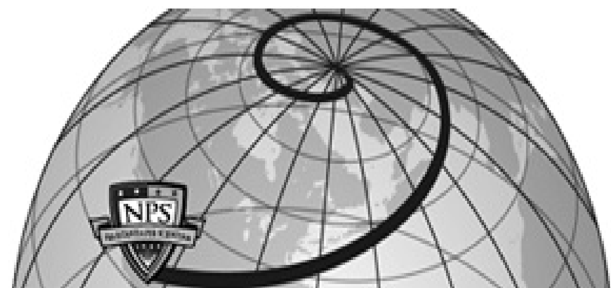
Calhoun
Institutional Archive of the Naval Postgraduate School



Calhoun: The NPS Institutional Archive

DSpace Repository

NPS Scholarship | Conferences

2018-04-30

Experience Searching for Causal Factors in Personal Process Student Data

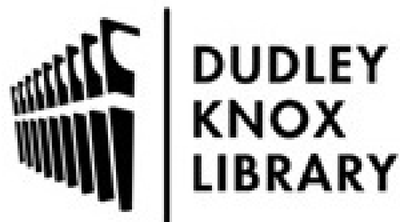
Nichols, William R. Jr.; Konrad, Michael

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/58780

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

SYM-AM-18-096

PROCEEDINGS
OF THE
FIFTEENTH ANNUAL ACQUISITION RESEARCH SYMPOSIUM

THURSDAY SESSIONS
VOLUME II

Acquisition Research:
Creating Synergy for Informed Change

May 9–10, 2018

March 30, 2018

Approved for public release; distribution is unlimited.

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL

Experience Searching for Causal Factors in Personal Process Student Data

William R. Nichols, Jr.—joined the Software Engineering Institute (SEI) in 2006. He serves as Team Software Process (TSP) Mentor Coach and maintains PSP and TSP Software Engineering Data. Prior to joining the SEI, he led a team at the Bettis Laboratory near Pittsburgh, PA, developing and maintaining nuclear engineering and scientific software. Dr. Nichols's research interests include software engineering project planning, process modeling, and economics. He received a PhD in physics at Carenegie Mellon University. [wrn@sei.cmu.edu]

Michael Konrad—is a Principal Researcher at the SEI, providing analytic support to various projects at the SEI using statistics, machine learning, and most recently, causal learning. Since 2013, Dr. Konrad has contributed to research in requirements engineering, software architecture, and system complexity measurement. From 1998 to 2013, he contributed to CMMI in many technical leadership roles. Prior to 1998, Dr. Konrad was a member of the teams that developed the original Software CMM and ISO 15504. He is coauthor of the Capability Maturity Model Integration for Development (CMMI-DEV). Dr. Konrad received his PhD in mathematics from Ohio University in 1978. [mdk@sei.cmu.edu]

Abstract

The objective of this study is to apply recently developed techniques to infer causality from observational software engineering data. Determining causation rather than just correlation is fundamental to selecting factors that control outcomes such as cost, schedule, and quality. The Tetrad tool's PC and FGES causal search algorithms were applied to software engineering data from 4940 programs written in the C programming language collected during Personal Software Process (PSP) training. PSP programs have previously been used in empirical research quantitative relationships between developer and project factors. Both algorithms successfully identified the expected relationships and did not find contradictory or implausible associations. Many of the available causal inference search algorithms require Gaussian distributional families with linear effects. The linear relationship may be especially important for software engineering research and may require prior knowledge and data transformation. Because software engineering has depended on small-scale, low-power experiments, often using non-representative students, inferring causal relationships would expand the insight available to researchers. Inferring causation from observational software engineering data shows much promise, but is currently limited by researcher understanding of the capability and limits of causal inference, the quality of the underlying data, and the general requirement for linear effects.

Introduction

Despite repeated calls for empirical studies in software engineering (Perry, Porter, & Votta, 2000) and guidelines for their conduct (Kitchenham & Dybå, 2004) it is usually impractical to run controlled software development experiments. Thus, most data in software engineering are observational, presenting challenges to causal inference. Without causation, selection or control of factors will not have the desired effect on outcomes. Understanding causation is fundamental to the forward-looking control of the software development process.

The epistemological problems of inferring causation from observational data are now being overcome (Pearl, Glymour, & Jewell, 2016; Spirtes, 2010) and accepted in research (Fedak et al., 2015). This study aims to apply causal search techniques to a previously-studied software engineering data set to validate the overall approach and gain experience with the capabilities and limitations of these methods.

Tools for Causal Inference

The University of Pittsburgh, Carnegie Mellon University (CMU), and the Pittsburgh Supercomputing Center serve as founding members of the Center for Causal Discovery (CCD). The CCD develops and maintains causal algorithms, software, and tools, including Tetrad. Tetrad enables users not only to search for causal graphs from a dataset, but also to estimate and evaluate parametric models. We applied the PC and FGES algorithms to selected data from PSP training. These two algorithms were chosen to exercise two different search approaches.

The PC algorithm, named after its creators Peter Spirtes and Clark Glymour, sets a Fisher Z-based *p*-value cutoff for conditional independence testing. The FGES algorithm uses a fast greedy approach to search the space of causal Bayesian networks to return the most probable model(s) based on the Bayesian information coefficient (BIC) score (Sanchez-Romero et al., 2018). Both algorithms assume that each node is a linear function of its parents plus a Gaussian noise term.

One advantage of score-based search algorithms over constraint-based search algorithms is that they can obtain quite accurate adjacencies within the causal graph equivalence class. Also, score-based algorithms typically output only directed or undirected edges. Because equivalence class scoring almost always favors one orientation over the other, bi-directed edges are rare. A limitation of score-based search algorithms is that they can be slow and might not scale as well as constraint-based searches.

Generally in our analyses, we don't have full knowledge of how well the assumptions of the various search algorithms are met, so we usually employ more than one algorithm. Also, because they rely on different mathematical mechanisms to construct the output graph (MEC), we favor applying one or more constraint-based searches and one or more score-based searches, and comparing for commonalities in the direct causal relationships that are identified. This provides some protection against the uncertainties about how well the various assumptions are met by the dataset analyzed. Employing two or more search algorithms based on different mathematical approaches for inferring causal structure also allows us to take advantage of their respective strengths (e.g., there is less ambiguity in the direction of causality with score-based searches).

