. How code works — internal documentation — is essential for the epidemiologist, or more generally any scientist. An example of a simple SIR epidemiological model's internal documentation can be found at
`http://www.harold.thimbleby.net/sir`

There are many tools to help manage documentation (Javadoc, Doxygen, ...). Literate programming is one very effective way of documenting code, and has been used for very large programming projects.[46] Literate programming has also been used directly to help publish clearer and more rigorous papers based on code[18] — a paper that also includes a wider review of the issues.

Documentation should be supplemented by details of algorithms and proofs of correctness (or references to appropriate literature). All the documentation needs to be available to enable others to correctly download, install and correctly use a program — and to enable them, should they wish, to repurpose it reliably for their own work. In addition, documentation requires specifications and, in turn, *their* documentation.

A important role of documentation is to cover configuration: how to get code to work — without configuration, code is generally useless. The most basic is a README file, which explains how to get going; more useful approaches to configuration include make files, which are programs that do the configuration automatically.

Without proper record keeping, code becomes almost impossible to maintain if programmers leave the project. Note that computer tools can make record keeping, laboratory books etc, trivial — if they are used.

(*7*) If code is not usable, even if it is "correct," it will not be used and interpreted correctly.

Usability is an important consideration:[47,48] is the program usable by its intended users so they can obtain correct results? Often the programmers developing code know it so well they misjudge how easy it will be for anyone else to use it — this is a very serious problem for the lone programmer (possibly working in another country) supporting a research team. Usability is especially important when programs are to be used by other researchers and by non-programmers, including epidemiologists.

(*8*) Without using existing solutions (libraries, APIs, etc) reinventing code merely reinvents bugs.

Reusing quality code (mathematical functions, database operations, user interface features, connectivity, etc) avoids having to develop it oneself, saves time and avoids the risks of introducing new bugs. The more code is reused, the more likely many people will have contributed to improving it — for example, reusing a standard database package will provide Atomicity, Consistency, Isolation, and Durability (so-called ACID properties) without any

further work (nor even needing to understanding what useful guarantees these basic properties ensure). Note that reusing code assumes the originators of the code followed good software engineering practice; equally, if the code being developed follows good software engineering practice, it too can be shared. Its quality improves through having scrutiny by the wider community, and in successful cases, leading to consensus on the methods. Indeed, reuse, scrutiny, consensus are the foundations of good science.

A special case of reuse is to use software tools to help with software development. The tools (if appropriately chosen) have been carefully developed and widely tested. Tools enable software developers to avoid or solve complex programming problems repeatedly and with ease.

(*9*) Poor programmers often fix bugs rather than the causes of bugs: complexity and obfuscation.

When a program doesn't quite do what is wanted, it is tempting to add more features or variables, or to treat the problem as an "exception" and program around it — which inserts more code and, almost certainly, more bugs. This way lies over-fitting, a problem familiar from statistics (and machine learning). Programs can be made over-complex and they can then do anything; an over-complex program may seem correct by accident. Instead, the hallmarks of good science are that of parsimony and simplicity; if a simple program can do what is needed it is more likely to be correct. A simpler program is easier to prove correct, easier to program, and easier to debug. A special case of needing simplicity is when fixing bugs: instead of fixing bugs one at a time, one should be fixing the *reasons* why the bugs have happened. Generally, when bugs are fixed, programmers should determine *why* the bugs occurred, and thence repair the program more strategically.

(*10*) International standards have been developed to support critical software development.

To ensure adherence to best practice and, importantly, to avoid being unaware of relevant methodologies, professional software development projects adopt and adhere to relevant standards, such as ISO/IEC/IEEE 90003:2018.[49] However, for safety-critical models or models of national policy significance, much stronger standards such as aviation software standards, such as RTCA DO-178C/EUROCAE ED-12C,[50] commonly called DO-178C, will be more appropriate. Publications should then cite the standards to which their computer models comply.

Note that medical device regulation, which has its own standards, is lagging behind professional software engineering practice, and currently provides no useful guidance for critical software development.[26]

(*11*) Other factors ....

Of course, there are many other factors to be considered for the development of critical code, such as using appropriate methods to ensure cybersecurity,[51,52] particularly while also being able to download secure updates.

For pandemic modelling specifically, understanding the limitations of numerical methods (in particular, how numerical methods are affected by the choice of programming language) is critical. Hamming is considered a classic,[53] but there is a huge choice available.

For reasons of space, the present paper does not discuss the issues raised by AI, nor the many very important, non-trivial social concerns, which have complex implications for professional software engineering, such as managing programming teams, data ethics, privacy, legal liability,[54] and software as a matter in law, as in disputes over model results.[55]

(*12*) Effective teamwork is essential because no individual has the capacity to develop non-trivial reliable software.

