Applying Bayesian Analysis Guidelines to Empirical Software Engineering Data

The Case of Programming Languages and Code Quality

Carlo A. Furia[1] · Richard Torkar[2,3] · Robert Feldt[2]

[1] Software Institute, USI Università della Svizzera Italiana, Switzerland

[2] Chalmers and the University of Gothenburg, Sweden

[3] Stellenbosch Institute for Advanced Study (STIAS), South Africa

2021–02–01

Abstract

Statistical analysis is the tool of choice to turn data into information, and then information into empirical knowledge. The process that goes from data to knowledge is, however, long, uncertain, and riddled with pitfalls. To be valid, it should be supported by detailed, rigorous guidelines, which help ferret out issues with the data or model, and lead to qualified results that strike a reasonable balance between generality and practical relevance. Such guidelines are being developed by statisticians to support the latest techniques for *Bayesian* data analysis. In this article, we frame these guidelines in a way that is apt to empirical research in software engineering.

To demonstrate the guidelines in practice, we apply them to reanalyze a GitHub dataset about code quality in different programming languages. The dataset's original analysis (Ray et al., 2014) and a critical reanalysis (Berger et al., 2019) have attracted considerable attention—in no small part because they target a topic (the impact of different programming languages) on which strong opinions abound. The goals of our reanalysis are largely orthogonal to this previous work, as we are concerned with demonstrating, on data in an interesting domain, how to build a principled Bayesian data analysis and to showcase some of its benefits. In the process, we will also shed light on some critical aspects of the analyzed data and of the relationship between programming languages and code quality.

The high-level conclusions of our exercise will be that Bayesian statistical techniques can be applied to analyze software engineering data in a way that is principled, flexible, and leads to convincing results that inform the state of the art while highlighting the boundaries of its validity. The guidelines can support building solid statistical analyses and connecting their results, and hence help buttress continued progress in empirical software engineering research.

1 Introduction

Empirical disciplines, including a substantial part of software engineering research, mine data for information, and then use the information as evidence to build, extend, and refine empirical knowledge. Statistical analysis is key to implementing this process; but statistical techniques are just tools, which need detailed *guidelines* to be applied properly and consistently. It is only through the combination of powerful statistical techniques and rigorous guidelines to apply them that we can distill empirical knowledge following a process that is consistent, rests on solid principles, and ultimately is more likely to lead to valid results with a higher degree of confidence.

Whereas frequentist statistical techniques have been commonplace in science for over a century—since the influential work of the likes of Pearson and Fisher—the state of the art in applied statistics is moving towards using *Bayesian* analysis techniques. As we discussed in previous work [13], recent developments in Bayesian analysis techniques (such as using Hamiltonian Monte Carlo fitting algorithms [8]) coupled with an increasing availability of the computing power needed to run them on large datasets have convincingly demonstrated the advantages of using Bayesian statistics and the flexibility and rigor of the analysis they support. More recently, applied statisticians have also been working out practical *guidelines* that can boost usability and impact of Bayesian statistical data analysis [1; 36; 15; 21]. In this paper, we present some of these guidelines and frame them in a way that is suitable for empirical research in the software engineering domain.

To demonstrate the guidelines in practice, we follow them to analyze a large dataset about the code quality of projects written in disparate programming languages and hosted on GitHub [34]. The empirical study that curated this dataset and performed the original analysis [34] was followed by a critical reanalysis by a different group of researchers [5]; as we recall in Section 1.1, the topic has received much attention and stirred some controversy. This visibility makes the dataset an attractive target for our own purposes.

In the paper, we go through various aspects of the data analysis performed in the previous studies [34; 5], illustrating the versatile features of Bayesian statistical models in practice. We demonstrate how the guidelines support an incremental and iterative analysis process, where several key features of a statistical model can be validated; this, in turn, encourages trying out different models and comparing them in a rigorous way—as opposed to blindly relying on one-size-fits-all rules of thumb. Following this process, we argue that some issues of the original analysis [34] or criticized by the follow-up reanalysis [5] could have been identified more easily. Furthermore, the limitations and actual impact of previous studies could have been framed more straightforwardly and more transparently. The conclusion of our exercise will be that flexible statistical techniques coupled with principled and structured guidelines can help address empirical research questions directly and transparently. This can lead to explanations that are nuanced and detailed, and hence, ultimately, that can become convincing foundations for building shared knowledge.

1.1 Dataset and previous studies

In this paper, we reuse the dataset collected and analyzed by Ray et al. in a paper published at the FSE conference [34]. A critical reproduction [5] of the original study, written by Berger et al. and published in the TOPLAS journal, triggered a prolonged controversy that reverberated on social media.

Here is the story so far, in the shortest terms possible:[1] based on their analysis of projects hosted by GitHub, the original study [34] (henceforth, "FSE") claimed to have found an association between certain programming languages and the bug proneness of code written in them. The reproduction study [5] (henceforth, "TOPLAS") criticized several aspects of FSE—most prominently, its data collection and classification practices—and questioned the soundness of some of its results. In a rebuttal, the authors of FSE defended their results;[2] and in a rebuttal of the rebuttal the authors of TOPLAS maintained their criticism.[3]

Our paper is *emphatically **not*** our attempt to jump into the fray: we do not have much to add to the subject matter of the controversy. However, we appreciate the interest that the controversial topic received, and see it as an opportunity to present our views on a different, but related, aspect: practices in statistical analysis. Both FSE and TOPLAS primarily use *frequentist* statistical techniques. Even in the best conditions, these techniques' flexibility is limited in comparison to the *Bayesian* statistical techniques we have been advocating [13; 42].

Overall, our contributions fall largely outside the focus of FSE and TOPLAS—except to the extent that they target the same domain and the same data. The core of both papers revolves around GitHub data, how it was collected and processed, and how the variables of interest have been operationalized. TOPLAS's main goal was to attempt to reproduce FSE's results, and hence it deliberately makes mostly

[1] Hillel Wayne provides a much more detailed account `https://www.hillelwayne.com/post/this-is-how-science-happens/`.
[2] `https://arxiv.org/abs/1911.07393`
[3] `http://janvitek.org/var/rebuttal-rebuttal.pdf`

limited changes to the statistical models. Our analysis takes the data as it was collected and made available by FSE's original study, and tries to make the most out of it following rigorous guidelines to apply flexible statistical practices—Bayesian statistics, that is.

1.2 Overview

The overall goal of this paper is demonstrating how Bayesian statistical techniques can be applied in a principled way to build suitable statistical models. These models can then be used to answer research questions in a flexible way, and to quantify limitations and uncertainties about what one can reliably infer from the models.

This complements our earlier work on Bayesian data analysis for empirical software engineering [13; 42], which:

- Argued for using Bayesian over frequentist statistics and showcased the former's flexibility on software engineering data [13].
- Suggested to analyze *practical* significance using a combination of Bayesian statistics and cumulative prospect theory, which helps stakeholders evaluate the impact of a technique or practice in a way that takes into account their constraints, available resources, and intuitive reasoning [42].

The current paper focuses on how to apply Bayesian data analysis in a principled way: following detailed guidelines and a structured workflow. Demonstrating the guidelines on FSE's dataset, our contributions address four aspects that are relevant to every study that involves statistical data analysis.

1.2.1 How to design a statistical model?

Modeling requires to exercise *judgement*—something that can be based on practices, customs, and heuristics, but is not completely reducible to a fixed set of rigid rules. Bayesian statistics emphasizes the *modeling* aspect of data analysis, and provides quantitative techniques to help ground heuristics and practices onto a robust and sound statistical framework.

Section 2 presents guidelines to build a Bayesian statistical model incrementally (adding features as needed), iteratively (improving a model based on the shortcomings of the previous ones), and rigorously (with quantitative criteria to assess a model's suitability). Our guidelines customize general guidelines developed by the Bayesian data analysis community to the scenarios that are common in empirical software engineering. Section 3 demonstrates, on the FSE dataset, that our guidelines provide principled ways of assessing the strengths and weaknesses of any statistical model for the analysis at hand.

1.2.2 How to spot data problems?

TOPLAS's criticism of FSE's analysis questions the accuracy of some of the *data* that was collected and how it was processed. For example, it says that "project size, computed in the FSE paper as the sum of inserted lines, is not accurate—as it does not take deletions into account" [5, §3.2]. Can Bayesian statistical techniques help discover problems with the data—such as inconsistencies, sparseness, and lack of homogeneity—that limit the validity and generalizability of the statistical analysis's results?

Naturally, no statistical technique (no matter how powerful) can supersede a careful analysis of construct validity [12; 32], which should precede analysis and lay the foundations for it. Still, applying the Bayesian guidelines that we present can ferret out issues with the data and highlight where uncertainty is more or less pronounced, so that we can heed any limitations when drawing conclusions. For example, Section 4.2 finds that the number of inserted lines performs poorly as a predictor, echoing TOPLAS's observation that it may not be a suitable measure of size.

1.2.3 How to assess significant results?

In previous work [13], we argued that Bayesian statistical techniques can help move away from a dichotomous (significant/not significant) framing of research questions—which comes typically with frequentist null hypothesis testing and is often artificially restrictive—and instead focus on *practical* significance [42].

The gap between statistical significance and practical significance is more likely to be wide when studying complex domains with plenty of confounding factors. The analysis of programming language data is a clear example of such complex domains. In this paper, we show how the Bayesian statistics guidelines support a nuanced analysis of complex models, and help keep the focus on concrete scenarios and practically relevant measures. Concretely, we show that Bayesian data analysis provides a flexible model of data distributions, which can be used to *predict* outcomes in different scenarios directly in terms of statistics that are based on variables in the problem domain.

Our analysis's conclusion will be that the key question "which programming languages are more fault prone" does not admit a simple straightforward answer—not with the analyzed data at least. Nevertheless, as we argued in [13] and now demonstrate in Section 4.3, practitioners can ask specific questions and answer them by running simulations on the Bayesian model, rather than having to rely on general results that may not be meaningful in their context.

1.2.4 How to build knowledge incrementally?

Every empirical study has limitations; lifting them requires to perform new experiments. Another advantage of Bayesian statistical models built using an incremental process is that they can be refined as we collect more data. This way, our models become better over time since they accurately reflect the evolving scientific knowledge in a certain area.

Section 4.4 discusses how applying Bayesian analysis guidelines helps plan for additional data collection based on the limitations of the analyzed data. Different experiments are no longer merely a loose collection around the same themes, but can be planned back-to-back in a way that progressively reduces the uncertainty in knowledge.

1.3 Contributions and organization

This paper makes the following contributions:

- It presents guidelines to apply Bayesian statistics following a systematic process that goes from building and validating the model to fitting and analyzing it.
- It demonstrates the guidelines by showing how to incrementally build a suitable statistical model to capture FSE's language quality data.
- It analyzes the fitted model to investigate the original questions of the effect of programming languages on fault proneness with a focus on practical scenarios.
- For reproducibility, all analysis scripts are available online together with additional results and detailed data visualization:

  REPLICATION PACKAGE: `https://doi.org/10.5281/zenodo.4472963` [14].

The rest of the paper is organized as follows. Section 2 illustrates Bayesian data analysis guidelines with an angle that is relevant for empirical software engineering. Section 3 follows the guidelines to incrementally build a model that is suitable to capture FSE's programming language data. Various models are rigorously evaluated and compared, so that the final model is arguably the "best" among them according to quantitative criteria. Section 4 analyzes the fitted model to study the original questions of which programming languages are associated with more or fewer faults. The results look at different scenarios and outline how further custom analyses could be built atop the same model. Finally, Section 5 discusses related work and Section 6 concludes with a brief summary and closing discussion.

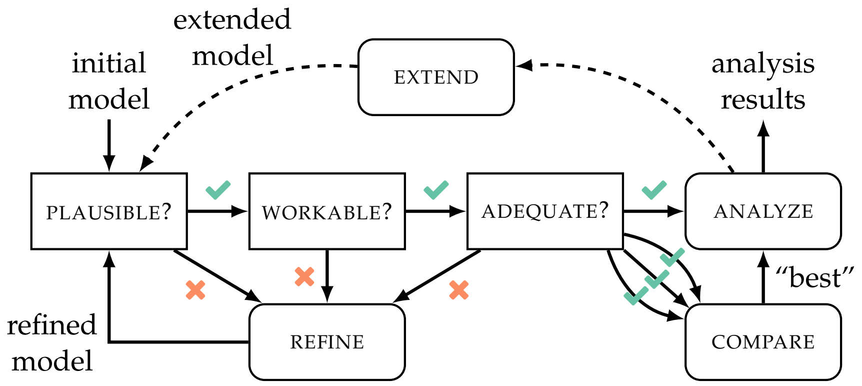


Figure 1: Process for Bayesian data analysis: starting from an initial model, assess whether it is *plausible*, *workable*, and *adequate*. If it lacks any of these characteristics, *refine* the model by adding detail and features. Models that pass all checks can be fitted and used to answer the analysis's specific questions. Different models that pass all checks can be rigorously *compared* to select those that perform "best" according to suitable criteria. The outer loop (dashed arrows) indicates that an analysis's results may also suggest to *extend* an adequate model so that it can answer more precise, or just different, questions; this outer loop is another source of multiple models that can be compared.

2 Bayesian data analysis guidelines

In the last decade, powerful Bayesian statistical analysis tools and languages—such as brms, Turing.jl, and Stan—have become widely available together with computational resources adequate to run them [9; 16; 31]. More recently, statisticians have also been introducing and refining guidelines on how to use these tools in a systematic way to perform principled Bayesian data modeling [1; 36; 15; 21]. In this section, we summarize these state-of-the-art guidelines while recasting them in a form suitable for empirical software engineering research.