Personal Software Process (PSP) Data

Dataset Summary

The Personal Software Process was developed by Watts Humphrey at the Software Engineering Institute to demonstrate how an individual can apply the process principles underlying the Capability Maturity Model for personal work. The PSP contains coherent frameworks for defining the development process and measurements for process and products. A progressive development process with activity steps, measurements, and a sequence of training assignment exercises is described *in A Discipline for Software Engineering* (Humphrey, 1995).

PSP classes are taught by trained and authorized instructors who submit resulting data to the Software Engineering Institute for use in research. Several versions of the course have been taught over the years; this study uses the 10-assignment course taught primarily through 2006 because it contains a large sample and consists primarily of professional software developers rather than university students. Additional descriptions of the data and prior analyses can be found in Rombach et al. (2008) and Vallespir and Nichols (2012). For this study, because we wanted to reduce the number of potential hidden confounding factors, we selected only programs using the C programming language and

students who completed the entire 10-program sequence. Although using only a subset of the data introduces some risk for bias, lack of hidden confounders is a key assumption for both search algorithms.

Data Attributes

When using PSP for the program exercises, students record the direct (i.e., stopwatch) time engaged in particular activities (e.g., planning, design, design review, code, code review, compile, test, post mortem analysis) along with information on the defects injected and removed in each phase. For the more focused analysis that is the subject of this report, we used only the total effort (sum of all activities), the construction activity effort (design and code), and the total defects for the 494 students using the C programming language and implementing all 10 assignments. The data variables we examined were as follows:

1. Assignment Average Minutes (abbreviated AsgAveMin)—for each of the 10 program assignments, the average of the log-transformed effort required by the 494 students using the C programming language to complete that assignment. AsgAveMin can be thought of as a proxy for the requirements size of each program. (AsgAveMin is defined for 10 assignments.)
2. Student Size Factor (abbreviated StuSizeFactor)—for each of the 494 students, the ratio of total new lines of code written by the student for the 10 program assignments compared to the total new lines of code written for the 10 program assignments averaged across all 494 students. (StuSizeFactor is defined for 494 students.)
3. Student Effort Factor (abbreviated StuEffFactor)—for each of the 494 students, the ratio of the student’s total effort for the 10 program assignments to the overall student average. Thus, StuEffFactor is very similar to StuSizeFactor, but focuses on a student’s total effort rather than the total new lines of code written. (StuEffFactor is defined for 494 students.)
4. Student Defect Arrival Rate (abbreviated StuDAR)—for each of the 494 students, the ratio of defects introduced during program design and code activities (that is, during construction to the construction effort). This factor characterizes a student’s specific tendency to introduce defects while developing software, using “defects per hour” as the unit. Using hours as the unit instead of the more common minutes should not affect causal inference and provides a better scale for the log-transform. (StuDAR is defined for 494 students.)
5. Construction Minutes (abbreviated ConstMin)—for each of the 494 students and 10 program assignments, the effort expended in construction (design and code) activities, measured in minutes. Thus, ConstMin does not include effort expended in planning, reviews, compile, and test. (ConstMin is defined for 494 students and 10 program assignments.)
6. Lines of Code for the product (abbreviated LOC)—for each of the 494 students and 10 program assignments, the sum of added and modified logical lines of code written. Thus, LOC does not include lines of code that were reused without modification. (LOC is defined for 494 students and 10 program assignments.)
7. Total Development Effort (abbreviated MinTot)—for each of the 494 students and 10 program assignments, the student’s total effort expended on the

program assignment, measured in minutes. (MinTot is defined for 494 students and 10 program assignments.)

8. Total Number of Defects Injected and Discovered (abbreviated DefTot)—the number of defects discovered after the step in which they were injected is completed. For example, any design defect discovered during coding would be counted, but not a defect both introduced and discovered in the same design phase. Also, a typo corrected during normal coding would not count, but a typo discovered by a compiler would. We asked students using interactive development environments to disable automated checking so that these defects would be visible and counted in compile. (DefTot is defined for 494 students and 10 program assignments.)

Note that AsgAveMin can be thought of as a proxy for the requirements size because each student must perform the same exact 10 exercises. The exercises vary in difficulty as does the amount of code required to implement those requirements. Thus, rather than using an estimate based on function points as a measure of requirements size, we estimate the relative size of specific exercises from the arithmetic average of the log-transformed effort expended on an exercise across all developers.

By taking the log transformation, we accomplish several purposes. First, we reduce the effect of outliers, helping to make the search algorithms we utilize less sensitive to outliers, and thus the model(s) returned from causal search more stable. Second, transformed distribution is approximately Gaussian, which the theorems for consistent convergence of the chosen algorithms require. Third, the log transformation will later help to linearize factor effects.

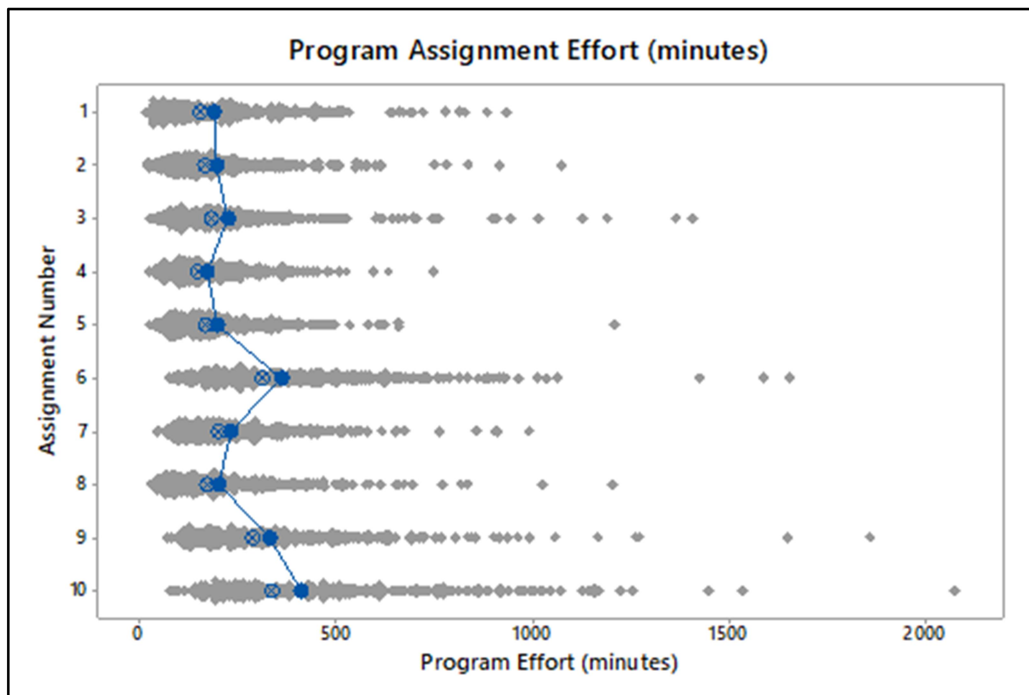


Figure 1. Individual Values Plot of Actual Effort for Each Programming Assignment