As this long list illustrates, Software Engineering is a complex and wide-ranging subject. Software engineering cannot be done effectively by individuals working alone, and that is without considering the complexities of the domain (in the present case, including pandemic modelling, mathematical modelling, public health policy, etc). Teamwork is essential.

Modern software is complex, and no one person can have the skills to understand all relevant aspects of all but the most trivial of programs. Furthermore, programming is a cognitively demanding task, and causes loss of situational awareness (that is, cognitive "overload" making one unable to track requirements beyond those thought to be directly related to the specific task in hand). The main solution to both problems is teamwork, to bring fresh insights, different mindsets and skills to the task.

Peer review of code is an essential teamwork practice in reliable program development:[36,56] it is easy to make programming mistakes that one is unaware of, and an independent peer review process is required to help identify such unnoticed errors.

Almost all software will be used by other people, and user interface design is the field concerned with developing usable and effective software. A fundamental component of user interface design is working with users and user testing: without engaging users, developers are very likely to introduce quirks that make systems less usable (often less safe) than they should be. In short, users have to be brought into the software team too.

Additional references for Supplementary Information

References numbered 1–33 appear in the reference list in the main paper.

[34] Vancouver Group, "Uniform requirements for manuscripts submitted to biomedical journals," *JAMA* 1997; **277**(11):927–934. DOI: 10.1001/jama.1997.03540350077040

[35] Sommerville, I., *Software Engineering*, 10th edition, Pearson, 2015.

[36] Knight, J., *Fundamentals of Dependable Computing for Software Engineers*, CRC Press, 2012.

[37] Zeller, A., *Why Programs Fails: A Guide to Systematic Debugging*, Morgan Kaufmann, 2006.

[38] Humphrey, W. S., *PSP: A Self-Improvement Process for Software Engineers*, Addison Wesley, 2005.

[39] Humphrey, W. S., *Winning with Software: An Executive Strategy*, Addison-Wesley Professional, 2001.

[40] Thimbleby, H. & Cairns, P., "Interactive numerals," *Royal Society Open Science* 2017; **4**(4):160903. DOI: 10.1098/rsos.160903

[41] Thimbleby, H., "Heedless Programming: Ignoring Detectable Error is a Widespread Hazard," *Software — Practice & Experience* 2012; **42**(11):1393–1407. DOI: 10.1002/spe.1141

[42] Chao, D. L., Halloran, M. E., Obenchain, V. J. & Longini, I. M. Jr, "FluTE, a Publicly Available Stochastic Influenza Epidemic Simulation Model," *PLOS Computational Biology* 2010; **6**(1):e1000656. DOI: 10.1371/journal.pcbi.1000656 Source code available at `http://GitHub.com/dlchao/FluTE`

[43] Abir, M., Nelson, C., Chan, E. W. *et al*,"RAND Critical Care Surge Response Tool: An Excel-Based Model for Helping Hospitals Respond to the COVID-19 Crisis," RAND Corporation, 2020. `http://www.rand.org/pubs/tools/TLA164-1.html`

[44] Alvarez, N. M., Gonzalez-Gonzalez, E. & Trujillo-de Santiago, G., "Modeling COVID-19 epidemics in an Excel spreadsheet: Democratizing the access to first-hand accurate predictions of epidemic outbreaks," *MedRxiv* (preprint) 2020. DOI: 10.1101/2020.03.23.20041590

[45] NPM Blog, 2016; 23 March. Accessed 10 April 2020. `http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm`

[46] Knuth, D. E., *Literate programming*, Center for the Study of Language and Information Publication Lecture Notes, *Lecture Notes* 1992; **27**.

[47] Shneiderman, B., Plaisant, C., Cohen, M. *et al*, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 6th edition, Pearson, 2016. `http://www.cs.umd.edu/hcil/DTUI6`

[48] Thimbleby, H., *Press On: Principles of Interaction Programming*, MIT Press, 2007.

[49] International Organization for Standardization (ISO), *Software engineering — Guidelines for the application of ISO 9001:2015 to computer software*, 2015. `http://www.iso.org/standard/74348.html`

[50] RTCA, *DO-178 — Software Considerations in Airborne Systems and Equipment Certification*, `http://my.rtca.org/NC__Product?id=a1B36000001IcmwEAC`

[51] Anderson, R., *Security Engineering*, Wiley, 3rd. edition, 2020.