A Bayesian model defines a statistical data-generating process in terms of a *prior* distribution of parameters θ and a *likelihood* that certain data is observed for each value of the parameters. *Fitting* such a model on some empirical data D then gives a *posterior* distribution of the same parameters that follows Bayes' theorem:

$$\underbrace{P(\theta \mid D)}_{\text{posterior}} \quad \propto \quad \underbrace{P(D \mid \theta)}_{\text{likelihood}} \times \underbrace{P(\theta)}_{\text{prior}} . \tag{1}$$

The posterior can then be used to compute the probability of other observations of interest in a predictive fashion. Our previous work [13] presented more details about Bayes' theorem and the roles of prior, likelihood, and posterior. In this paper, we focus on how to build a Bayesian model in practice: Bayesian modeling involves choosing components in a way that is sound and principled, and that works for the data and domain that we are targeting.

Figure 1 illustrates a key idea of the *guidelines* for Bayesian analysis presented here: developing a statistical model is a process of *iterative refinement*, which starts from a very simple (possibly simplistic) *initial model* that is gradually refined. Each iteration goes through a series of *steps* that assess the model's suitability in terms of the following characteristics:

Plausibility: Is the model consistent with (expert) knowledge about the data domain?

Workability: Can the model effectively and accurately be fitted using the available numerical algorithms?

Adequacy: Can the model capture the characteristics of the empirical data?

STEP	PRIOR	LIKELIHOOD	ALTERNATIVE MODELS	EMPIRICAL DATA	NEW DATA FOR PREDICTION
plausible?	✓				
workable?	✓	✓			
adequate?	✓	✓		✓	
compare			✓		
analyze	✓	✓		✓	✓

Table 1: For each step of a Bayesian statistical analysis (checks of plausibility, workability, adequacy, model comparison, and analysis), the model components (PRIOR and LIKELIHOOD), competing ALTERNATIVE models, and kinds of DATA (EMPIRICAL and NEW FOR PREDICTION) that the step primarily *tests*.

These steps help assess the *utility* (or suitability) of a model and its trade-offs. Each step puts additional requirements on a model, and checks whether the model is well-equipped to faithfully capture the observed data and to analyze it. As shown in Table 1, each step broadens the scope of what model components are checked; in particular, the actual empirical data is only used in the *adequate* step, whereas the previous steps generate simulated data using priors and likelihood. As summarized in Table 2, each step has a possible *outcome* (what it establishes about the model), which is supported by analysis *artifacts* that document the step. The following sections describe the steps, artifacts, and outcomes in some detail. Section 3 will apply the steps on the main case study of programming language data.

To make the description concrete, and thus easier to follow, we illustrate what the steps compute on a toy model that estimates an adult person's height h as follows:

$$h \sim \mathsf{Normal}(\mu, \sigma) \tag{2}$$
$$\mu \sim \mathsf{Normal}(170, 50) \tag{3}$$
$$\sigma \sim \mathsf{HalfCauchy}(0, 1) \tag{4}$$

The model's parameters θ are μ and σ, whereas the data D records the value of outcome variable h for several persons. There are no predictor variables in this simplistic model, but otherwise these would also be recorded in the data for every person. The likelihood is defined by (2), which is in fact a probability distribution of the data h given parameters μ and σ. The two remaining equations (3),(4) define the prior probability distributions of the same parameters without any dependency on the data.

2.1 Plausible model

Once we have chosen parameters, likelihood, and priors our model definition includes all required parts. Then, we can "run" the model—that is, sample from it—and analyze how likelihood, data, and priors constrain the model parameters of interest [20]. The first step of this analysis focuses on the priors, which should be neither too constraining nor unreasonably permissive. This step is typically called *prior predictive simulations* or *prior predictive checks* [36; 26] and works as follows: sample the priors broadly; using the sampled distribution (ignoring the actual empirical data, which are not used in this step), run the model to get a distribution of the model variables of interests (typically, the outcome variable); check that this distribution is plausible for the variables that it measures.

The key principle is that, if the priors are properly chosen, this process should determine a distribution that allows all plausible values for the variables but gives vanishing small probability to values that are practically impossible or contradict established scientific knowledge. In summary, prior predictive checks answer the question: *Does sampling from the priors lead to a plausible range of parameter values?*

Here is how prior predictive simulation would work on our toy example. First, we sample random values for parameters μ and σ from their prior distributions (3),(4). We then plug each sampled pair of

STEP	ARTIFACTS	OUTCOME
plausible?	1. Prior predictive simulation plots; 2. Justification for priors if they disallow certain values.	The priors allow a broad range of possible values and give low probability to values that are unlikely to occur in the domain.
workable?	1. Simulation-based calibration of z score and shrinkage; 2. Fitting diagnostic metrics.	Fitting the model works computationally and does not exhibit pathological behavior.
adequate?	Posterior predictive checks plots.	The model can generate data similar to the empirical observations.
compare	Information-criteria ranking and scores of competing alternative models.	The chosen model achieves a bias-variance trade-off better than the alternative models.
analyze	1. Posterior plots based on the empirical data; 2. Distribution plots and summary statistics of any domain-specific variables of interest.	Quantitative answers to the analysis's specific questions.

Table 2: The ARTIFACTS that are typically produced, and the OUTCOME that follows from each step of a Bayesian statistical analysis (when the step succeeds).

values $\overline{\mu}, \overline{\sigma}$ into the likelihood (2) and sample values of h from $\mathsf{Normal}(\overline{\mu}, \overline{\sigma})$. Since the outcome variable h measures an adult person's height, the priors should be such that this sampled distribution of h freely allows heights between, say, 0 and 300 cm, whereas it disallows negative heights and assigns very small probabilities to heights above 300 cm—since no human on record has ever been that tall.

Prior predictive simulation is not cheating [26]. As long as we do not set priors based on the actual empirical data that we are going to analyze, but only through what we know about the data *domain* independent of how we measured it, it is sensible to use our existing knowledge to rule out priors that would lead to impossible or clearly implausible results. Seen in this light, the possibility of choosing priors is a big advantage of Bayesian analysis, and clearly so in an empirical science. Using existing knowledge to guide new analyses, we can develop sequences of studies that, taken together, progressively sharpen knowledge in a specific area. The alternative is that every software engineering research contribution remains an "island unto itself" without clear connections to the related literature and the field as a whole [13].

Nevertheless, the reasons for choosing certain priors that make the model *plausible* should be explicitly justified. In practice, and to the extent that it is possible, empirical software engineering studies should explicitly state which published results, common sense, or "folk knowledge" justify the choice of priors and their plausibility behavior. Section 3.3 demonstrates how to do that for the paper's case study.

2.2 Workable model

Once we have ascertained that the chosen priors are consistent with plausible parameter values, the second step checks whether our model *works* computationally—that is, *fitting* the model does not incur divergence or other numerical problems, and the fitting process eventually reaches a stationary state that properly identifies a posterior distribution. An emerging technique to do so is *simulation-based calibration* [40; 36], which relies on a consistency property of Bayesian models: first, simulate parameter and data values from the priors and likelihood as done in prior predictive simulations; then, using Bayes' theorem, combine the simulated parameter and data samples to get a *posterior* distribution of the model's parameter; if the model is consistent, the posterior distribution obtained in this way should resemble the prior distribution. Simulation-based calibration typically uses the *rank* of the sampled prior values within the sequence of sampled posterior values as a measure of similarity. These ranks should be uniformly distributed in a workable model; it they are not, it means that the model is misspecified or the fitting algorithm produces biased samples [36]. In either case, the model misbehaves computationally, and we have to either revise it or try other fitting algorithms to analyze it.

Here is how simulation-based calibration would work on our toy example. As in prior predictive checks, we first sample parameter values from the priors and use those to build samples of the outcome variable h. We then use these samples as data (instead of the actual empirical data) and combine them again with priors and likelihood using Bayes' theorem (1). This gives a new sampled distribution of the parameters μ and σ, which we compare with that obtained by sampling the priors directly in the first step.

Simulated observations produced by simulation-based calibration can also be used to study the *sensitivity* of our model, if we *quantify* the difference between values that are sampled from the priors and those sampled from the posterior, and use this difference as a measure of sensitivity. For example, we may compare, for every model parameter, mean and standard deviation of the prior samples to mean and standard deviation of the posterior samples. If we combine mean and standard deviation into z-score and shrinkage (a ratio of variances), different ranges for these metrics identify different pathological behavior [36]: overfitting (high z-score and high shrinkage), poor identification (low z-score and low shrinkage), and conflict between prior and observations (high z-score and low shrinkage).[4]

While promising, simulation-based calibration is a cutting-edge technique that is still undergoing major developments; none of the statistical analysis tools that are more widely used for Bayesian analysis support it out-of-the-box. Instead, these tools offer other metrics to assess workability that are specific to the fitting algorithms based on dynamic Hamiltonian Monte Carlo which they implement. These are the metrics that are usually available and how we can use them to assess workability of a model:

- A *divergent transition* in the sequences of samples indicates a possible numerical error; workable models should have few divergent transitions—ideally none.
- The sampling process is repeated a few times independently; each sequence of sampling is called a *chain*. In a workable model, different chains should be statistically similar: the ratio $\widehat{R}$ of within-to-between chain variance should converge to 1 as the number of samples grows. A common rule of thumb for finite sampling is that $\widehat{R} < 1.01$, which indicates a stationary posterior distribution.
- The *effective sample size* is the fraction of all samples that are independent, that is not autocorrelated. We typically want it to be at least 10% for each parameter we estimate (and that the absolute number of independent samples is a few hundreds); lower values may indicate that sampling is ineffective.
- Finally, we can also visually inspect the plots that trace the samples in every chain. When the different lines look mixed up (like a "hairy caterpillar" [26]), it is one more sign that the fitting process works well.

Section 3.4 uses these metrics to analyze the workability of our models.

2.3 Adequate model

If the previous analysis steps were successful, we determined that the priors are sensible (*plausibility*) and that fitting the model is a converging process (*workability*); it remains to check whether the model adequately captures reality. The third step thus fits the model using the actual empirical data (which was not used in the previous two steps) and performs *posterior predictive checks*: using the posterior distribution of parameters fitted on the actual empirical data, simulate new observations and compare them to the data. If the two are consistent, it means that the model can generate data similar to the observed data, and hence it captures the empirical observations *adequately*.

Here is how posterior predictive checks would work on our toy example. Similarly as in simulation-based calibration, we combine data and prior samples using Bayes' theorem (1); the key difference is that we now use the actual observed data (the height of real people) instead of simulated data. This gives

[4]See Betancourt's discussion for detailed examples: `https://betanalpha.github.io/assets/case_studies/principled_bayesian_workflow.html`.

a posterior predictive distribution of parameters μ and σ, which, in turn, we sample; then, we plug the sampled parameter values into the likelihood to get a distribution of h—the so-called *posterior predictive distribution*, since it expresses the information about the posterior indirectly in terms of prediction of model (outcome) variables. In an adequate model, the posterior predictive distribution generates data somewhat similar to the actual observed data.

Section 3.5 discusses the results of posterior predictive checks on the programming language case study.

2.4 Model comparison

Information criteria such as WAIC [45] and LOO [44] assess a kind of relative adequacy by measuring deviance or other information-theoretic metrics between a model's predictions and the data. In a nutshell, these metrics assess how well each model performs out-of-sample predictions compared to other competing models. Thus, information criteria measures are *relative*: they are useful to compare the adequacy of a model relative to another but cannot gauge a model's adequacy in absolute terms. Section 3.6 uses information criteria to compare different models for the programming language data analysis.

2.5 Iterative refinement

After a candidate model goes through the steps described above, we have a clear understanding of its strengths and weaknesses. When the model fails specific steps, we also learn what aspects we have to change to refine it: the priors of an implausible model need changing; an unworkable model needs to be refactored in a way that works computationally; an inadequate model may require more information (typically in the form of additional variables or parameters) for it to be consistent with the data (for example, to properly capture inter-group variability).

Model design is an *iterative* process which gradually refines an initial model to improve it. Usually, we start from a deliberately very simple model and make it more complex as needed [36]. However, we can also do the opposite: start from a so-called *maximal* model, and then simplify it as long as it retains the characteristics of plausibility, workability, and adequacy [30]. In practice, we may even alternate simplification and refinement (detail-adding) steps starting from a canonical model [29, pp. 103–104] until we are satisfied with the results.

The presentation of the results of a Bayesian data analysis need not discuss the models in the same order in which they were designed and evaluated; it does not even need to present all models, but can simply present the final model as long as its choice can be soundly justified a posteriori (and, preferably, a reproducibility package exists). Regardless of how we choose to present the overall outcome of an analysis, expands the flexibility of the modeling process, supports making informed choices about each aspect of a model, and helps focus on and quantify relative benefits of each model in terms of the trade-offs that matter for the ongoing analysis.

3 Bayesian data analysis of programming language data

Equipped with a high-level understanding of the modeling and analysis guidelines that we outlined in Section 2, we apply them to perform the analysis of the FSE data. The overall outcome of the work described in this section will be a carefully designed, suitable statistical model of this data. In Section 4, we will analyze this model to understand what it tells us about the original questions on programming languages and code quality.

To mitigate the risk of mono-operational bias, the first author prepared the data for analysis in R, and the second author developed the first complete analysis, which then the first and third author revised. Finally, all three authors validated the final revised analysis, which is presented here. In addition, the second author did not read the publication that originated the dataset [34] or its reanalysis [5] until after completing the first complete analysis. This reduced the chance that the others' design decisions, or some characteristics of the data they highlighted, biased our application of the modeling guidelines.

3.1 Data

FSE's authors released the original dataset—obtained by mining information from GitHub repositories—upon request from TOPLAS's authors. TOPLAS performed first a repetition of FSE's analysis on the same dataset, and then a reanalysis on a revised dataset obtained by "alternative data processing and statistical analysis to address what [they] identified as methodological weaknesses of the original work" [5, Sec. 4]. The main difference between FSE's original dataset and TOPLAS's revised dataset is that the latter removes some duplicated data, TypeScript projects (which often do not include much actual TypeScript code), and the V8 project (whose JavaScript code in the dataset is mostly tests). Finally, TOPLAS's replication package includes FSE's original dataset alongside TOPLAS's revised dataset.