Figure 1 shows that individual effort distributes widely for each programming exercise. The bulls-eye symbols indicate the median and the circles (connected by a line)

are the mean. The differences are statistically significant differences between program assignments. The mean values are also shown in a scatterplot of size versus effort in Figure 2 , which also shows that the size varies by nearly a factor of three, while effort varies by roughly a factor of two. We interpret slope as code production rate and the intercept as a start-up cost. We decided to use only the effort factor for this study because for the aggregate, the correlation between size and effort is very strong, suggesting that adding size does not add much information. In this figure, the averages were computed from the log transformed data, then retransformed into the natural units of LOC and minutes.

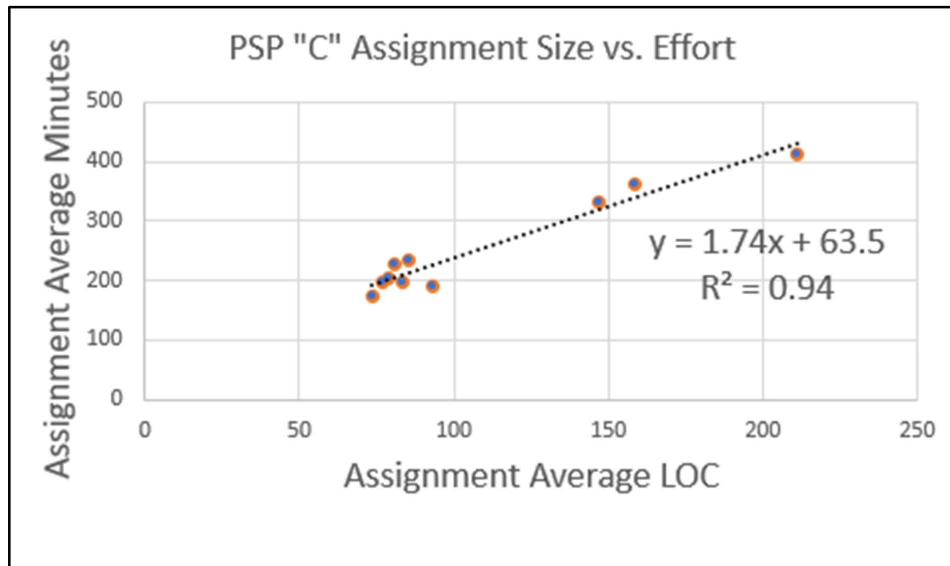


Figure 2. Correlation Between Program Size and Effort for 10 PSP "C" Programming Language Exercises

We also included factors that account for programmer variability. The software engineering literature contains numerous studies reporting variation in programmer productivity (Sackman, Erikson, & Grant, 1968; Curtis, 1981; Valett & McGarry, 1989; DeMarco & Lister, 1999; Card, 1987; DeMarco & Lister, 1985; Sheil, 1981). Few studies, however, explicitly report individual differences in defects or size of solutions to similar problems. Nonetheless, a more recent work (Caliskan et al., 2018) reports that individual programmer characteristics can be identified from the compiled (and even optimized) binary. We decided to explicitly account for programmer idiosyncrasies in coding style, line counting standards, and solution approach with programmer-specific factors that affect product size, defect counts, and production rates. This is supported by a separate ANOVA analysis of the data that finds that such programmer factors are statistically significant and approximately doubles the amount of variance accounted for by the coefficient of determination (from $R^2 \approx 0.3$ to $R^2 > 0.6$). The untransformed distributions of student factors (StuSizeFactor and StuEffFactor) seem to follow lognormal distributions as shown in Figure 3 (note the heavy skew to the right); and the log-transformed data (lnStuSizeFactor and lnStuEffFactor) thus approximately follow normal distributions as shown in Figure 4. Defect arrival rates (StuDAR), untransformed and transformed, are shown in Figure 5 and Figure 6.

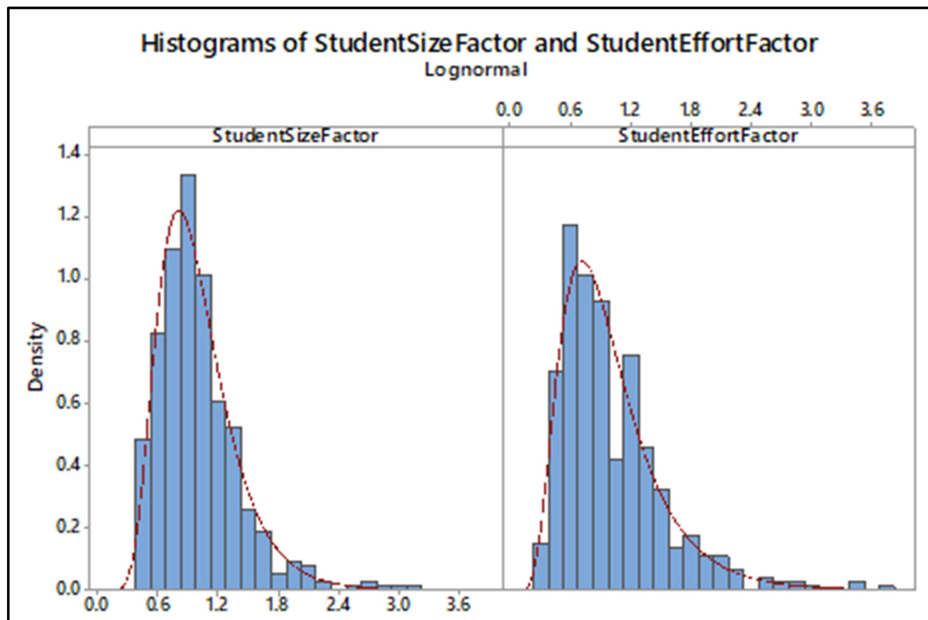


Figure 3. Histogram of Student Size and Effort Factors (494 Points)

The untransformed student-dependent size and effort factors are shown in Figure 3. The log-transformed distributions approximate a Gaussian as shown in Figure 4. Please note that the scales in the paired plots may differ and ordinates are displayed below the left hand plot but above the right hand plot.

