# What Good Is Bayesian Data Analysis for Software Engineering?

Carlo A. Furia
Chalmers University of Technology, Gothenburg, Sweden
furia@chalmers.se bugcounting.net

Empirical software engineering, like every rigorous empirical discipline, uses *statistics* for two purposes: summarizing data; and assessing whether data support a hypothesized model. The second purpose is more subtle because it has to address the problem of *induction*: generalizing a widely applicable model from specific observational data.

A bunch of influential statisticians approached the problem head on at the beginning of the twentieth century, and developed a collection of readily applicable techniques that collectively go under the name *statistical hypothesis testing*. Thanks to the prominent position of those statisticians in the scientific community, as well as to their and others' incisive popularization effort, statistical hypothesis testing has become a prevalent practice in pretty much all experimental sciences.

Only in the last few decades have we started to realize that statistical hypothesis testing techniques—at least in the ways in which they have normally been used—have serious theoretical and practical shortcomings, which make them prone to drawing incorrect conclusions [5], and ill-suited to address the induction problem. This short text outlines the problems with classical statistical hypothesis testing, and recommends using alternative techniques based on *Bayesian statistics*, which are largely immune to the shortcomings of statistical hypothesis testing, and support a robust induction process.

## I. How Statistical Hypothesis Testing Works

Let us outline how statistical hypothesis testing works on a concrete example from my previous research.

The Rosetta Code study [7] compared the features of solutions to programming tasks written in different programming language. One of the experiments, which I single out here merely as an interesting example, ran Haskell and Python solutions to 28 problems on the same inputs and compared their running time performance. The statistical summaries of these experiments show a mixed picture: Haskell was faster in 15 problems (a narrow majority), with a median speedup of 1.3 compared to Python; however, the mean speedup is 27 *in favor of Python*, due to two problems where Haskell was particularly inefficient. How to decide whether this data indicates a genuine speed advantage of one language over another?

Statistical hypothesis testing provides means precisely to answer such questions. While the details vary from test to test, the general approach is the same: given two sets of data, a statistical test computes a statistics called the $p$-value. Roughly speaking, the $p$-value is the *probability* of observing the two given data sets under the assumption of no difference—the so-called null hypothesis. In our example, the data sets are the running times of Haskell and Python on each problem, and the $p$-value is the probability of recording those running times assuming there is no fundamental performance difference between Haskell and Python—and thus any observed difference is due to chance.

The way statistical testing are normally used, a small $p$-value (typically $p < 0.05$) leads to rejecting the null hypothesis and to concluding that the empirically measured difference is *statistically significant*; conversely, a large $p$-value suggests that the empirically measured difference may be due to chance alone, and thus either the experiment is inconclusive or the differences are insignificant. The Wilcoxon signed-rank test is a suitable hypothesis test for our language comparison; applied to the Haskell vs. Python data set, it gives a $p$-value of 0.33, and thus the data does not warrant claiming any significant difference between Haskell's and Python's performance.

## II. What is Wrong with Statistical Hypothesis Testing

Upon closer look, the analysis based on statistical hypothesis testing leaves much to be desired. The most egregious issue is that the $p$-value is the probability $\mathbb{P}[data \mid H_0]$ of the data given the null hypothesis $H_0$; however, to accept or reject hypothesis $H_0$ we really need the other probability $\mathbb{P}[H_0 \mid data]$. In general $\mathbb{P}[H_0 \mid data]$ and $\mathbb{P}[data \mid H_0]$ may be very different, so knowing the $p$-value alone is of no help to accept or reject a hypothesis based on experimental data!

Even assuming specific conditions ensure that the $p$-value is related to the probability that the data support a hypothesis, deciding whether to accept or reject a model using a fixed probability threshold—any threshold, really—risks being arbitrary. There is nothing special about a 1%, 5%, or 10% probability that ensures that borderline cases are treated consistently. Another problem is that a small $p$-value leads to rejecting a hypothesis, but it does not necessarily help find out which alternative hypothesis holds. Going back to the Haskell vs. Python comparison, even if statistical hypothesis testing had told us to reject the hypothesis that Haskell and Python are as fast, we would still be left with the problem of deciding which is faster: Haskell (as suggested by the median) or Python (as suggested by the mean).

Statisticians have been well aware of these intrinsic shortcomings of statistical hypothesis testing; over decades, they

have developed a number of techniques, such as effect sizes, confidence intervals, and statistical regression, that can help improve the reliability of statistical analysis. All of these techniques are still largely rooted in the same basic principles as statistical hypothesis testing, constituting the corpus of so-called *frequentist statistics*. A distinct sets of approaches, which we advocate here, relies on the distinct set of techniques known as *Bayesian statistics* whose peculiar advantages we outline in the remainder.