In our analysis, we focus on the original FSE dataset,[5] because we would like to see whether a Bayesian data analysis can help spot issues and inconsistencies in the data that may hinder replication attempts—and, conversely, that may make replication run-of-the-mill if addressed early on. Our replication package includes all analysis details, including the results of fitting the same models on TOPLAS's revised dataset (which we do not discuss here for brevity).

FSE's dataset includes information about 1 578 165 commits, which we group by project and language giving 1 127 datapoints. The attributes that are relevant for our analysis are:

project:	the project's name
language:	the used programming language
commits:	the total number of commits in the project
insertions:	the total number of inserted lines in all commits
age:	the time passed since the oldest recorded commit in the project
devs:	the total number of users committing code to the project
bugs:	the number of commits classified as "bugs"

The values of attributes *commits*, *insertions*, *age*, and *devs* vary greatly between projects. When this happens, it is customary to transform the data using a logarithmic function, so that the variability is over a smaller range whose unit corresponds to an "order of magnitude" difference. Both FSE's and TOPLAS's analyses log-transformed these attributes; we do the same: henceforth, *commits*, *insertions*, *age*, and *devs* represent the natural *logarithm* of the total number of commits, inserted lines, and so on.

3.2 Modeling

A Bayesian statistical model consists of three components: parameters to estimate, likelihood, and priors. We introduce each component to describe a complete model for our data. Precisely, we build three models—$\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$—of increasing complexity.[6] Table 3 summarizes the outcome of the steps in Figure 1 for the three models. Mirroring Table 2's structure, Table 4 outlines the artifacts produced in each step, and the conclusions that the analysis draws about each model's suitability. The rest of this section details the models and the outcome of the guidelines' suitability checks presented in Section 2.

[5]Which we obtained from TOPLAS's public replication package.

[6]As discussed in Section 2.5, for clarity we present the three models at once but we actually designed them over several iterated applications of the guidelines.

STEP	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$
plausible?	✓	✓	✓
workable?	✓	✓	✓
adequate?	✗	✓	✓
compare	–	✗	✓

Table 3: All three models are *plausible* and *workable*, but model $\mathcal{M}_1$ is *not adequate* because it cannot accurately capture the regular features of the dataset. Model *comparison* between $\mathcal{M}_2$ and $\mathcal{M}_3$ shows that the latter performs much better concerning out-of-sample predictions, and hence we will use $\mathcal{M}_3$ for the rest of the analysis.

STEP	ARTIFACTS	OUTCOME
plausible?	1. Prior predictive simulation plots in Figure 4; 2. Justification for priors: typical relations between project size and number of known bugs [37].	The priors of all three models allow a very broad range of possible values for the number of bugs that may exist; extremely high numbers are still possible but with much probability.
workable?	Fitting diagnostic metrics reported in Section 3.4: $\widehat{R}$, effective sample size, no divergent transitions, trace plots (details in the replication package).	Fitting all three models works computationally and reaches convergence.
adequate?	Posterior predictive checks plots in Figure 5.	Model $\mathcal{M}_1$ cannot generate a distribution similar to that observed in the data (Figure 5a), whereas models $\mathcal{M}_2$ and $\mathcal{M}_3$ can.
compare	Information-criteria scores of competing models in Table 5.	Model $\mathcal{M}_3$ clearly outperforms $\mathcal{M}_2$ in how it can predict data out of the sample used for fitting.
analyze	1. Posterior plots based on the empirical data in Figures 9 and 11; 2. Distribution plots and summary statistics of any domain-specific variables of interest in Figures 6 and 10.	Quantitative answers to the analysis's specific questions in Section 4.

Table 4: A summary of the ARTIFACTS produced by the analysis of the three models, and the OUTCOME of each step of the analysis in terms model suitability. This summary instantiates Table 2 for the programming language data analysis.

$$\begin{aligned} \mathit{bugs}_i &\sim \mathsf{NegativeBinomial}(\lambda_i, \phi) \\ \log(\lambda_i) &= \Pi_i + L_i \\ \Pi_i &= \alpha \\ L_i &= \alpha_{\mathit{language}_i} \end{aligned}$$

(a) Model $\mathcal{M}_1$

$$\begin{aligned} \mathit{bugs}_i &\sim \mathsf{NegativeBinomial}(\lambda_i, \phi) \\ \log(\lambda_i) &= \Pi_i + L_i \\ \Pi_i &= \alpha + \beta^{\mathrm{C}} \cdot \mathit{commits}_i + \beta^{\mathrm{I}} \cdot \mathit{insertions}_i \\ &\quad + \beta^{\mathrm{A}} \cdot \mathit{age}_i + \beta^{\mathrm{D}} \cdot \mathit{devs}_i \\ L_i &= \alpha_{\mathit{language}_i} \end{aligned}$$

(b) Model $\mathcal{M}_2$

$$\begin{aligned} \mathit{bugs}_i &\sim \mathsf{NegativeBinomial}(\lambda_i, \phi) \\ \log(\lambda_i) &= \Pi_i + L_i + P_i \\ \Pi_i &= \alpha + \beta^{\mathrm{C}} \cdot \mathit{commits}_i + \beta^{\mathrm{I}} \cdot \mathit{insertions}_i \\ &\quad + \beta^{\mathrm{A}} \cdot \mathit{age}_i + \beta^{\mathrm{D}} \cdot \mathit{devs}_i \\ L_i &= \alpha_{\mathit{language}_i} + \beta^{\mathrm{C}}_{\mathit{language}_i} \cdot \mathit{commits}_i \\ &\quad + \beta^{\mathrm{I}}_{\mathit{language}_i} \cdot \mathit{insertions}_i + \beta^{\mathrm{A}}_{\mathit{language}_i} \cdot \mathit{age}_i \\ &\quad + \beta^{\mathrm{D}}_{\mathit{language}_i} \cdot \mathit{devs}_i \\ P_i &= \alpha_{\mathit{project}_i} \end{aligned}$$

(c) Model $\mathcal{M}_3$

Figure 2: The likelihoods of statistical models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$. Colors highlight the terms that are added to each model compared to the previous ones.

3.2.1 Likelihood (and parameters)

Generalized linear models are a broad category of statistical models that are so flexible that they can be "applied to just about any problem" [18] when modeling empirical data. The likelihood of a generalized linear model is a probability distribution over certain parameters, which are generalized linear functions of the variables chosen as predictors. The values drawn from the distribution correspond to the outcome that we are modeling.

Distribution family. In our case, the *outcome* variable is *bugs*, which always is a nonnegative integer. Therefore, we should select a likelihood distribution suitable for "counting"—that is, one in the Poisson family. The single-parameter Poisson is the distribution in this family with the highest information entropy [24], and hence it should be the customary initial choice.

Nevertheless, building a model using the single-parameter Poisson quickly reveals that it cannot account for the fact that the distribution of *bugs* in the data is *overdispersed*: its mean $\mu_{bugs} = 501$ is much smaller than its variance $\sigma^2_{bugs} = 15\,031\,006$. This justifies selecting the slightly more complex *negative binomial* distribution $\mathsf{NegativeBinomial}(\lambda, \phi)$. The two parameters λ and ϕ represent[7] rates that together determine the distribution's mean λ and variance $\lambda + \lambda^2/\phi$, which can take different values to accurately capture overdispersion. This is in contrast to the Poisson distribution whose mean and variance coincide. The negative binomial distribution is also the same distribution selected, for the same reason, by both FSE's original analysis and TOPLAS's reanalysis.

Model $\mathcal{M}_1$. The first model we consider, called $\mathcal{M}_1$, is very simple: it assumes that the rate λ is a function of two terms only. The first term Π is a constant intercept α; the symbol Π highlights that it is a population-level term. The second term L is an additional intercept $\alpha_{language}$ that depends only on the *language* used in each observation; the symbol L highlights that it is a language-level term. Figure 2a

[7] `https://mc-stan.org/docs/2_20/functions-reference/nbalt.html`

shows $\mathcal{M}_1$'s overall likelihood, where the logarithm function *links*[8] the linear function of the parameters and λ so that the latter is always a nonnegative number—as it should be in a "counting" distribution.

Model $\mathcal{M}_1$ is obviously too simple to capture the variability in the data with high accuracy. Nonetheless, it is a useful starting point to understand the key relations between variables and to bootstrap the process that leads to incrementally more refined and precise models. In its simplicity, it highlights that the key predictor (the "treatment") is the programming language used in each project, whose relation with the number of bugs we would like to capture. Finally, even a simplistic model serves as a useful *baseline* to compare to more complex models—as a sanity check that the additional complexity that we are going to add to the models brings measurable improvements over the baseline.

Model $\mathcal{M}_2$. The second model we consider, called $\mathcal{M}_2$, is a standard linear-regressive model with negative binomial likelihood. Model $\mathcal{M}_2$'s population-level term Π is a linear function with intercept α and a slope β for each predictor variable commits, insertions, age, and devs. In addition, like model $\mathcal{M}_1$, $\mathcal{M}_2$ includes a language-level term L that consists of an intercept that depends on the language used in each observation (a so-called "varying intercept" model [19]). Figure 2b shows $\mathcal{M}_2$'s overall likelihood.

Model $\mathcal{M}_2$ is the closest to the regressive models used in FSE and TOPLAS. The only difference is how each model accounts for the dependence on the programming *language*: FSE and TOPLAS use different kinds of *contrasts* [34; 5], whereas we simply add an intercept language-level term—thus making our models *multilevel* [18]. Multilevel modeling comes natural with Bayesian statistics, both because we do not have to worry too much about adding layers to the model (unlike with frequentist techniques, changing such characteristics of the model does not require changing the fitting algorithm) and because we can just model the quantities of interest directly and *compute* any derived quantity *after* we fit the model's *posterior* distribution (unlike with frequentist techniques, which mostly provide only point estimates without distributional information).

Model $\mathcal{M}_3$. The third model we consider, called $\mathcal{M}_3$, is a multilevel model that tries to capture the effect of the programming language with greater detail. Model $\mathcal{M}_3$'s population-level term Π is identical to $\mathcal{M}_2$'s. Its language-level term L is considerably more complex, since it introduces a linear model with different intercepts $\alpha_{language}$ and slopes $\beta_{language}$ for each programming language (a so-called "varying intercepts and varying slopes" model, also commonly known as "varying effects" model [19]). Unlike the population-level term Π, the language-level term L *pools* the information about each data cluster—where clusters are identified by the used programming language. Since it clusters by programming language, this partial pooling may help capture more accurately the effects of choosing a programming language instead of another; at the same time, it also shares information among clusters so that some information from larger clusters (languages with many projects) can sharpen the information from smaller clusters (languages with fewer projects). This also means that partial pooling helps protect from *overfitting*, as learning takes place first separately on each cluster, and then is "regularized" by sharing its results among different clusters.

The three models' focus on the programming language reflects our intuitive expectation that the relation between programming languages and proneness to bugs is an important one—regardless of whether it turns out to be significant or negligible in the end. At the same time, adding predictors other than the programming language *controls for* confounding factors that may have a stronger correlation with the number of bugs. But what if the intrinsic differences between *projects* turn out to dominate the discrepancies in code quality? For instance, different projects may have wildly different protocols to report, triage, and fix bugs, which might have an effect on the observed number of bugs.

In order to hedge against this possible confounding factor, model $\mathcal{M}_3$ also includes a term P: an additional intercept $\alpha_{project}$ that depends only on each observation's *project*; the symbol P highlights that it is a project-level term, which will help in quantifying the intrinsic variability across projects. Figure 2c shows $\mathcal{M}_3$'s overall likelihood.

[8]In other words, the link function converts measures from the probability space to the outcome space.

$$
\begin{aligned}
\alpha &\sim \mathsf{Normal}(0, 5) \\
\alpha_{language} &\sim \mathsf{Normal}(0, \sigma_\alpha) \\
\sigma_\alpha &\sim \mathsf{Weibull}(2, 1) \\
\phi &\sim \mathsf{gamma}(0.01, 0.01)
\end{aligned}
$$

(a) Priors of model $\mathcal{M}_1$

$$
\begin{aligned}
\alpha &\sim \mathsf{Normal}(0, 5) \\
\beta &\sim \mathsf{Normal}(0, 0.5) \\
\alpha_{language} &\sim \mathsf{Normal}(0, \sigma_\alpha) \\
\sigma_\alpha &\sim \mathsf{Weibull}(2, 1) \\
\phi &\sim \mathsf{gamma}(0.01, 0.01)
\end{aligned}
$$

(b) Priors of model $\mathcal{M}_2$

$$
\begin{aligned}
\alpha &\sim \mathsf{Normal}(0, 5) \\
\beta &\sim \mathsf{Normal}(0, 0.5) \\
\alpha_{language} &\sim \mathsf{Normal}(0, \sigma_\alpha) \\
\beta_{language} &\sim \mathsf{Normal}(0, \sigma_\beta) \\
\sigma_\alpha, \sigma_\beta, \alpha_{project} &\sim \mathsf{Weibull}(2, 1) \\
\mathcal{L} &\sim \mathsf{LKJ}(2) \\
\phi &\sim \mathsf{gamma}(0.01, 0.01)
\end{aligned}
$$

(c) Priors of model $\mathcal{M}_3$

Figure 3: The priors of statistical models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$. Colors highlight the terms that are added to each model compared to the previous ones.

3.2.2 Priors

As we demonstrated in the previous section, choosing the likelihood typically requires making justified modeling choices, which depend on the kind of analysis we would like to carry out.

When choosing the priors, in contrast, we can often rely on standard recommendations that primarily depend on the domain of each variable. This does not mean that priors (or likelihoods, for that matter) can be always chosen blindly using a fixed table of recommendations. In the following sections, as we go through the various steps of the Bayesian data analysis workflow, we will validate our choices of priors and likelihood. If validation fails, we have to go back and revise the model: priors, likelihoods, or both.

Specifying priors. Specifying a prior means defining an initial probability distribution for a parameter of the model. The parameters are those whose distribution will be estimated according to the information provided by the data.

The least informative priors are completely *flat* distributions, which assign the same infinitesimal probability to any value of the parameter; this is the default behavior in frequentist statistics. Flat priors are usually a poor choice: first, since they stretch a probability distribution over an infinitely large support, they tend to generate infinitesimal probabilities that may cause *numerical* rounding errors; second, a prior with no information whatsoever about the realistic parameter domain is prone to *overfitting* the data.