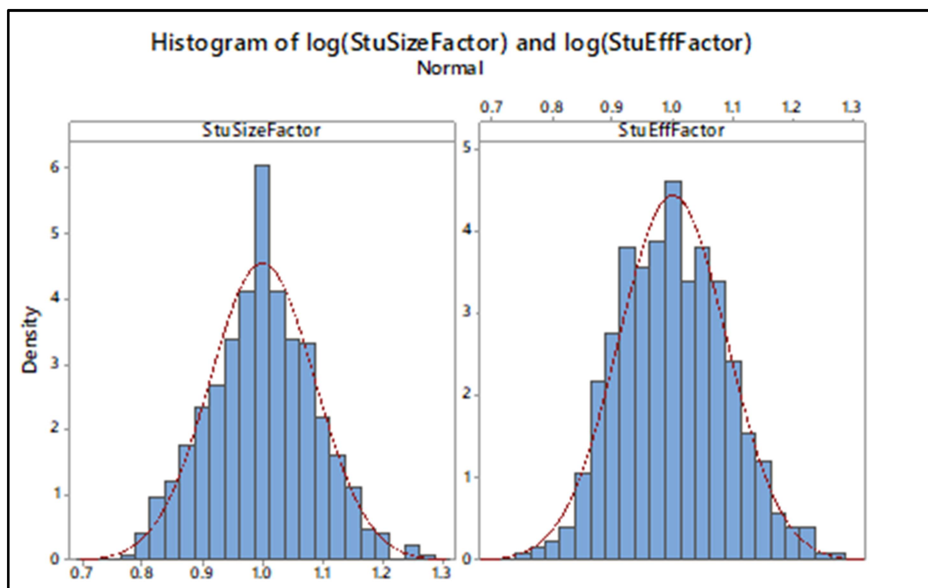


Figure 4. Histograms of Log-Transformed Student Size and Effort Factors

The defect arrival rate is the student's rate of injecting defects during design and coding activities. Because this can be zero, we have added an offset of 1.0 in the transformation to prevent negative rates. This small offset will not affect the search provided that the distribution is approximately Gaussian. The StuDAR at zero are likely an artifact of data gathering practices. We are, however, reluctant to clean the data because of the threat of introducing bias.

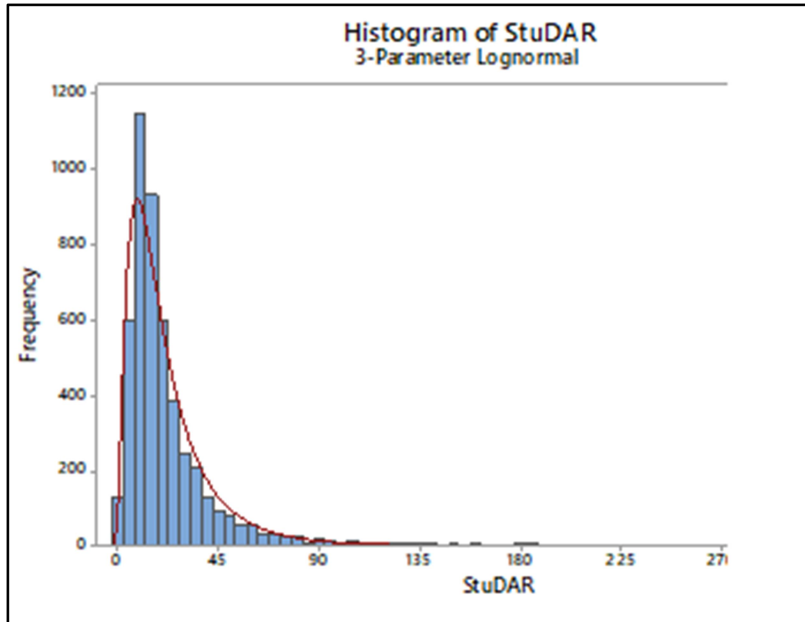


Figure 5. Student Defect Arrival Rate (494 points)

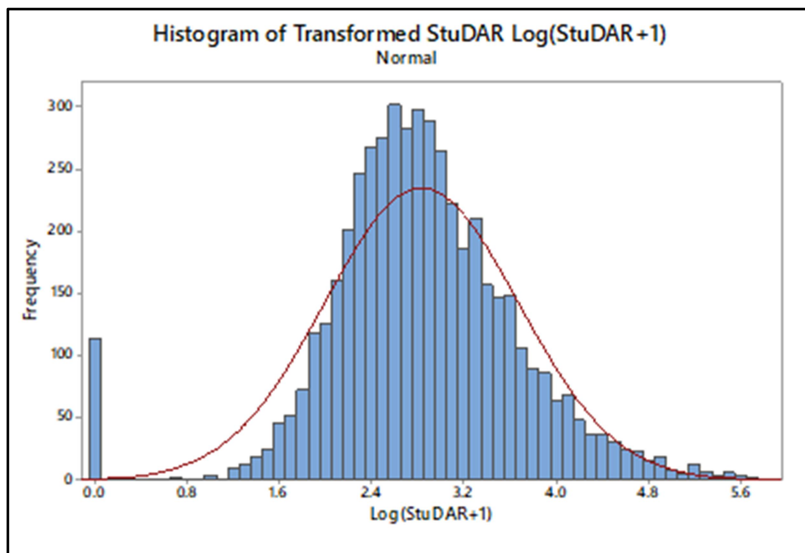


Figure 6. Log of the Student Defect Arrival Rate (494 Points)

For our initial model, we examine the construction time (ConstMin) as a candidate causal contributor to lines of code and defects, and total effort (respectively LOC, DefTot, and MinTot). See Figure 7 for a histogram of ConstMin. Mathematically, construction time is a product of the construction rate times and the product size. Because this rate may already be implicit in other programmer factors, we avoided using it in this analysis. Moreover, that rate may be causal, but uncontrollable. Other factors, including estimation accuracy, effort in review, and review rates, will be considered in future work. For this analysis we use the construction effort because the expected model described in a later section is simple to construct and interpret, thus helping to validate the overall approach.