[52] Shostack, A. & Zurko, M. E., "Secure Development Tools and Techniques Need More Research That Will Increase Their Impact and Effectiveness in Practice," *Communications of the ACM* 2020; **63**(5):39–41. DOI: 10.1145/3386908

[53] Hamming, R. W., *Numerical Methods for Scientists and Engineers*, Dover Publications Inc., 1987.

[54] Schneier, B., *Click Here To Kill Everybody — Security and Survival in a Hyper-connected World*, W. W. Norton & Company, Inc, 2018.

[55] Mason, S. & Seng, D., *Electronic Evidence*, 4th edition, Humanities Digital Library, 2017. DOI: 10.14296/517.9781911507079

[56] Baum T, Leßmann H & Schneider K, "The Choice of Code Review Process: A Survey on the State of the Practice," Product-Focused Software Process Improvement: 18th International Conference, *Lecture Notes in Computer Science* 2017; **10611**:111–127. DOI: 10.1007/978-3-319-69926-4_9

Data summary for recent papers

Legend:

- * code may be available on request
- ** some or all code directly accessible
- † only trivial comments
- ‡ helpful comments explaining code, explaining intent rather than rephrasing the code
- § uses code repository (typically GitHub)
- Δ uses data repository (typically Dryad, Figshare) for data or code

Ref	Availability of data	Availability of code

1	On request	*On request (requested)
2	On GitHub	**†§On GitHub, trivial comments, no documentation
3	Following Twitter policy, data used in the generation of the algorithm is not available	Due to the sensitive and potentially stigmatizing nature of this tool, code not available
4	On request	*On request (requested)
5	Nothing available	Nothing available
6	Not available due to HIPAA compliance agreement but are available on reasonable request. Plans an open-source release of data and ML model summer 2020	**†§On GitHub, poor commenting, no documentation
7	Not available due to patient privacy and lack of informed consent	No code available
8	Not available due to institutional restrictions on data sharing and privacy concerns	§Empty GitHub repository
9	Not available as they have not been legally certified as de-identified. Data may be available by the time of publication by request	**†§On GitHub, trivial comments, no documentation
10	Not available due to restrictions in the ethical permit, but may be available on request	**†§On GitHub, trivial comments, no documentation
11	No data	Minor documentation in supplementary material, no code
12	Not available due to patient privacy	**†§Some code on GitHub, minor comments, no documentation
13	ΔData available on Dryad	**‡Code and samples available in Rmarkdown
14	Restricted access – not available	Restricted access – not available
15	ΔData available on Dryad	*‡Code available. Minor comments
16	ΔData available on Dryad	*‡Commented code on Dryad
17	On request	*†R lightly commented
18	No data required	*†Unrunnable code fragment
19	Data embedded in PDF	No code available
20	ΔData available on Dryad	*‡Some code in Matlab
21	ΔPartial data on Dryad	**‡§On GitHub, documented R, including manual

22	No data required	No code provided
23	Data cited, not all available	**†§On GitHub. Trivial documentation
24	△On Figshare	**On Figshare, large amount of disorganised and undocumented code
25	△Data on Dryad	No code provided
26	Data on various web sites	No code available
27	Data on request	"The coding used to train the artificial intelligence model are dependent on annotation, infrastructure, and hardware, so cannot be released." (!) Algorithm (not code) available on request.
28	Data on request	*Python scripts can be requested
29	△Unspecified location on large website requiring registration	**§On GitHub, some documented Matlab code
30	Available to researchers who meet criteria for access to confidential data	Not available
31	Data conditional on approved study proposal	**§On GitHub, uncommented
32	Data on large websites	**§On GitHub, uncommented

Summary of data

Number of papers relying on code	=	32	100%
Have some or all code accessible	=	12	37%
Use code repository, e.g., GitHub (1 was empty)	=	11	34%
Helpful comments	=	5	16%
Trivial comments	=	8	25%
No or uninsightful comments (e.g., copyright)	=	19	59%
Some or all code on request	=	8	25%
No code available	=	12	37%
Use data repository, e.g., Dryad	=	8	25%

Sampled papers

[1] Callahan, A., Steinberg, E., Fries, J.A. *et al*. "Estimating the efficacy of symptom-based screening for COVID-19," *Nature Digital Medicine* 2020; **3**(95). DOI: 10.1038/s41746-020-0300-0 Accessed 14 July 2020.

[2] Kanzler, C.M., Rinderknecht, M.D., Schwarz, A. *et al*. "A data-driven framework for selecting and validating digital health metrics: use-case in neurological sensorimotor impairments," *Nature Digital Medicine* 2020; **3**(80). DOI: 10.1038/s41746-020-0286-7 Accessed 14 July 2020.