A better choice are *weakly informative* priors, which still carry very little specific information but perform much better than flat priors computationally and protect against overfitting the data. Bayesian data analysis tools can often suggest *default* weakly informative priors that may work well in many cases. In our analysis, we will define our priors—following standard recommendations—but very often using default priors would have lead to overall similar results.

Model $\mathcal{M}_1$. As shown in Figure 3a, we use weakly informative priors for model $\mathcal{M}_1$ that are based on the normal distribution. As we will see during the plausibility analysis (Section 3.3), model $\mathcal{M}_1$ is so simplistic that its performance is not affected much by the choice of priors; nonetheless, we discuss its priors in some detail because we will build on them to choose priors for the more complex models.

The intercept α's prior has mean 0 (that is, we do not know a priori whether the intercept is positive or negative) and standard deviation 5. Remember that the estimated parameter λ is log-transformed (see Figure 2b); therefore, we can appreciate how weakly constraining this prior is: two standard deviations on each side of zero span the interval from $e^{-10} \simeq 0$ to $e^{10} \simeq 22\,000$ on the bug counting scale; that is, the prior only assumes that a project's bugs are up to 22 000 with 95% probability—which is not a strong assumption at all. Besides, a normal distribution has *infinite* support, and hence it does not rule out any count of bugs if the data provides evidence for it. The prior for the language-level intercept $\alpha_{language}$ is also a normal distribution with mean 0; however, choosing the same standard deviation σ_α for every language would defeat the purpose of having language-level intercepts. Instead, we let σ_α be a random variable, and assign a prior to it. Distributions with support limited to positive values are suitable priors for standard deviations—which must be nonnegative values. In this case, we use a Weibull for σ_α and the default Gamma for the dispersion parameter ϕ of the negative binomial.

Model $\mathcal{M}_2$. In addition to the default weakly informative priors for α, $\alpha_{language}$, and ϕ, $\mathcal{M}_2$ needs a prior for the slope parameter vector β. Here too we use a simple normal distribution with mean 0 (so that there is no bias in the possible direction of each predictor's effect) and standard deviation 0.5 (which still allows for a broad variability on the logarithmic scale). The β's standard deviations are smaller than the α's because α determines the population average, and then β moves this average according to each predictor's effect (which needs only introduce a smaller variation relative to the average). Figure 3b shows the overall priors for $\mathcal{M}_2$.

Model $\mathcal{M}_3$. We choose the prior for the new part of $\mathcal{M}_3$—the language-level slopes $\beta_{language}$—similarly to how we chose the language-level intercepts $\alpha_{language}$: a normal with mean 0 and a random variable σ_β for standard deviation. However, there is an additional technicality that we need to handle: vectors $\alpha_{language}$ and $\beta_{language}$ are not independent but are components of a single *multivariate* normal distribution with a variance *matrix* S. Variance matrix S combines diagonal matrices with the components of $\alpha_{language}$ and $\beta_{language}$ and a covariance matrix $\mathcal{L}$. The customary prior for covariance matrices is a multivariate Lewandowski-Kurowicka-Joe distribution; LKJ(2) is a weakly informative prior using this distribution, which assigns low probabilities to extreme correlations. Finally, the project-level intercept $\alpha_{project}$'s prior is also a Weibull—a weakly informative distribution that constrains the standard deviation for $\alpha_{project}$ to be a non-negative value. Figure 3c shows the overall priors for $\mathcal{M}_3$.

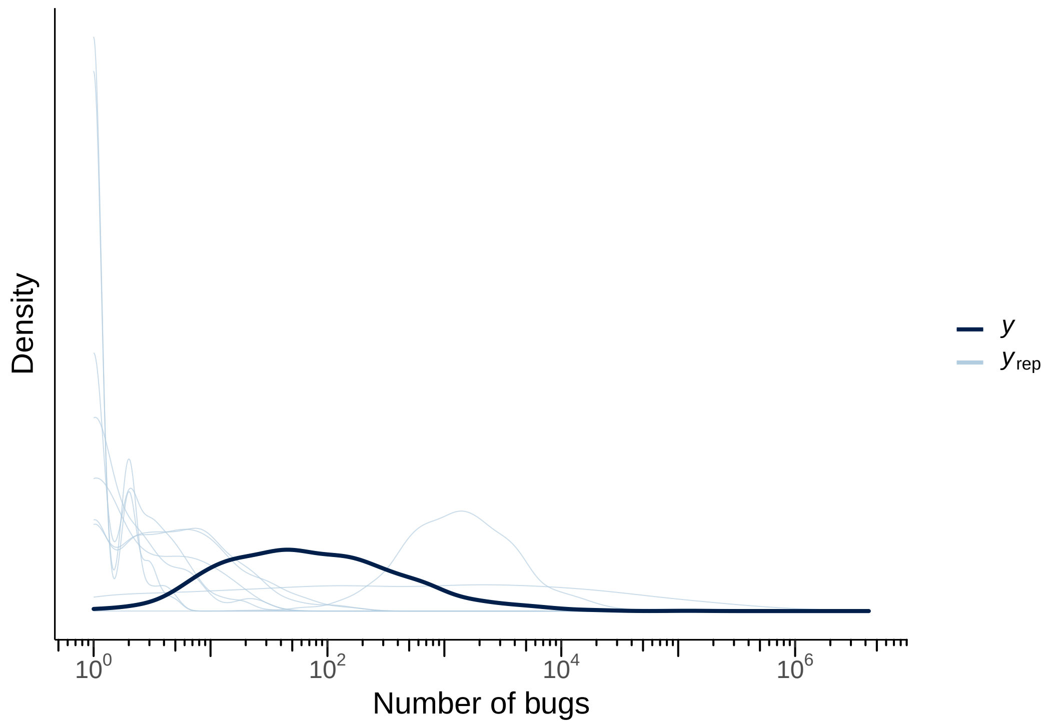
3.3 Plausibility

The prior predictive checks are straightforward for all three models, confirming that our choice of priors—based on standard recommendations for these kinds of models—leads to plausible outcomes. As an example, Figure 4a shows several distributions of the outcome variable *bugs* obtained with prior predictive simulations of $\mathcal{M}_2$. These span a very wide support that goes from zero[9] up to over a million bugs per project. While there are no theoretical limits on the number of bugs in a project independent of its size, it is realistic that most projects have less than one million *known* bugs, and the majority of projects have less than a few thousands—simply because not many projects have more than one known bug for each line of code [37], and hence a project's size in lines of code is a workable upper bound on the number of distinct bugs. Anyway, the priors still allow even larger bug counts, but assign to them increasingly smaller probabilities. Figure 4a also displays the empirical distribution of bug counts in FSE's dataset; this visually confirms that the priors are not too restrictive and reflect reasonable expectations. The prior predictive checks of model $\mathcal{M}_3$ is shown in Figure 4b, and leads to qualitatively similar conclusions—in fact even stronger, given that the priors stretch past an astronomical number of bugs—about the model's plausibility. So do the prior predictive checks of model $\mathcal{M}_1$, which we do not show for brevity.

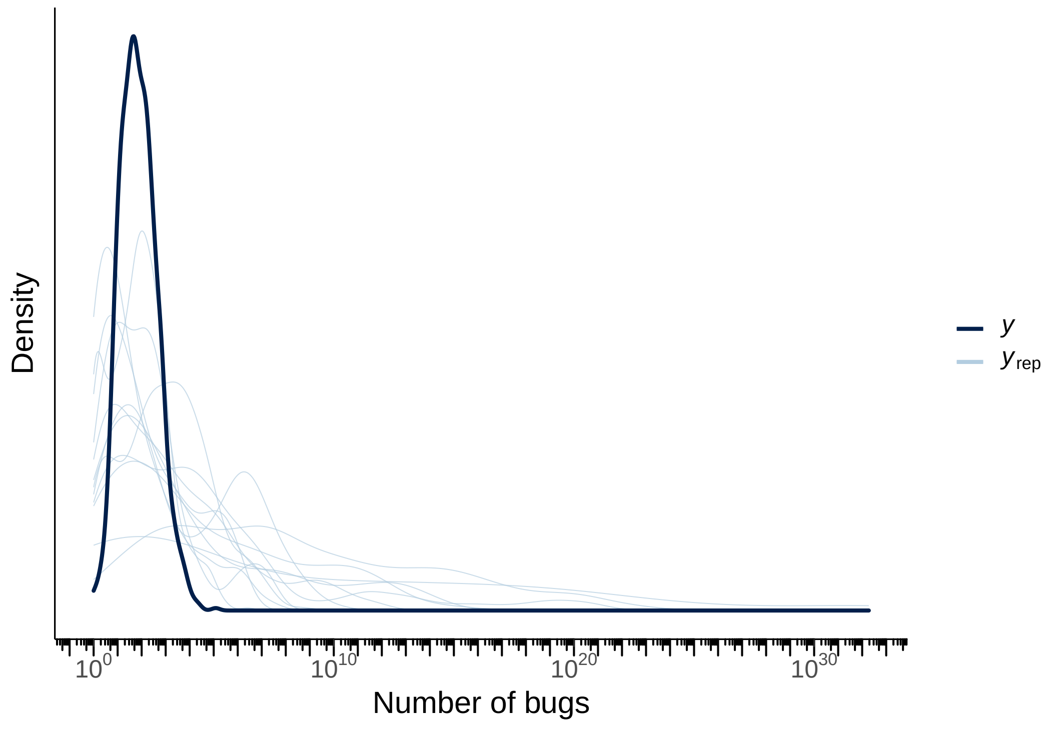
3.4 Workability

Section 2.2 outlined *simulation-based calibration* and Hamiltonian Monte Carlo validation metrics to assess a model's workability. We use the latter, which are extensively supported by Stan, to determine whether

[9]The logarithmic link function guarantees a lower bound of zero.



(a) Prior predictive simulation for $\mathcal{M}_2$.



(b) Prior predictive simulation for $\mathcal{M}_3$.

Figure 4: Prior predictive simulation plots for models $\mathcal{M}_2$ and $\mathcal{M}_3$: each thin light blue line pictures one simulated distribution of the number of bugs in a project drawn from the priors. For comparison, the thick dark blue line pictures the distribution of the number of bugs in the measured data. The horizontal scale is logarithmic in base 10.

MODEL	RANK	DIFFERENCE	STANDARD ERROR
$\mathcal{M}_3$	1	–	–
$\mathcal{M}_2$	2	-172.0	23.6
$\mathcal{M}_1$	3	-1963.0	57.0

Table 5: Each MODEL is RANKed (from better to worse) according to the PSIS-LOO information criterion. The score DIFFERENCE between each model and the immediately better one in the ranking, as well as the STANDARD ERROR of such difference, quantify the difference in adequacy between models.

the sampling process for each of the three models reached a stable state.

$\widehat{R}$—the ratio of within-to-between chain variance—is < 1.01 for all three models; this indicates that the chains have converged towards a stationary posterior probability distribution. The *effective sample size* is at least 0.11, 0.17, 0.13 for all parameters in each of the three models, and confirms that sampling effectively converged. Fitting all three models does not run into any *divergent transitions*, and the trace plots of the models (included in the replication package) look well-mixed. In summary, all three models work well computationally.

3.5 Adequacy

Let us first study the *adequacy* of our models with posterior predictive simulations: we visually compare the distribution of number of bugs per project in our dataset to several simulated distributions using the fitted models. Figure 5a indicates that $\mathcal{M}_1$ is *not* adequate: it is too simplistic to capture the data's features; in particular, the means of the simulated distributions are more than ten times larger than the mean of the data. In contrast, $\mathcal{M}_2$ passes this adequacy test: Figure 5b shows that the model's predictions look similar to the data. Model $\mathcal{M}_3$ also passes the visual adequacy test based on the posterior predictive simulations.

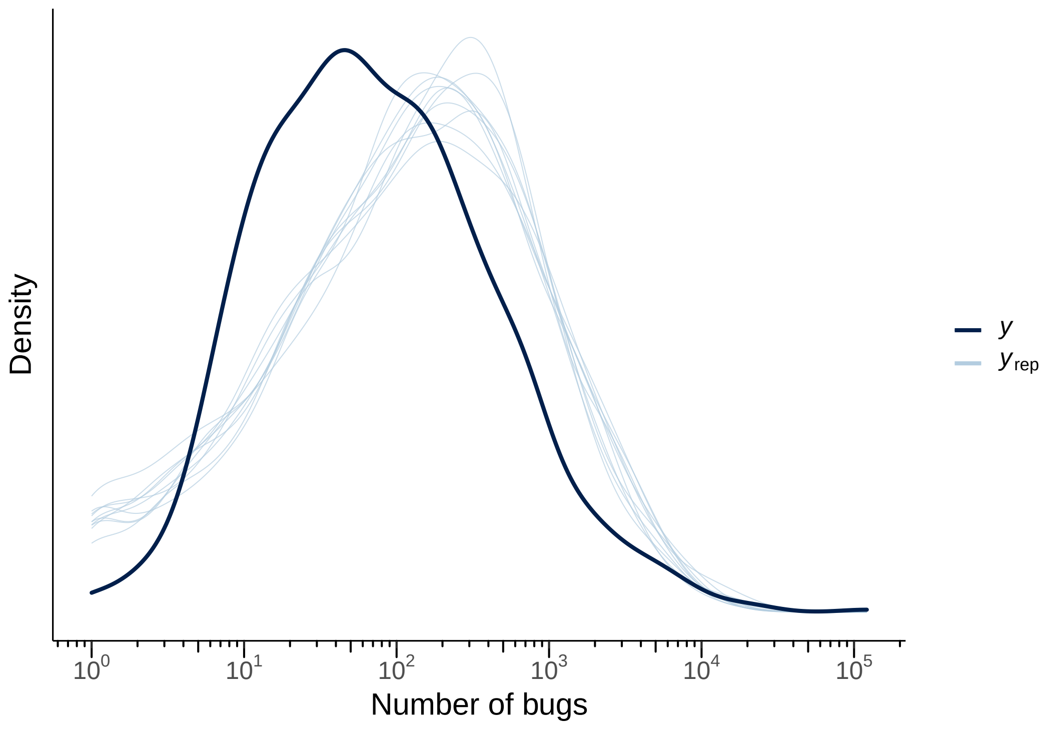
3.6 Model comparison

It is now clear that $\mathcal{M}_1$ is too simplistic, but how to choose between $\mathcal{M}_2$ and $\mathcal{M}_3$? Information criteria, which measure the relative adequacy of different models fitted on the same data, can help answer this question. We use the increasingly popular PSIS-LOO information criterion [44], which works well with models fitted using dynamic Hamiltonian Monte Carlo.[10] In a nutshell, the criterion ranks the three models according to their relative adequacy. It also gives a *difference* score that measures how well each model performs out-of-sample prediction relative to the next one in the ranking, and a *standard error* of the difference, which quantifies how much more adequate a model is compared to another. Table 5 displays the scores for the three models. Model $\mathcal{M}_3$ is ranked first; $\mathcal{M}_2$ comes second but its score is a whopping 7 standard errors worse than $\mathcal{M}_3$'s; and $\mathcal{M}_1$ is, unsurprisingly, a distant last.