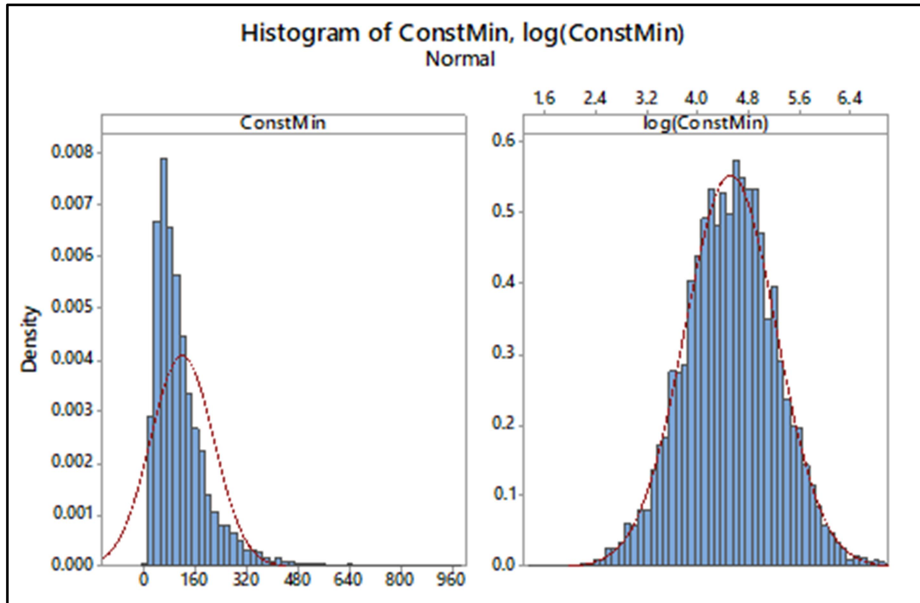


Figure 7. Distribution of Construction Effort and Log Transform of Construction Minutes of Effort (4940 Points)

The outcomes of interest are the program size measured in Lines of Code (LOC; shown in Figure 8), total effort measured in minutes (MinTot; shown in Figure 9) and total defects (DefTot; shown in Figure 10).

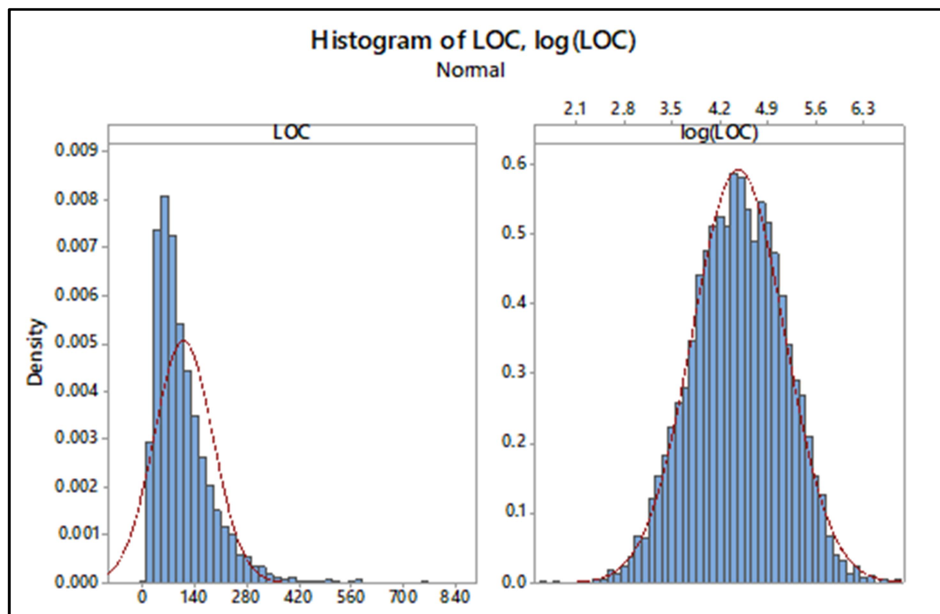


Figure 8. Students' Sum of Added and Modified Logical Lines of Code (LOC) and Log-Transformed LOC for Each Student Exercise Pair (4940 points)

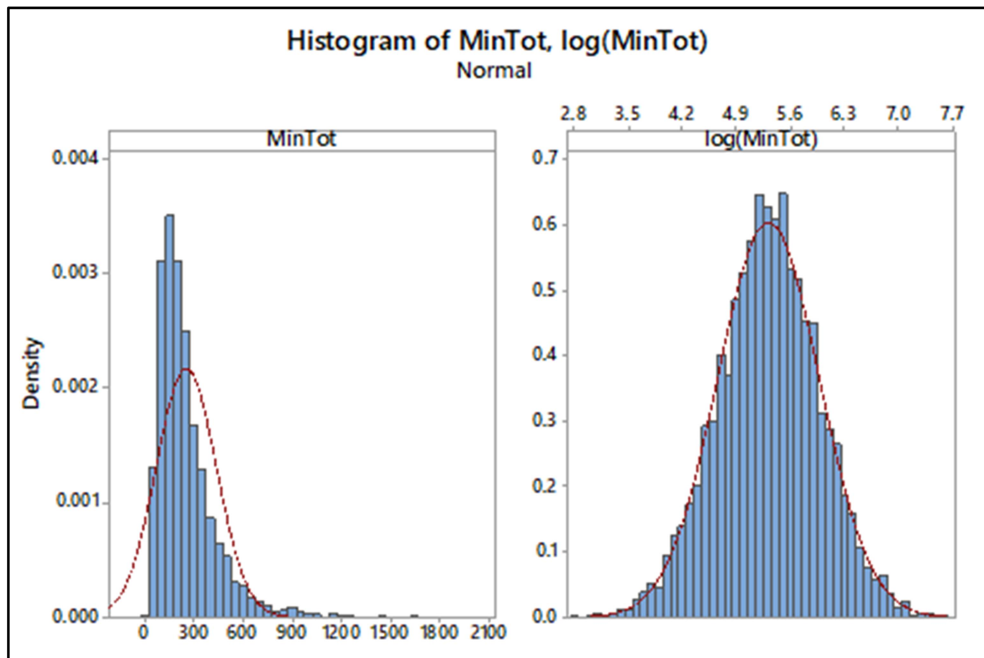


Figure 9. Students' Total Effort in Minutes (MinTot) and Log-Transformed MinTot for Each Student Exercise Pair (4940 Points)