[3] Roy, A., Nikolitch, K., McGinn, R. *et al*. "A machine learning approach predicts future risk to suicidal ideation from social media data," *Nature Digital Medicine* 2020; **3**(78). DOI: 10.1038/s41746-020-0287-6 Accessed 14 July 2020.

[4] Levine, D.M., Co, Z., Newmark, L.P. *et al*. "Design and testing of a mobile health application rating tool," *Nature Digital Medicine* 2020; **3**(74). DOI: 10.1038/s41746-020-0268-9 Accessed 14 July 2020.

[5] Kannampallil, T., Smyth, J.M., Jones, S. *et al*. "Cognitive plausibility in voice-based AI health counselors," *Nature Digital Medicine* 2020; **3**(72). DOI: 10.1038/s41746-020-0278-7 Accessed 14 July 2020.

[6] Huang, S., Kothari, T., Banerjee, I. *et al*. "PENeta scalable deep-learning model for automated diagnosis of pulmonary embolism using volumetric CT imaging," *Nature Digital Medicine* 2020; **3**(61). DOI: 10.1038/s41746-020-0266-y Accessed 14 July 2020.

[7] Dhruva, S.S., Ross, J.S., Akar, J.G. *et al*. "Aggregating multiple real-world data sources using a patient-centered health-data-sharing platform," *Nature Digital Medicine* 2020; **3**(60). DOI: 10.1038/s41746-020-0265-z Accessed 14 July 2020.

[8] Hofer, I.S., Lee, C., Gabel, E. *et al*. "Development and validation of a deep neural network model to predict postoperative mortality, acute kidney injury, and reintubation using a single feature set," *Nature Digital Medicine* 2020; **3**(58). DOI: 10.1038/s41746-020-0248-0 Accessed 14 July 2020.

[9] Norgeot, B., Muenzen, K., Peterson, T.A. *et al*. "Protected Health Information filter (Philter): accurately and securely de-identifying free-text clinical notes," *Nature Digital Medicine* 2020; **3**(57). DOI: 10.1038/s41746-020-0258-y Accessed 14 July 2020.

[10] Choi, D., Park, J.J., Ali, T. *et al*. "Artificial intelligence for the diagnosis of heart failure," *Nature Digital Medicine* 2020; **3**(54). DOI: 10.1038/s41746-020-0261-3 Accessed 14 July 2020.

[11] Hilton, C.B., Milinovich, A., Felix, C. *et al*. "Personalized predictions of patient outcomes during and after hospitalization using artificial intelligence," *Nature Digital Medicine* 2020; **3**(51). DOI: 10.1038/s41746-020-0249-z Accessed 14 July 2020.

[12] Li, M.D., Chang, K., Bearce, B. *et al*. "Siamese neural networks for continuous disease severity evaluation and change detection in medical imaging," *Nature Digital Medicine* 2020; **3**(48). DOI: 10.1038/s41746-020-0255-1 Accessed 14 July 2020.

[13] Hoffman J. I., Nagel R., Litzke V., Wells D. A. and Amos W., "Genetic analysis of Boletus edulis suggests that intra-specific competition may reduce local genetic diversity as a woodland ages," *Royal Society Open Science* 2020; **7**:200419. Accessed 22 July 2020.

[14] Grönquist P., Panchadcharam P., Wood D., Menges A., Rüggeberg M. and Wittel F. K., "Computational analysis of hygromorphic self-shaping wood gridshell structures," *Royal Society Open Science* 2020; **7**:192210. Accessed 22 July 2020.

[15] Amos W. "Signals interpreted as archaic introgression appear to be driven primarily by faster evolution in Africa," *Royal Society Open Science* 2020; **7**:191900. Accessed 22 July 2020.

[16] Gordon M., Viganola D., Bishop M., Chen Y., Dreber A., Goldfedder B., Holzmeister F., Johannesson M., Liu Y., Twardy C., Wang J. and Pfeiffer T. "Are replication rates the same across academic fields? Community forecasts from the DARPA SCORE programme," *Royal Society Open Science* 2020; **7**:200566. Accessed 22 July 2020.

[17] Evans D, Field A. P. "Predictors of mathematical attainment trajectories across the primary-to-secondary education transition: parental factors and the home environment," *Royal Society Open Science* 2020; **7**:200422. Accessed 22 July 2020.

[18] Beale N, Battey H, Davison AC, MacKay RS. "An unethical optimization principle," *Royal Society Open Science* 2020; **7**:200462. Accessed 22 July 2020.