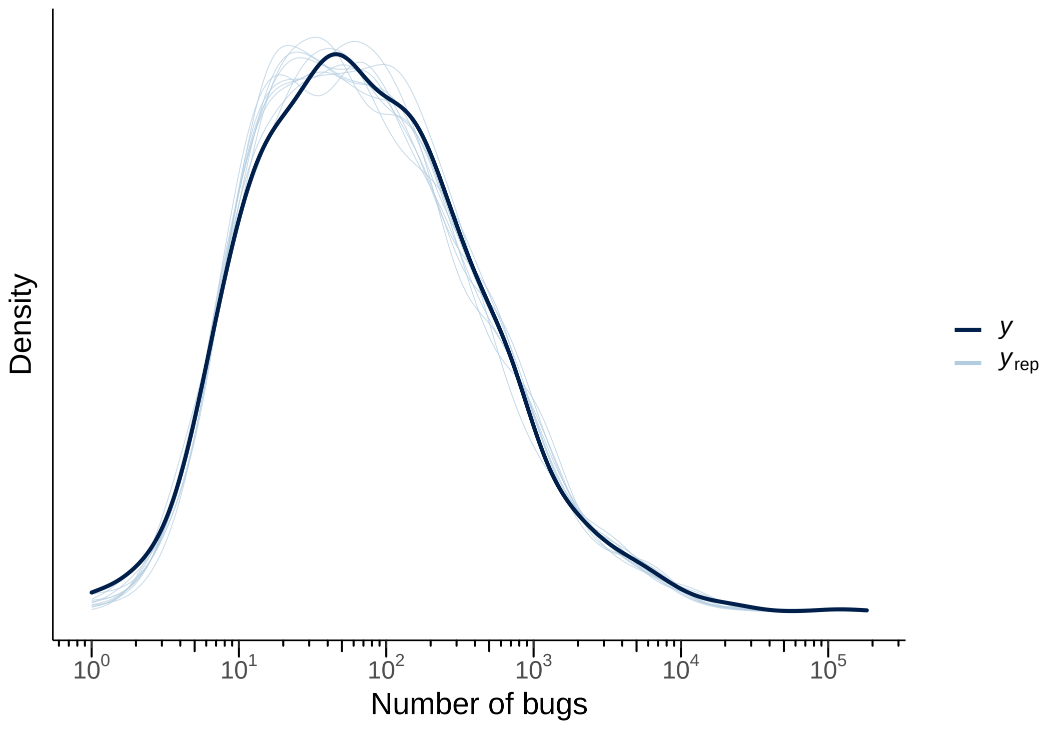
We conclude that $\mathcal{M}_3$ is the "best" model among the three according to a variety of criteria. Our analysis will thus use $\mathcal{M}_3$.

At some point, one has to stop adding model features and finalize a model for the current analysis. Nevertheless, statistical modeling is never really done: as more insights, more data, or new techniques become available, we could go back to the drawing board and refine the latest model to better capture all available information.

[10]Compared to more traditional information criteria—such as AIC, BIC, and WAIC—PSIS-LOO can handle non-Gaussian likelihoods (as can WAIC) and also provides diagnostics useful for further analyzing whether a model's posterior behaves well numerically.



(a) Posterior predictive checks for $\mathcal{M}_1$.



(b) Posterior predictive checks for $\mathcal{M}_2$.

Figure 5: Posterior predictive checks plots for models $\mathcal{M}_1$ and $\mathcal{M}_2$: each thin light blue line pictures one simulated distribution of the number of bugs in a project drawn from the posterior. For comparison, the thick dark blue line pictures the distribution of the number of bugs in the measured data. Model $\mathcal{M}_1$ fails the check because the simulated distributions deviate substantially from the data's. The horizontal scale is logarithmic in base 10.

4 Bayesian statistical analysis: Results

When applied following a structured process—like the one we described in Section 2—Bayesian statistics does not simply produce dichotomous answers to research questions. The outcome of a Bayesian data analysis is a posterior probability distribution, which we can probe from different angles to get nuanced answers that apply to specific scenarios. In this section we are going to do this for our case study.

Section 4.1 discusses how the Bayesian data analysis process guided our choice of *models*: modeling is a looping process, whose feedback also informs us about key characteristics of the data we are analyzing. Not all data features have the same influence on the statistics. Section 4.2 illustrates how Bayesian analysis can point to variables with brittle or negligible predictive power that may indicate problems in how certain measures were operationalized. After understanding the features and limitations of the fitted model, Section 4.3 addresses the original study's research questions in practical settings; and Section 4.4 outlines follow-up studies that could address some of the outstanding limitations.

4.1 Modeling

When following the Bayesian data analysis process in Figure 1, statistical modeling is based on principles and on checks that the models are suitable. This is in contrast to most frequentist statistical practices, which are primarily based on rules of thumb, conventions ("recipes"), and generic results, but may lack operational model-checking processes that can assess how much confidence we can put in a certain modeling choice.

Section 3 described such a principled Bayesian modeling process applied to the programming language data. The first outcome was ruling out $\mathcal{M}_1$ as inadequate, which was unsurprising given that $\mathcal{M}_1$ ignores most of the information that could explain the dataset's variability. Still, even checks with predictable outcomes are useful: if they fail, they confirm that a more realistic model is needed and provide a minimal effectiveness yardstick; if they succeed, they avoid an overly complicated model. This can be especially important in the context of the software engineering industry, where a complex model can be more costly to understand, collect data for, and maintain.

The second outcome of Section 3's analysis was indicating that, while both $\mathcal{M}_2$ and $\mathcal{M}_3$ are adequate, $\mathcal{M}_3$ clearly outperforms $\mathcal{M}_2$ in out-of-sample predictive capabilities. Informally, this means that $\mathcal{M}_3$ fits the data well, while still avoiding overfitting. In other words, $\mathcal{M}_3$'s additional complexity over $\mathcal{M}_2$ is justified by its much better effectiveness. This outcome is specific to the data that we are analyzing, and is not something that can be determined a priori for all models. Iteratively creating multiple models and then comparing them is thus an important part of any analysis.

If we compare the definitions of $\mathcal{M}_2$ and $\mathcal{M}_3$—in particular, their model specifications in Figure 2b and Figure 2c—we can attribute $\mathcal{M}_3$'s superior performance to its unique features. Unlike $\mathcal{M}_2$, which only includes population-level effects that control for project characteristics other than the programming language, $\mathcal{M}_3$ includes a project-specific intercept and controls for the same characteristics with language-specific slopes. Thus, we see that clustering per project *and* per language captures the dataset's characteristics much better: if we do not do that, we may lose some of the "signal" in the data, or conflate different effects and associate them with a single generic predictor.

An indirect advantage of Bayesian data analysis comes from the techniques that are commonly used to fit Bayesian models: flexible algorithmic techniques such as dynamic Hamiltonian Monte Carlo that can fit, in principle, models of arbitrary complexity—in contrast to *ad hoc* frequentist techniques that only work for specific, and often limited, distributional families. On the other hand, Bayesian models are often more effective not simply because they can be more complex. More complex models invariably fit better, but unwarranted complexity leads to *overfitting*: a model fits the data perfectly but fails to generalize. Bayesian analysis techniques include several features that specifically limit the risk of overfitting when exploring more expressive models:

- Multi-level models, such as $\mathcal{M}_3$, introduce *partial pooling*, which smoothens differences between

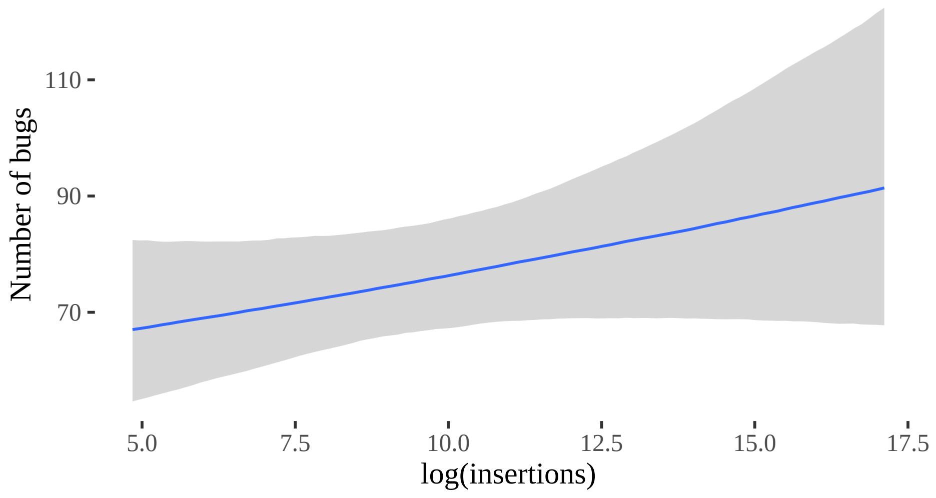


Figure 6: Conditional effects of variable *insertions* (the logarithm of the total number of lines added to a project) on the outcome variable *bugs* (the number of bugs in a project), corresponding to the marginal distribution derived from the posterior of model $\mathcal{M}3$.

groups of different size. In our case study, the data about some programming languages is more scarce than the data about others. For example, only 25 projects use Perl, whereas more than 200 use JavaScript; thus, overfitting Perl's data is a more serious risk than overfitting JavaScript's. Partial pooling works by transfering some of the information learned by fitting the larger groups to tune the fitting of the smaller groups, thus reducing the risk of overfitting the latter (and the whole dataset as a result).

- Prior predictive simulations—discussed in Section 3.5—check that the priors we have chosen are *regularizing*: they are not so constraining that they prevent learning from the data, but they are also not so weak that they cannot prevent overfitting the data. Being able to choose priors, to select different priors, and to quantitatively compare their effectiveness is a distinct advantage of Bayesian statistics. Frequentist statistics usually have flat priors, which are the most prone to overfitting.

- The information criteria that we used to select $\mathcal{M}3$ measure the out-of-sample prediction performance of one model relative to the others. Models that are unnecessarily complex will overfit the data, and hence perform worse predictions for *new* data (different from the sample that has been used for fitting).

4.2 Spotting data problems

The rich information provided by Bayesian data analyses may also highlight issues with the quality of (parts of) the data that is analyzed, and suggest which measures need to be cleaned up or improved.

Measuring size. Code size is a basic yet essential measure of complexity, which correlates with lots of other useful metrics of quality [22]. Therefore, controlling for project size is essential when analyzing heterogeneous projects. To this effect, the FSE study included a variable *size* in their regressive model, which measures the total number of inserted lines in all project commits—and which we called *insertions* in our models to make its actual meaning more transparent. The TOPLAS analysis criticized this choice of size metric—which does not take deletions and merges into account—and reported discrepancies between the raw commit data and the totals in FSE's dataset. Does our Bayesian analysis offer any hints about the reliability of *insertions* as a measure of size?

A few results actually single out *insertions* as a poor predictor compared to the others:

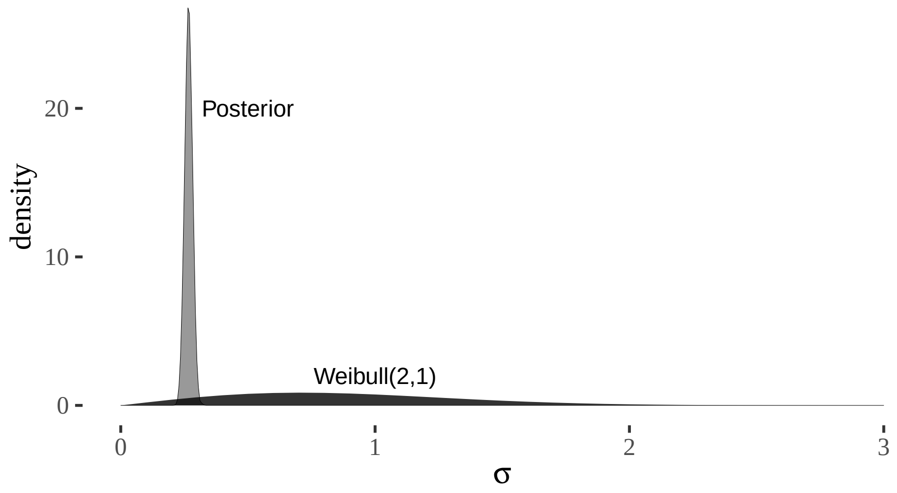


Figure 7: Comparison of prior Weibull$(2, 1)$ and posterior for project-level intercept $\alpha_{project}$'s standard deviation in model $\mathcal{M}_3$. The drastic restriction in uncertainty indicates that the data *swamps* the priors.