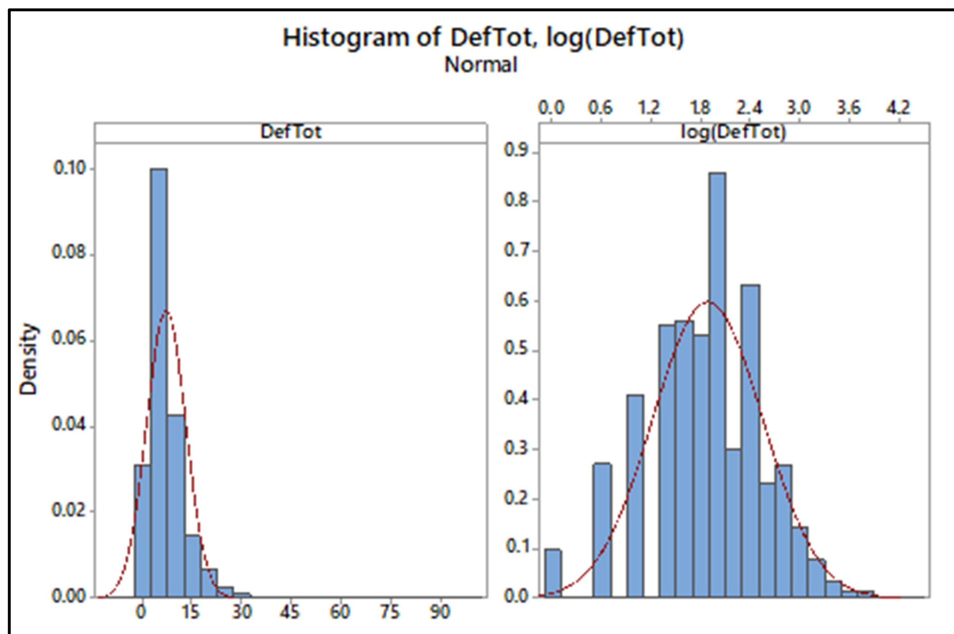


Figure 10. Students' Total Number of Defects Injected and Discovered (DefTot) and Log-Transformed DefTot for Each Student Exercise Pair (4940 Points)

Data Analysis Approach

Research Objective

For this study, we wanted to

- evaluate the effectiveness of applying the causal search algorithms to the PSP software engineering data.
- determine effective data transformations that facilitate correct use of the selected causal search algorithms.

The PSP data is useful for this research because

1. All students develop programs from the same specifications.
2. The measurements framework is required and reinforced by instructors.
3. The data has been analyzed before and is familiar (Rombach et al., 2008; Grazioli, Nichols, & Vallespir, 2014; Vallespir & Nichols, 2011; Vallespir & Nichols, 2012).

Based on prior analysis and experience, we expected some correlating factors to exhibit a causal relationship.

Expected Models

PSP data is used in planning and tracking projects that are run according to the Team Software Process (TSP). In TSP planning, estimates of component size, conversion factors to lines of code, overall production rates, activity effort distributions, historical defect injection rates, and activity defect removal yield are used to predict likely outcomes (Nichols, 2012).

For a PSP programming assignment (i) and student (j) pair, we would expect—without examining any process data—that the program size can be estimated by the untransformed values:

$$LOC_{ij} = ReqSize_i \times SSF_j \tag{1}$$

In the above, we can use the average effort (*AsgAveMin*) as a proxy for *ReqSize*. The actual size will likely vary based on factors such as design versus code effort. This is not yet modeled.

Likewise, we expect the student effort (j) for each program should be related to the assignment size and student dependent factor:

$$MinTot_{ij} = ReqSize_i \times SEF_j \tag{2}$$

Total development time should also be influenced by other factors, including design-specific effort review time and review effectiveness (not included in this model) and the actual defect arrivals. The defect arrivals should be related to the student's (j) defect tendencies, the program size (i), and the actual effort in construction (ij). It is during construction (design and coding) that most defects are injected.

$$DefTot_{ij} = ConstMin_{ij} \times StuDAR_j \tag{3}$$

That these relationships are products is a problem for the search algorithms. We will, therefore, take advantage of the observation that the values from the left hand side of these equations distribute approximately lognormally, by using log transforms. The resulting transformed equations and data thus consists of a linear sum of normally-distributed data, making them more suitable for the search algorithms we intend to employ.

If the causal search is successful in finding consistent causal models involving these four introduced factors for problem requirements size and three developer traits (respectively, $ReqSize_i$, SSF_j, SEF_j, and $StuDAR_j$), we will have added support to the case that these factors can be useful for prediction and mitigation in software development, at least at the individual and team levels for planning and tracking.

Causal Discovery

We ran the PC search algorithm with alpha = .05 (a hyper-parameter defining the *p*-value cutoff for inferring conditional independence) and the domain-knowledge constraints described below.

Tetrad allows users to add constraints regarding the required presence or absence of a particularly-oriented direct causal relationship between two nodes (i.e., parameters). For example, a causal link may be required or forbidden or the direction restricted. A known temporal order can be enforced by placing the nodes into knowledge-box tiers such that causality is only permitted forward, not backward. Adjacencies between nodes appearing within the same tier may be allowed or forbidden. We chose to structure tiers as follows:

- Tier 1: Assignment Average Minutes (AsgAveMin), Student Size Factor (StuSizeFactor), Student Effort Factor (StuEffFactor), and Student Defect Arrival Rate (StudDAR)
- Tier 2: Construction Minutes (ConstMin)
- Tier 3: Lines of Code (LOC), Total Effort (MinTot), and Total Number of Defects Injected and Discovered (DefTot)

Essentially the tiers correspond to pre-development inputs (characteristics of the problem or developers), in-process data (the construction effort in minutes), and process outputs (size, effort, and total defects). The default setting is that nodes in the same tier can have direct causal relationships between them, but a node in a lower tier (assigned a higher number) cannot have an oriented edge pointing to a node in a higher tier (assigned a lower number).

Results

Node Links

The search results are shown in Figure 11 and Figure 12. There are some differences but no explicit contradictions.