## III. Bayesian Statistics in a Nutshell

Bayesian statistics directly addresses the problem of induction based on Bayes theorem of conditional probabilities:

$$\mathbb{P}[H \mid D] = \frac{\mathbb{P}[D \mid H] \cdot \mathbb{P}[H]}{\mathbb{P}[D]},$$

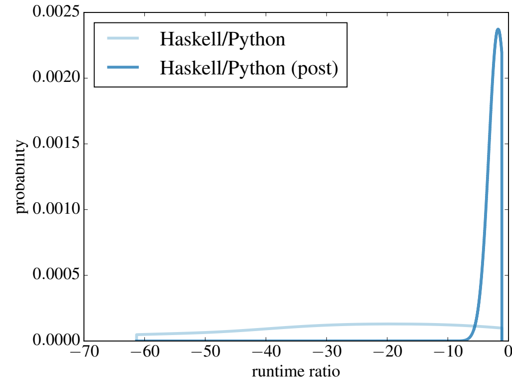
which relates the probability $\mathbb{P}[D \mid H]$ of observing data $D$ under hypothesis $H$ to the "converse" probability $\mathbb{P}[H \mid D]$ that hypothesis $H$ holds given observed data $D$.

Rather than narrowly targeting the null hypothesis $H_0$, Bayes theorem easily applies to a complete probability distribution over possible observations. In the case of the language performance comparison, the goal is estimating the *posterior* probability $\mathbb{P}[\textit{haskell-to-python speedup} \mid \textit{data}]$, where *data* are the measured speedups—following the convention that a negative speedup $-s$ indicates that Haskell is $s$ times faster than Python, and a positive speedup $+s$ indicates that Python is $s$ times faster than Haskell.

In addition to the experimental *data*, applying Bayes theorem requires knowing the unconditional probability $\mathbb{P}[H]$—in the running example, a probability distribution over the *haskell-to-python speedup* before performing the experiments. This probability is called the *prior*, and it is a way in which Bayesian statistics may incorporate additional information—for example other studies on the same subject. If we prefer not to bias the probability estimation by the results of other studies, or simply if no reliable prior information is available, we can pick an *uninformative* prior which assigns uniform probability. For the running example, the Computer Language Benchmarks Game [8] includes benchmark data comparable to the Rosetta Code study, and thus we use its distribution of speedups as prior. For Haskell and Python, its prior distribution has mean and median around $-24$, which indicate that Haskell has consistently been much faster than Python in [8]'s 12 benchmarks.

The next key ingredient to apply Bayes theorem is the distribution $\mathbb{P}[D \mid H]$—called *likelihood* in Bayesian lingo—which quantifies a probability over *differences* between observations and expectations. For the running example, it amounts to an estimate of $\mathbb{P}[s \mid t]$—the probability of observing a certain speedup $s$ given that a speedup $t$ was expected (the hypothesis). We base the likelihood on an extended data set from [8], which suggests a range of variability due to different experimental choices in comparing Haskell and Python (such as input size and kind of algorithm for the same problem).

The probability $\mathbb{P}[D]$ can be derived from the prior and likelihood as a normalization factor without additional information. By doing this in the running example, we obtain the following *posterior* distribution of speedups (runtime ratios):




A distinctive advantage of Bayesian statistics is that it estimates a complete *distribution* of data. In the picture, we can easily see that the distribution's support only comprises negative speedups, which correspond to Haskell being consistently faster than Python. Additionally, we see how the application of Bayes theorem provides a rigorous way to integrate apparently contradictory—or otherwise not completely consistent—evidence obtained in different experimental conditions. The prior (which comes from [8]) indicated that Haskell is usually much faster than Python; but the new data (which comes from [7]) was inconclusive as to which is faster. By combining them, Bayes theorem strongly supports the hypothesis that Haskell is faster than Python, but also suggests that the average speedup may be not great.

## IV. Bayesian Statistics in Software Engineering?

The limitations of statistical hypothesis testing, as well the potential advantages of using Bayesian analysis, have been often noted in other experimental disciplines [2], [5], [4]. Nonetheless, statistical hypothesis testing is still extensively used in practice, while Bayesian analysis remains more of a specialized knowledge; empirical software engineering, in particular, still has to incorporate Bayesian techniques in its recommended data analysis practices [9], [1], [6].

Answering the titular question in the affirmative, I argued that there is much to be gained by introducing Bayesian analysis in empirical software engineering practices:

- Bayes theorem correctly addresses the problem of induction—unlike the defective hypothesis testing;
- estimating complete distributions rather than single probabilities supports analysis of nuanced cases;
- priors and likelihoods provide means to naturally incorporate the results of other studies, in a way that behooves the scientific method.

A technical report [3] offers a more rigorous presentation of Bayesian techniques, with three full-fledged case studies from empirical software engineering research.

## REFERENCES


[1] Andrea Arcuri and Lionel C. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test., Verif. Reliab.*, 24(3):219–250, 2014.

[2] Jacob Cohen. The earth is round ($p < .05$). *American Psychologist*, 49(12):997–1003, 1994.

[3] Carlo A. Furia. Bayesian statistics in software engineering: Practical guide and case studies. http://arxiv.org/abs/1608.06865, August 2016.

[4] Steven N. Goodman. Toward evidence-based medical statistics. 1: The $p$ value fallacy. *Annals of Internal Medicine*, 130(12):995–1004, 1999.

[5] John P. A. Ioannidis. Why most published research findings are false. *PLoS Med*, 2(8), 2005.

[6] Andreas Jedlitschka, Natalia Juristo Juzgado, and H. Dieter Rombach. Reporting experiments to satisfy professionals' information needs. *Empirical Software Engineering*, 19(6):1921–1955, 2014.

[7] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in Rosetta Code. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 778–788. ACM, 2015.

[8] The computer language benchmarks game. http://benchmarksgame.alioth.debian.org/, Aug 2016.

[9] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012.