- The 95% probability estimate of its population-level effect includes zero (namely, the (credibility) interval is $[-0.01, 0.06]$), which indicates some uncertainty about whether more inserted lines are associated with more or fewer bugs on average. Variable *insertions*'s mean estimated effect is still positive, but other predictors have more clearly defined effects.
- The plot of *insertions*'s conditional effect on the number of *bugs* in Figure 6 visually confirms a large uncertainty (again, compared with the other predictors'), which also increases with larger values of insertions.
- The *varying* effect of *insertions* tend to have larger variance than other predictors for every language.
- If we remove *insertions* from $\mathcal{M}_3$, the resulting model's predictive performance is practically indistinguishable from $\mathcal{M}_3$'s.[11]

All in all, our analysis indicates that *insertions* does not appear to be a particularly useful predictor, and hence it may not be a reliable measure of code size.

Inter-project variability. Section 4.1 showed that a project-specific intercept—which we introduced in $\mathcal{M}_3$—provides better out-of-sample prediction capabilities. The flip side is that several features of the data vary considerably from project to project.

The 1 127 data rows are somewhat sparse among the 729 projects: 64% of all projects appear in a single row; another 26% in two rows. Despite these characteristics, the data *swamps* the priors: it determines a very precise (that is, narrow) posterior distribution—shown in Figure 7—of the project-specific intercept $\alpha_{project}$. In other words, the uncertainty about the contribution of each project to the overall number of bugs is quite limited. Conversely, the bug distributions of randomly drawn projects that we get by simulating from $\mathcal{M}_3$'s fitted posterior differ considerably in shape, support, and mean (see Figure 8). In all, project-specific characteristics are an important source of information in the data, which other control variables cannot fully capture.

[11] Variable selection—an analysis technique that we do not describe in the paper for brevity—also suggests to drop variable *insertions*. See the paper's replication package for details about this additional analysis.

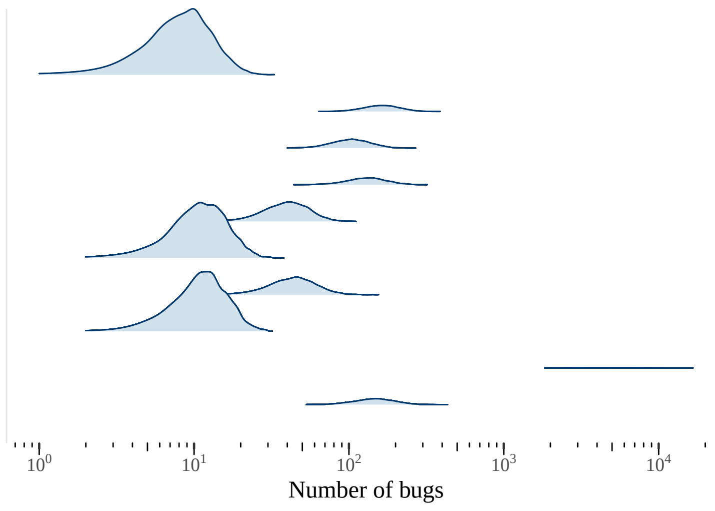


Figure 8: Posterior predictions of the bug distributions of 10 projects drawn randomly from the posterior of $\mathcal{M}_3$. The horizontal axis is logarithmic in base 10. The marked differences in shape and location among the distributions indicate that projects are heterogeneous.

4.3 Practical significance

Let us now address the original study's research questions. For brevity, our analysis won't consider criteria to classify languages ("language classes" such as procedural, functional, scripting, and so on), projects ("application domains" such as application, database, framework, and so on), or bugs ("bug types" such as algorithm, concurrency, performance, and so on). These are largely orthogonal to the main focus of the present paper. Instead, we focus on the key first research question:

RQ. *Are some languages more defect-prone than others?*

Our analysis's information is condensed in the fitted model $\mathcal{M}_3$, which we can use to generate a distribution of bugs for every language. In order to do this, we have to pick the other inputs of the model: the number of commits, insertions, age, and developers of the hypothetical projects whose number of bugs we are estimating. While, in principle, these inputs could be any situation or scenario that we want to investigate, it is sensible to start exploring values that are close to those observed in the data used to fit the model (following the usual assumption that the sample is representative of the entire population).

4.3.1 Ranking all languages

Figure 9 displays the distributions as violin plots for three combinations of input values: the dataset's *median*, *maximum*, and *minimum* number of commits, insertions, age, and developers. Each plot lists the languages in decreasing order of median predicted number of bugs: from most error prone (left) to least (right).

The three plots indicate that that the relative ordering of languages can change conspicuously according to the conditions. For example, C#'s defect proneness is average for projects with large or median size and age; but it becomes better than average for smaller, younger projects. In contrast, C++ is among the second most defect prone languages for median or small projects; whereas it ranks better for large projects. Similarly, the relative rank of some language pairs varies considerably: for example, Erlang is less error prone than Go in large projects; the opposite is true in median and small projects.

A few languages' ranks fluctuate wildly: Objective-C is the most defect prone language except in small projects, when it is among the least; TypeScript even goes from least defect prone on large projects (and second-least on median projects) to most defect prone on small projects. These jumps are so extreme that

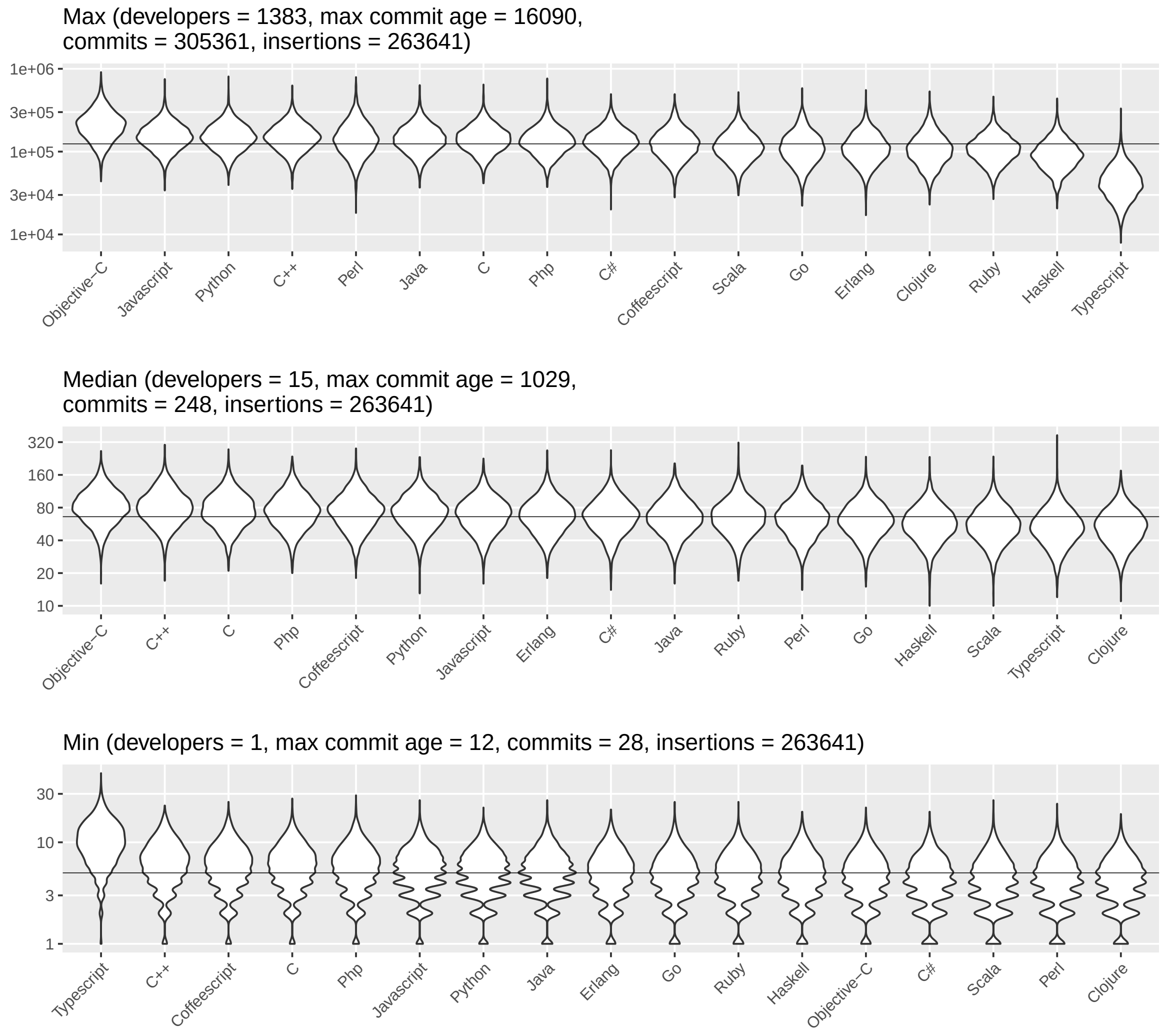


Figure 9: Violin plots of the distributions of number of bugs per language, obtained from the posterior of $\mathcal{M}_3$. Languages are sorted, left-to-right in each sub-plot, by decreasing values of the distributions' medians. The input variables other than *language* are set to the empirical dataset's *maximum* values in the top chart; to the *median* values in the middle chart; and to the *minimum* values in the bottom chart. The vertical axes' scales are logarithmic in base 10. Horizontal lines mark the median number of bugs across all languages.

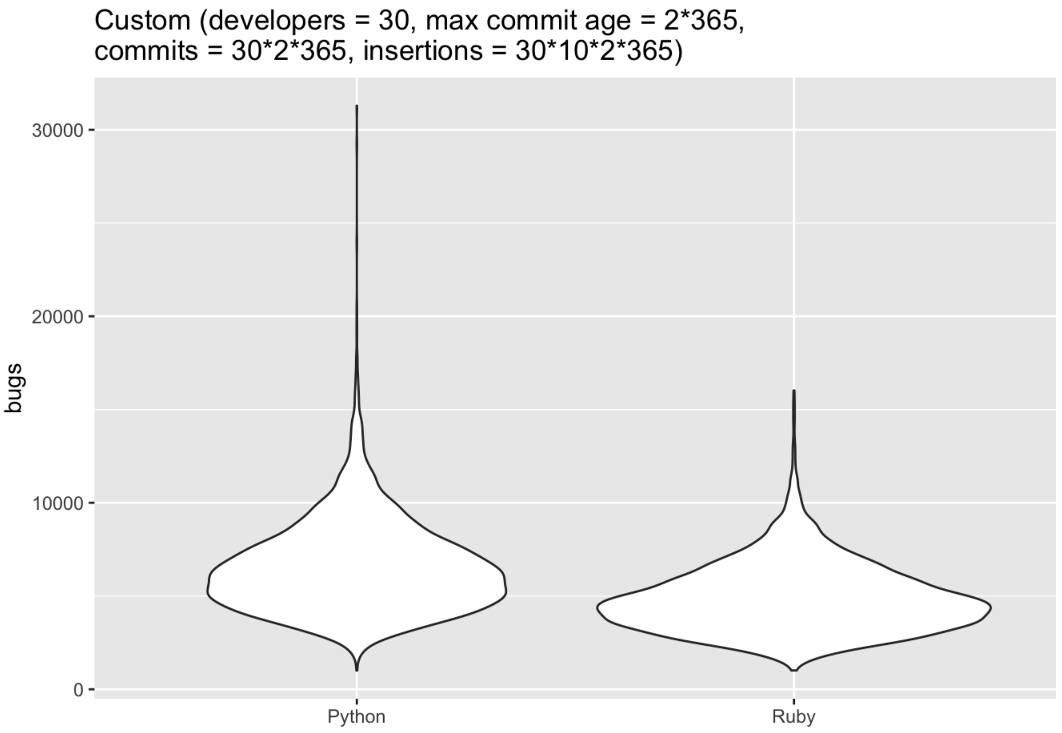


Figure 10: Violin plots of the bug distributions of Python and Ruby obtained from the posterior of $\mathcal{M}_3$. The input variables other than *language* capture a hypothetical project with 30 developers, an age of 2 years, 1 commit per developer every day, and 10 lines added by each developer every day.

they may indicate that the data about these languages is somewhat inconsistent or at least patchwork. Indeed, TOPLAS's reanalysis reported that only about a third of the commits classified as TypeScript in FSE's data actually included TypeScript code; and Ray et al.'s extended version [33] of their original FSE study dropped several projects classified as TypeScript. We did not further look into Objective-C's data, despite its high rank fluctuations, because we wanted to use the original data without changes. Nevertheless, this is one clear example of how Bayesian analysis can help spot data problems, and hence bolster better substantiated analyses. This observation about the fickle influence of some languages also corroborates the evidence that other project-specific characteristics might weigh comparatively more than the used programming language.

Figure 9 also shows that the bug distributions per language are spread out widely—especially for some languages and especially for projects that are large and long-running—and their ranges extensively overlap. The heterogeneity of project-specific characteristics may also contribute to these features; for example, if projects written in language X tend to be on the large side compared to projects written in language Y, the uncertainty in Y's error proneness when used for large projects would dominate the comparison with X. This suggests that the data we analyzed does not warrant summarizing the language differences using a single ranking of defect-proneness.

4.3.2 Custom scenarios

While a single ranking of languages according to their absolute defect-proneness would have little practical meaning, we can still zoom in on specific conditions that are relevant *in practice* for a specific project and see what the fitted model can tell us concerning those conditions.

Imagine, for example, we are planning a project that involves around 30 developers who can code in

Python or Ruby; we estimate the project will run over 2 years, generating an average of 1 commit and 10 lines inserted per programmer per day. Plugging these numbers ($\mathit{developers} = 30$, $\mathit{age} = 2 \times 365$, $\mathit{commits} = 30 \times 1 \times 2 \times 365$, and $\mathit{insertions} = 30 \times 10 \times 2 \times 365$) into the fitted model $\mathcal{M}_3$, we get the estimated bug distributions for Python and Ruby shown in Figure 10. In this scenario, Python tends to be worse (more bugs) than Ruby, since the latter's distribution has a lower mean, a shorter tail towards high number of bugs, and more mass around lower values.