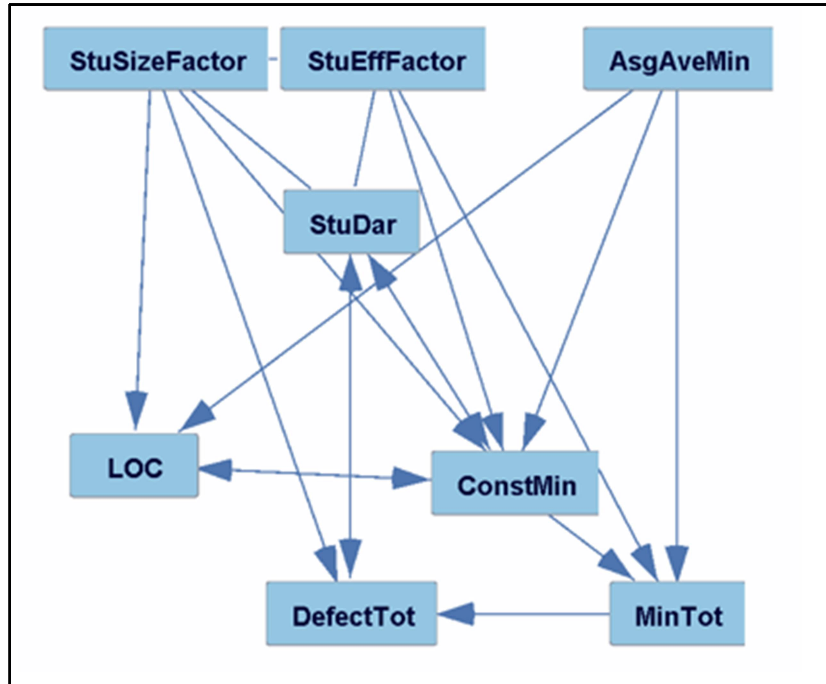


Figure 11. Resulting DAG From Tetrads PC Search Algorithm for Data From Programs Written in C

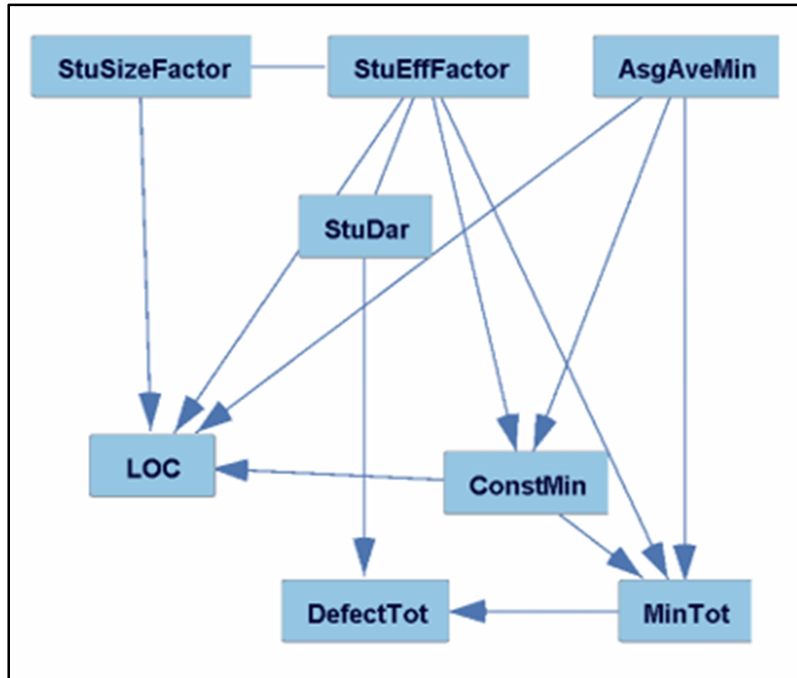


Figure 12. Resulting DAG From Tetrads FGES Search Algorithm for Data From Programs Written in C

The following direct causal edges occur in both graphs:

1. AsgAveMin --> ConstMin
2. AsgAveMin --> LOC
3. AsgAveMin --> MinTot
4. ConstMin --> MinTot
5. MinTot --> DefectTot
6. StuEffFactor --> ConstMin
7. StuEffFactor --> MinTot
8. StuSizeFactor --> LOC

The following undirected causal edge occurs in both graphs:

1. StuEffFactor --- StuDAR (This means there's evidence of a direct causal relationship between the two nodes, but there's insufficient information to determine the direction.)

The following directed causal edges occur in only one graph:

1. ConstMin --> LOC (The graphs returned by PC shows as a bidirected edge, meaning there's evidence for a hidden confounder of the two nodes.)
2. StuDAR --> DefectTot (The graphs returned by PC shows as a bidirected edge, meaning there's evidence for a hidden confounder of the two nodes.)
3. StuEffFactor --> LOC
4. StuSizeFactor --> ConstMin
5. StuSizeFactor --> DefectTot

The following undirected causal edge occurs in only one graph:

1. StuSizeFactor --- StuDAR

Threats to Validity

Internal Validity

That the PSP data collection forms filled out and collected during PSP training were consistent and completely filled in did not necessarily mean that students recorded complete and accurate data. Others (Johnson et al., 2003) found significant errors, though these seem primarily to be in hand calculations prior to automated tool support. Our analysis relies only upon direct measures for which all further calculations are performed by the data-gathering spreadsheet used during training. Nonetheless, data correctness validation relies upon the diligence of the student and instructor.

The difficulty of the exercises or rigor required by the course may lead to a survival bias. Excepting the LOC counter, which is the very first exercise, the series of exercises consists of writing programs that must perform statistical and floating-point math calculations. Some are more complicated and challenging to program than others, and many students failed to complete the course, perhaps in part due to challenges they encountered while working on these problems. Our analysis uses only data from students who completed the course.

There may also be a bias in the experience of the students regarding domain experience with statistics or programming experience in general. The data used for the study includes 494 software practitioners who took part in the training at the Software Engineering Institute (SEI) or at external locations. The data provided limited information that could help us measure the programmers' experience. We have indications of years of programming and the number of lines of code in the target language, but no information about domain experience. It is likely that more domain-experienced programmers (i.e., those with strong elementary knowledge of statistics and linear algebra) performed better.