[19] Cherevko AA, Gologush TS, Petrenko IA, Ostapenko VV, Panarin VA. "Modelling of the arteriovenous malformation embolization optimal scenario," *Royal Society Open Science* 2020; **7**:191992. Accessed 22 July 2020.

[20] Soczawa-Stronczyk AA, Bocian M. "Gait coordination in overground walking with a virtual reality avatar," *Royal Society Open Science* 2020; **7**:200622. Accessed 22 July 2020.

[21] Duruz S, Vajana E, Burren A, Flury C, Joost S. "Big dairy data to unravel effects of environmental, physiological and morphological factors on milk production of mountain-pastured Braunvieh cows," *Royal Society Open Science* 2020; **7**:200638. Accessed 22 July 2020.

[22] Azevedo EZD, Dantas DV, Daura-Jorge FG. "Risk tolerance and control perception in a game-theoretic bioeconomic model for small-scale fisheries," *Royal Society Open Science* 2020; **7**:200621. Accessed 22 July 2020.

[23] Abdolhosseini-Qomi AM, Jafari SH, Taghizadeh A, Yazdani N, Asadpour M and Rahgozar M. "Link prediction in real-world multiplex networks via layer reconstruction method," *Royal Society Open Science* 2020; **7**:191928. Accessed 22 July 2020.

[24] Webster J and Amos M. “A Turing test for crowds,” *Royal Society Open Science* 2020; **7**:200307. Accessed 22 July 2020.

[25] Zhu Y-l, Wang C-J, Gao F, Xiao Z-x, Zhao P-l, Wang J-y. “Calculation on surface energy and electronic properties of CoS_2,” *Royal Society Open Science* 2020; **7**:191653. Accessed 22 July 2020.

[26] Yu B, Scott CJ, Xue X, Yue X, Dou X. “Derivation of global ionospheric Sporadic E critical frequency (f_oEs) data from the amplitude variations in GPS/GNSS radio occultations,” *Royal Society Open Science* 2020; **7**:200320. Accessed 22 July 2020.

[27] Joon-myoung K, Younghoon C, Ki-Hyun J, Soohyun C, Kyung-Hee K, Seung D B, Soomin J, Jinsik P and Byung-Hee O, “A deep learning algorithm to detect anaemia with ECGs: a retrospective, multicentre study,” *Lancet Digital Health* 2020; **2(7)**:e35867. Accessed 24 July 2020.

[28] Hongling Z, Cheng C, Hang Y, Xingyi L, Ping Z, Jia D, Fan L, Jingyi W, Beitong Z, Yonge L, Shouxing H, Yulong X, Binran W, Guohua W, Xiaoyun Y and Ye Y, “Automatic multilabel electrocardiogram diagnosis of heart rhythm or conduction abnormalities with deep learning: a cohort study,” *Lancet Digital Health* 2020; **2(7)**:e34857. Accessed 24 July 2020.

[29] Fung R, Villar J, Dashti A, Ismail L C, Staines-Urias E, Ohuma E O, Salomon L J, Victora C G, Barros F C, Lambert A, Carvalho M, Jaffer Y A, Noble J A, Gravett M G, Purwar M, Pang R, Bertino E, Munim S, Min A M, McGready R, Norris S A, Bhutta Z A, Kennedy S H, Papageorghiou A T and Ourmazd A, “Achieving accurate estimates of fetal gestational age and personalised predictions of fetal growth based on data from an international prospective cohort study: a population-based machine learning study,” *Lancet Digital Health* 2020; **2(7)**:e36875. Accessed 24 July 2020.

[30] Sabanayagam C, Xu, D, Ting D S W, Nusinovici S, Banu R, Hamzah H, et al, “A deep learning algorithm to detect chronic kidney disease from retinal photographs in community-based populations,” *Lancet Digital Health* 2020; **2(7)**:e295302. Accessed 24 July 2020.

[31] Monteiro M, Newcombe V F, Mathieu F, Adatia K, Kamnitsas K, Ferrante E, Das T, Whitehouse D, Rueckert D, Menon D K and Glocker B, “Multiclass semantic segmentation and quantification of traumatic brain injury lesions on head CT using deep learning: an algorithm development and multicentre validation study,” *Lancet Digital Health* 2020; **2(7)**:e31422. Accessed 24 July 2020.

[32] Liu K-L, Wu T, Chen P-T, Tsai Y M, Roth H, Wu M-S, Liao W-C and Wang W, “Deep learning to distinguish pancreatic cancer tissue from non-cancerous pancreatic tissue: a

retrospective study with cross-racial external validation," *Lancet Digital Health* 2020; **2(7)**:e30313. Accessed 24 July 2020.