Whether this evidence is sufficiently strong to determine choosing one language over another depends on myriad other factors that are incidental, such as the availability of programmers familiar with one language, the cost of training new ones, the usability of the programming language for the project at hand, and so on. Whatever the practical constraints and requirements may be, the fitted model can help us answer them by providing estimates complete with a quantification of their uncertainty. Realistically, any estimate about the size and development time of a project is also likely to be somewhat uncertain; therefore, we would run *multiple* simulations and weigh the evidence summarized by each one against the confidence we have in the corresponding scenario occurring.

More generally, the results of a principled Bayesian analysis facilitate a quantitatively accurate transfer of knowledge to practitioners and other researchers. Rather than relying only on overly broad conclusions about the impact of different programming languages, using simulations of custom scenarios drives follow-up work more precisely: a practitioner can judge whether the uncertainty in a specific comparison is too large to base a decision on it; a researcher can decide whether more data is needed to claim more general conclusions.

4.3.3 Statistical significance

Our analysis so far has focused on concrete scenarios defined in terms of tangible measures in the data domain—such as number of bugs and project age. In contrast, widespread statistical practices (mostly of a frequentist flavor) try to answer research questions by analyzing *statistical significance*, which measures generic characteristics of a statistical model.

In a standard regression analysis, one usually assesses the *statistical significance* of each coefficient in the model (also called "effect") by checking whether it differs from zero with a certain probability. For example, we could compute the distribution of the estimate of coefficient $\alpha_{language}$ for every language. If α_X is negative with, say, 95% probability, we would conclude that language X is associated with fewer bugs than average with that probability; in other words, X is "statistically significantly" less error prone than other languages.

FSE and TOPLAS both proceed in such a way, but using frequentist coefficient estimates instead of a posterior probability distribution on their models—which are similar to our $\mathcal{M}_2$—leading to their findings about which languages are more error prone than others.[12] What about model $\mathcal{M}_3$ fitted on the same data? The 95% probability intervals of $\alpha_{language}$ include the origin for *every language*, except TypeScript's which is strictly positive—but, as we have commented above, the uncertainty about TypeScript's data puts any results about this language on shaky grounds. Overall, the canonical analysis of statistical significance is just inconclusive on our model.

To some extent, this outcome is a side effect of $\mathcal{M}_3$'s greater complexity over simpler models. There is a trade-off between the complexity of a model (which brings greater expressiveness and better predictive performance) and its interpretability. The criteria we used to choose $\mathcal{M}_3$ over the simpler $\mathcal{M}_2$ ensure that the former's additional complexity is justified by its much better effectiveness. However, a simple interpretation is no longer feasible: $\mathcal{M}_3$ includes slope coefficients that also vary with each language, as well as a project-level contribution; how each language-specific term interacts with the others is not something that can be simply estimated with a single coefficient independent of the predictors' values.

We should appreciate that this is more a feature than it is a limitation. While mathematically simple models are nice to have, not all data analysis problems can be addressed with a basic model. Bayesian

[12]This is explained in detail in TOPLAS's repetition [5, § 2.2.3], which uses FSE's model; TOPLAS's reanalysis [5, § 4.2.1] suggests a different statistical measure.

analysis techniques do not just support fitting complex models but provide the means to handle their complexity and to perform a convincing analysis without resorting to formulaic measures of “significance”. The individual model characteristics are not easy to interpret in isolation, so that we are forced to interpret the model by providing concrete conditions—the number of commits, age, and so on—which ground our generic research question onto scenarios that are realistic and meaningful for our purposes. In other words, it may be cumbersome to reason about statistical significance in a Bayesian model but it is natural to reason about *practical significance*—which is what matters most in the end to answer our research questions.[13]

4.3.4 Effect sizes

Section 4.3.1 argued that the fault proneness of a language over another strongly depends on the conditions in which the languages are to be used. If we have specific scenarios in mind, we can just simulate those as discussed in Section 4.3.2.

Another approach is to compare languages pairwise by simulating their performance on a population that resembles the observed data. Since the comparisons are quantitative—in the form of derived distributions—they can be seen as an effect size, but relative to each language pair instead of absolute for all languages at once.

As usual, simulations are derived from the posterior, which entails that there is no multiple comparisons problem [28]: all information is encoded jointly by the posterior; the pairwise comparisons are just projections of some of that information. For the same reason, we do not have to commit to a certain way of comparing languages when we build the model (for example, by choosing how to encode contrasts): we just select the “best” model according to its performance, and then derive all the information we are interested in from the model fitted on the data.

Concretely, take two languages ℓ_1 and ℓ_2 that we want to compare for bug proneness. For every data point d in the empirical data, we set, in the posterior, all predictors except the language to their values in d. Then, we simulate the distribution of the expected difference $bugs_{\ell_1} - bugs_{\ell_2}$ in bugs produced when using one language over the other.[14]

Figure 11 plots the distributions comparing two pairs of languages, which we selected to demonstrate qualitatively different outcomes of the pairwise comparisons. The distribution of $bugs_{\text{C\#}} - bugs_{\text{C}}$ in Figure 11a covers only nonnegative values, which means C# was consistently more fault prone than C. The distribution of $bugs_{\text{CoffeeScript}} - bugs_{\text{Go}}$ in Figure 11b covers negative values more often than positive ones, denoting that Go tends to be more fault prone. However, we can compute that CoffeeScript is more error prone around 8.5% of the times.

In the end, we did not really answer the original research question—not with a definitive, straightforward answer at least. Instead, our analysis identified sources of uncertainty in the data, provided means of simulating custom scenarios, and compared pairs of languages in conditions similar to the collected data’s. This is a solid basis to understand what questions can and cannot be answered by the data, and to plan follow-up data collections and analyses that zero in on understanding specific outcomes.

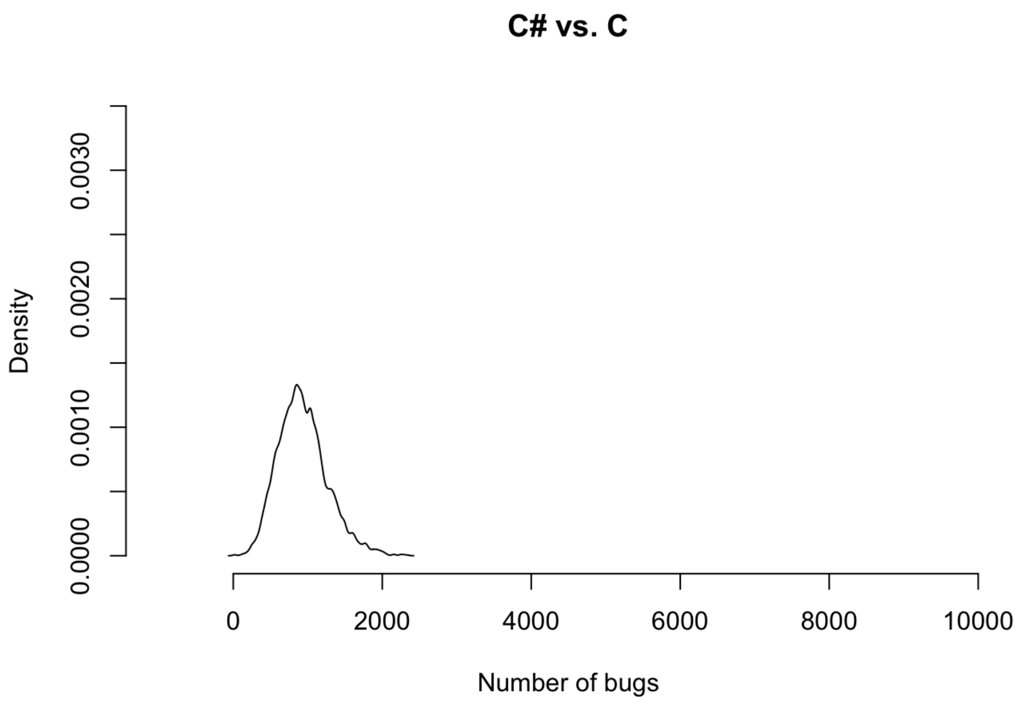
4.4 Planning the next study

Our analysis shed light on the the relationship between programming languages and fault proneness, but also discovered restrictions on how general the findings can be and which factors should be considered. How can we make further progress in this line of research—beyond the limitations of what is available in FSE’s dataset?

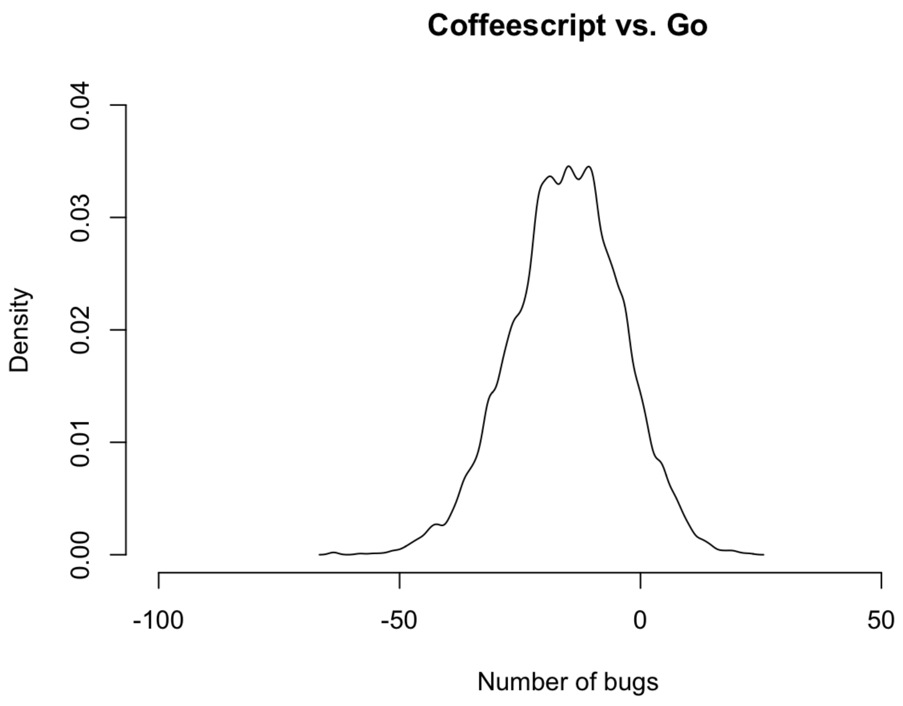
The outcome of our analysis—in particular, the issues discussed in Section 4.2—help to plan follow-up studies too. A recurring issue was the clear impact of project-specific characteristics, which sometimes

[13]In related work, we discuss in greater detail methods to analyze practical significance based on Bayesian data analysis [42].

[14]We can compute the absolute difference in number of bugs because the difference is between samples where the language is the only project characteristics that changes.



(a) Posterior distribution of the difference $bugs_{\text{C\#}} - bugs_{\text{C}}$ when predictors are set to the same values as in the empirical data. This indicates that C# is consistently more fault-prone than C, since it leads to more bugs.



(b) Posterior distribution of the difference $bugs_{\text{CoffeeScript}} - bugs_{\text{Go}}$ when predictors are set to the same values as in the empirical data. This indicates that Coffeescript is somewhat less fault-prone than Go.

Figure 11: Probability distributions of the difference in bug proneness between pairs of languages according to the posterior distribution with population data.

dominate over language-specific features. There are at least two ways of better accounting for project features. One is collecting more data that characterize projects along more dimensions; for example, a project's domain, the development process it uses, the expertise of its developers, and so on. The other way is to give up generality and focus on analyzing a specific, homogeneous set of projects: the more characteristics are similar among projects the more accurately the impact of programming languages can be singled out.

When we simulated different scenarios in Section 4.3, we found that the *uncertainty* in the outcome is more pronounced for certain languages than others. For example, the uncertainty about Objective-C's and Perl's fault proneness is very pronounced on large projects (see the vertical spread in their violin plots of Figure 9). To reduce this uncertainty, we should collect more data on large Objective-C and Perl projects, focus on smaller projects, or a combination of both.

More generally, Bayesian models and techniques help zoom out of each individual study to considering a line of studies in the same subject area. Each study collects additional data, refines the knowledge that we have of the area, and identifies further aspects that can be improved—something follow-up studies will do. In a way, this realizes a sort of optimization process whose goal is maximizing knowledge over time. Bayesian optimization algorithms exist that carry out this process automatically on a large dataset that can be analyzed incrementally [35]; scientific research deploys processes that do something similar on a much longer time scale and with key contributions from human intuition. The benefits we highlight are not only conceptual; the posteriors from previous studies can be directly used when creating priors for follow-up studies. This can thus enable a more direct and precise way for research studies to build on each other and gradually refine the scientific knowledge.

5 Related work

Bayesian data analysis guidelines. The last decade's progress in algorithms and tools for Bayesian data analysis has been impressive [9; 16; 31; 25] but, without practical support, it is not sufficient to promote widespread usage in the empirical sciences. Recent work about developing guidelines to apply Bayesian data analysis techniques [15; 36; 21], which Section 2 summarized in a form amenable to software engineering empirical research, has been trying to close this gap. While these proposals differ in their intended audience and level of detail, they all build on the basic view [17] that an analysis should go through multiple models, refine them in several iterations, and compare them. Then, Gabry et al. [15] focus on visualization and how to use it throughout a workflow; Schad et al. [36], instead, introduce quantitative checks and illustrate them for a specific scientific area (the cognitive sciences). Our guidelines combine elements from both [15; 36] but illustrate them in a way that is amenable to empirical software engineering research practices. Very recently, Gelman et al. [21] presented an early draft of a book that will further refine some of these Bayesian analysis workflows and guidelines. Also very recently, van de Schoot et al. [43] published an accessible primer on Bayesian statistics and modeling for scientists.

Guidelines on using statistics in empirical software engineering. Over the years, several guidelines for using statistics in empirical software engineering have been proposed—all of them focusing on frequentist statistics, which remain the norm in empirical software engineering [11]. Arcuri and Briand [2] focus on analyzing experiments with randomized algorithms, and highlight the importance of checking the assumptions of each statistical significance test. They also advocate for extensively using non-parametric statistical tests and effect size measures. Menzies and Shepperd [27] catalog "bad smells" in data analytics studies and discuss remedies to excise them. Among the techniques they recommend are up-front power analysis, reporting effect sizes and confidence limits, and using robust statistics and sensitivity analysis. A recent literature review of ours [11] found the positive impact of such empirical guidelines on the maturity of statistical practice in empirical software engineering research: statistical testing, non-parametric tests, and effect sizes have all been increasingly used in the field over the last 5–10 years.

Replication in software engineering research. Recent years have finally seen replication studies become more popular in software engineering research. Nevertheless, Da Silva et al.'s systematic literature reviews found that internal replications (done by the same authors as the original study) are still much more common than external replications (done by an independent group of authors) [10]. Unsurprisingly, Bezerra et al.'s related literature review found that internal replications are much more likely to confirm the results of the replicated study than external replications [6], and used this result to question the value of replications compared to meta-analyses. Both literature reviews found hardly any examples of *reanalyses* (replications limited to data analysis); similarly, a taxonomy for replications in software engineering does not explicitly mention reanalysis [3].

In fact, we tried searching for "*reanalysis + software engineering*" in publication databases and found very few relevant hits—mostly papers revisiting qualitative data such as interview transcripts, and reanalyzing them to address new questions or theories. As one example, Bjarnason et al. [7] developed a new theory by reanalyzing interview transcripts from an earlier study of theirs. In contrast, Tantithamthavorn et al. [41] revised a meta-analysis of machine learning in software defect prediction [38] and found that several predictor variables of the original study where co-linear. Based on a reanalysis of a subset of the same data, they also questioned some of the original results and implications. This criticism was later disputed, on statistical grounds, by the original study's authors [39]. Our previous work about using Bayesian analysis in empirical software engineering also performed reanalyses of previous studies using Bayesian techniques [13; 42]. Another noticeable external reanalysis is of course Berger et al. [5]'s of Ray et al. [34], which we summarized in Section 1.

Overall, reanalyses of software engineering data remain uncommon—especially compared to other scientific areas where they are widespread forms of publication, including those using Bayesian statistics (for example, in astronomy [23] and medicine [4]).

6 Conclusions

Reaping the benefits of Bayesian statistics requires more than powerful analysis techniques and tools. In this paper, we presented practical guidelines to build, check, and analyze a Bayesian statistical model that summarize recently developed suggestions brought forward by prominent statisticians and cast them in a format that is amenable to empirical software engineering research.

We then applied the guidelines to analyze a large dataset of GitHub projects that was previously used to study the impact of programming languages on code quality [34]. This study was later criticized by a reproduction attempt that failed to confirm some of the originally claimed results [5]. Our reanalysis using Bayesian statistics identified some shortcomings of the data that also emerged in the reproduction attempt (such as the large uncertainty associated with data for programming languages such as TypeScript) and pointed to other possible effects that were not fully accounted for by the frequentist models of the previous studies [34; 5] (such as the disproportionate differences that are project-specific rather than language-specific). Moving on to the previous studies' main research question ("Are some languages more defect-prone than others?"), our Bayesian model lent itself to evaluating the effect of programming languages in different concrete scenarios rather than in terms of generic "statistical significance". We found that the impact of programming languages can vary considerably with other contextual conditions, and hence the original research question does not admit a simple, generally valid answer—at least not with the analyzed data.

Throughout our reanalysis, a key advantage of Bayesian techniques was that they can be used to *quantify* any derived measures of interest, as well as the *uncertainty* that comes with each measure. Such capabilities are useful not only to infer results in each study, but also to present and share them in a robust way with other researchers and practitioners. A Bayesian quantitative framework focused on practical significance can also help plan the next studies in a research area—thus steadying the long-term progress of software engineering empirical research and enhancing its broader impact.

References

[1] Balazs Aczel, Rink Hoekstra, Andrew Gelman, Eric-Jan Wagenmakers, Irene G. Klugkist, Jeffrey N. Rouder, Joachim Vandekerckhove, Michael D. Lee, Richard D. Morey, Wolf Vanpaemel, Zoltan Dienes, and Don van Ravenzwaaij. 2020. Discussion points for Bayesian inference. *Nature Human Behaviour* 4, 6 (2020), 561–563. `https://doi.org/10.1038/s41562-019-0807-z`

[2] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Hawaii, USA, 1–10. `https://doi.org/10.1145/1985793.1985795`

[3] Maria Teresa Baldassarre, Jeffrey C. Carver, Oscar Dieste, and Natalia Juristo Juzgado. 2014. Replication types: Towards a shared taxonomy. In *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14*, Martin J. Shepperd, Tracy Hall, and Ingunn Myrtveit (Eds.). ACM, London, UK, 18:1–18:4. `https://doi.org/10.1145/2601248.2601299`

[4] Philip M. W. Bath. 2007. Can we improve the statistical analysis of stroke trials? Statistical re-analysis of functional outcomes in stroke trials. *Stroke* 38, 6 (2007), 1911–1915. `https://doi.org/10.1161/STROKEAHA.106.474080`

[5] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the Impact of Programming Languages on Code Quality: A Reproduction Study. *ACM Transactions on Programming Languages and Systems* 41, 4 (2019), 21:1–21:24. `https://doi.org/10.1145/3340571`

[6] Roberta M. M. Bezerra, Fabio Q. B. da Silva, Anderson M. Santana, Cleyton V. C. de Magalhães, and Ronnie E. S. Santos. 2015. Replication of Empirical Studies in Software Engineering: An Update of a Systematic Mapping Study. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2015, Beijing, China, October 22-23, 2015*. IEEE Computer Society, Beijing, China, 132–135. `https://doi.org/10.1109/ESEM.2015.7321213`

[7] Elizabeth Bjarnason, Kari Smolander, Emelie Engström, and Per Runeson. 2016. A theory of distances in software engineering. *Information and Software Technology* 70 (2016), 204–219. `https://doi.org/10.1016/j.infsof.2015.05.004`

[8] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. 2011. *Handbook of Markov Chain Monte Carlo*. CRC press, Florida, USA.

[9] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76, 1 (2017), 1–32.

[10] Fabio QB Da Silva, Marcos Suassuna, A César C França, Alicia M Grubb, Tatiana B Gouveia, Cleviton VF Monteiro, and Igor Ebrahim dos Santos. 2014. Replication of empirical studies in software engineering research: A systematic mapping study. *Empirical Software Engineering* 19, 3 (2014), 501–557. `https://doi.org/10.1007/s10664-012-9227-7`

[11] Francisco Gomes de Oliveira Neto, Richard Torkar, Robert Feldt, Lucas Gren, Carlo A. Furia, and Ziwei Huang. 2019. Evolution of statistical analysis in empirical software engineering research: Current state and steps forward. *Journal of Systems and Software* 156 (2019), 246–267. `https://doi.org/10.1016/j.jss.2019.07.002`

[12] Robert Feldt and Ana Magazinius. 2010. Validity threats in empirical software engineering research-an initial survey. In *International Conference on Software Engineering and Knowledge Engineering*. 374–379.

[13] Carlo A. Furia, Robert Feldt, and Richard Torkar. 2019. Bayesian data analysis in empirical software engineering research. *IEEE Transactions on Software Engineering* -, - (2019), 1–1. https://doi.org/10.1109/TSE.2019.2935974

[14] Carlo A. Furia, Richard Torkar, and Robert Feldt. 2021. *Replication Package*. https://doi.org/10.5281/zenodo.4472963

[15] Jonah Gabry, Daniel Simpson, Aki Vehtari, Michael Betancourt, and Andrew Gelman. 2019. Visualization in Bayesian workflow. *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 182, 2 (2019), 389–402. https://doi.org/10.1111/rssa.12378

[16] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A language for flexible probabilistic inference. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018*. MLResearchPress, Canary Islands, Spain, 1682–1690.

[17] Andrew Gelman. 2004. Exploratory data analysis for complex models. *Journal of Computational and Graphical Statistics* 13, 4 (2004), 755–779.

[18] Andrew Gelman and Jennifer Hill. 2007. *Data analysis using regression and multilevel/hierarchical models*. Vol. Analytical methods for social research. Cambridge University Press, Cambridge, UK. xxii, 625 p pages.

[19] Andrew Gelman, Jennifer Hill, and Aki Vehtari. 2020. *Regression and other stories*. Cambridge University Press, Cambridge, UK. https://books.google.se/books?id=SZFKzQEACAAJ

[20] Andrew Gelman, Daniel Simpson, and Michael Betancourt. 2017. The prior can often only be understood in the context of the likelihood. *Entropy* 19, 10 (Oct 2017), 555. https://doi.org/10.3390/e19100555

[21] Andrew Gelman, Aki Vehtari, Daniel Simpson, Charles C. Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák. 2020. Bayesian workflow. arXiv:2011.01808 [stat.ME]

[22] Yossi Gil and Gal Lalouche. 2017. On the correlation between size and metric validity. *Empirical Software Engineering* 22, 5 (Oct. 2017), 2585–2611. https://doi.org/10.1007/s10664-017-9513-5

[23] Philip C Gregory. 2011. Bayesian re-analysis of the Gliese 581 exoplanet system. *Monthly Notices of the Royal Astronomical Society* 415, 3 (2011), 2523–2545.

[24] Edwin T. Jaynes. 2003. *Probability theory: The logic of science*. Cambridge University Press, Cambridge.

[25] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. 2009. The BUGS project: Evolution, critique and future directions. *Statistics in medicine* 28, 25 (2009), 3049–3067.

[26] Richard McElreath. 2020. *Statistical rethinking: A Bayesian course with examples in R and Stan* (2 ed.). CRC press, Florida, USA.

[27] Tim Menzies and Martin Shepperd. 2019. "Bad smells" in software analytics papers. *Information and Software Technology* 112 (2019), 35–47. https://doi.org/10.1016/j.infsof.2019.04.005

[28] Rupert G. Miller. 1981. *Simultaneous statistical inference* (2nd ed.). Springer-Verlag, Berlin, Heidelberg.

[29] Radford M. Neal. 1996. *Bayesian Learning for Neural Networks*. Springer-Verlag, Berlin, Heidelberg.

[30] Juho Piironen, Markus Paasiniemi, and Aki Vehtari. 2020. Projective inference in high-dimensional problems: Prediction and feature selection. *Electronic Journal of Statistics* 14, 1 (2020), 2155–2197. https://doi.org/10.1214/20-EJS1711

[31] Martin Plummer. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd International Workshop on Distributed Statistical Computing: Workshop on Distributed Statistical Computing*. Achim Zeileis, Vienna, Austria, 10 pages.

[32] Paul Ralph and Ewan Tempero. 2018. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018* (Christchurch, New Zealand) *(EASE'18)*. Association for Computing Machinery, New York, USA, 13–23. `https://doi.org/10.1145/3210459.3210461`

[33] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. 2017. A large-scale study of programming languages and code quality in GitHub. *Commun. ACM* 60, 10 (Sept. 2017), 91–100. `https://doi.org/10.1145/3126905`

[34] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in Github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 155–165. `https://doi.org/10.1145/2635868.2635922`

[35] Bin Xin Ru, Mark McLeod, Diego Granziol, and Michael A. Osborne. 2018. Fast information-theoretic Bayesian optimisation. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, Stockholm, Sweden, 4381–4389. `http://proceedings.mlr.press/v80/ru18a.html`

[36] Daniel J. Schad, Michael Betancourt, and Shravan Vasishth. 2020. Toward a principled Bayesian workflow in cognitive science. *Psychological methods* -, - (June 2020), –. `https://doi.org/10.1037/met0000275`

[37] Maximilian Scholz and Richard Torkar. 2020. An empirical study of Linespots: A novel past-fault algorithm. *arXiv e-prints* -, -, Article arXiv:2007.09394 (July 2020), 19 pages. arXiv:2007.09394 [cs.SE]

[38] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering* 40, 6 (2014), 603–616. `https://doi.org/10.1109/TSE.2014.2322358`

[39] Martin Shepperd, Tracy Hall, and David Bowes. 2017. Authors' Reply to "Comments on 'Researcher Bias: The Use of Machine Learning in Software Defect Prediction"'. *IEEE Transactions on Software Engineering* 44, 11 (2017), 1129–1131. `https://doi.org/10.1109/TSE.2017.2731308`

[40] Sean Talts, Michael Betancourt, Daniel Simpson, Aki Vehtari, and Andrew Gelman. 2018. Validating Bayesian inference algorithms with simulation-based calibration. *arXiv preprint arXiv:1804.06788* (2018).

[41] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Comments on "Researcher bias: the use of machine learning in software defect prediction". *IEEE Transactions on Software Engineering* 42, 11 (2016), 1092–1094. `https://doi.org/10.1109/TSE.2016.2553030`

[42] Richard Torkar, Carlo A. Furia, Robert Feldt, Francisco Gomes de Oliveira Neto, Lucas Gren, Per Lenberg, and Neil A. Ernst. 2021. A method to assess and argue for practical significance in software engineering. *Transactions on Software Engineering* -, - (2021), 13 pages. `https://doi.org/10.1109/TSE.2020.3048991`

[43] Rens van de Schoot, Sarah Depaoli, Ruth King, Bianca Kramer, Kaspar Märtens, Mahlet G. Tadesse, Marina Vannucci, Andrew Gelman, Duco Veen, Joukje Willemsen, and Christopher Yau. 2021.

Bayesian statistics and modelling. *Nature Reviews Methods Primers* 1, 1 (14 Jan 2021), 1. `https://doi.org/10.1038/s43586-020-00001-2`

[44] Aki Vehtari, Andrew Gelman, and Jonah Gabry. 2017. Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC. *Statistics and Computing* 27, 5 (2017), 1413–1432. `https://doi.org/10.1007/s11222-016-9696-4`

[45] Sumio Watanabe. 2010. Asymptotic equivalence of Bayes cross validation and widely applicable information criterion in singular learning theory. *Journal of Machine Learning Research* 11 (Dec. 2010), 3571–3594.