There is some risk of maturation bias. The PSP course deliberately teaches estimation, design techniques, and review of both code and designs, while the developers may gain experience within the numeric programming domain. The maturation effect on total defects and defects escaping into test is evident. Although total code production rate appears to be unchanged, the overall rate does initially slow, then return to near its original level. The risks with respect to inference on the effects of PSP training were addressed in Rombach et al. (2008) and Hayes and Over (1997); however, there could also be process-drift effects.

External Validity

We report only on results from the C programming language. By selecting only data from students using C, we have mitigated issues arising from the use of different programming languages between subjects (e.g., in determining StuSizeFactor) at the expense of some generality. Different languages can be more or less suited to solving certain programming problems, and this could affect the assignment normalization factors. Future analyses will compare results with other programming languages.

Because the work was performed in class settings rather than under normal industrial conditions, there is a risk that the results might not generalize beyond the academic setting.

Construct Validity

The program assignment requirements were deliberately left vague on some key points, mostly related to the specifics of input and output formats, error checking, and so forth. Individual interpretations of the requirements could vary somewhat, adding some variation to implemented program size and effort.

Our proxy for an independent measure of program size, the average effort in minutes (AsgAveMin), aggregated these individual programmer choices and thus might be subject to some systematic bias.

Discussion

All the identified causal relationships have face plausibility. Larger assignments cause more effort and larger amounts of code. The expected relationships for size and effort both appear reasonable. The relationship with defects is less clear. The FGES search algorithm finds the StuDAR and total effort (MinTot) rather than construction effort causing defects, while PC-Stable has a bidirected edge (double arrow) between DefectTot and StuDAR (indicating a hidden confounder). PC also has a connection from StuSizeFactor to DefectTot.

It is possible that the algorithms lead to slightly different models because the data is from mixed-causal systems or is insufficient. Another possibility is that there is overlapping information in these specific variable constructs (creating dependencies with hidden variables). To avoid this problem it is often preferable to use only direct rather than derived values; however, that approach runs into the problem that both PC and FGES assume linear relationships between a child node and its parents, plus a Gaussian noise term. Additional work will be needed to find better variable selections with minimal variable overlap, paired with the most appropriately-selected causal search algorithms. In particular, there are search algorithms that were designed to search for non-linear causal relationships among variables having skewed noise distributions; and there are still variants, as well, that endeavor to take into account hidden confounders.

We offer several observations of lessons learned during this exploratory work.

First, it is imperative to visualize the data. We are not yet certain of the sensitivity to deviations from Gaussian, but single peaks and lack of outliers are surely important. Moreover, the distribution characteristics affect the available transforms. We have focused on simple relationships, but these will not get us from planning through production. Much work remains to model more complex and stepwise systems.

Second, mechanistic relationships with direct measures can become complicated. Using derived measures risks including the same factor multiple times and can lead to mathematical artifacts because of ratios. The requirement for linear effects constrains the available choices in ways that are not immediately obvious. Moreover, it is not yet clear that the natural mechanistic models can be successfully transformed for analysis by the algorithms currently available.

Third, while transformations can sometimes simplify the problem, they make the data relationships less intuitive.

Forth, count data, such as defects, can be poorly behaved with low numbers. We counter this first by counting all defects, not just test, and by using an offset of “+1.” Nonetheless, the distributions can become noticeably discrete on the left-hand side of the peak.

Fifth, the data quality and consistency are a concern. Long lists of problems with software engineering data are available (Shull, Singer, & Sjøberg, 2008). The analyst must be keenly aware of the strengths and weaknesses of particular datasets.

Ultimately, the causal mechanisms we expected to see do appear in the resulting models and implausible causal mechanisms do not. Which specific models are better predictors of future performance is left for future work.

Conclusion

The PC and FGES search algorithms returned results that are generally consistent with each other and with overall expectations.

Causal inference methods should be applied in software engineering, but with caution. We have made only initial steps toward assessing the degree to which different search algorithms are sensitive to deviations from the assumptions about shape (Gaussian for some, skewed for others), outliers, linear effects, or homoscedasticity. Real datasets are likely to be subject to problems of construct validity, measurement inconsistency, determinism, and process drift. Guidelines on reporting data characteristics and the sensitivity of different algorithms will be included in future work.

References

Caliskan, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R., & Narayanan, A. (2018). When coding style survives compilation: De-anonymizing programmers from executable binaries. *Network and Distributed Systems Security (NDSS) Symposium 2018.* doi:10.14722/ndss.2018.23304

Card, D. N. (1987). A software technology evaluation program*. Information and Software Technology, 29*(6), 291–300. doi:10.1016/0950-5849(87)90028-0

Curtis, B. (1981). Substantiating programmer variability. *Proceedings of the IEEE, 69*(7), 846.

DeMarco, T., & Lister, T. (1985). Programmer performance and the effects of the workplace. *Proceedings of the 8th International Conference on Software* (pp. 268–272).

DeMarco, T., & Lister, T. (1999). *Peopleware: Productive projects and teams*. New York, NY: Dorset House.

Fedak, K. M., Bernal, A., Capshaw, Z. A., & Gross, S. (2015). Applying the Bradford Hill criteria in the 21st century: How data integration has changed causal inference in molecular epidemiology. *Emerging Themes in Epidemiology*, *12*. doi:10.1186/s12982-015-0037-4

Grazioli, F., Nichols, W., & Vallespir, D. (2014, January). An analysis of student performance during the introduciton of the PSP: An empirical cross-course comparison. In *TSP Symposium 2013 Proceedings* (CMU/SEI-2013-SR-022; pp. 11–21). Retrieved from http://repository.cmu.edu/cgi/viewcontent.cgi?article=1772&context=sei

Hayes, W., & Over, J. W. (1997, December). *The Personal Software Process (PSP): An empirical study of the impact of PSP on individual engineers* (Report No. CMU/SEI-97-TR-001). Retrieved from https://resources.sei.cmu.edu/asset_files/TechnicalReport/1997_005_001_16565.pdf

Humphrey, W. S. (1995). *A discipline for software engineering*. Boston, MA: Addison-Wesley Longman Publishing.

Johnson, P. M., Agustin, J., Chan, C., Moore, C., Miglani, J., & Doane, W. E. J. (2003). Beyond the Personal Software Process: Metrics collection and analysis for the

differently disciplined. In *Proceedings of the 25th International Conference on Software Engineering* (pp. 641–646). doi:10.1109/ICSE.2003.1201249

Kitchenham, B. A., & Dybå, T. (2004). Evidence-based software engineering. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04).* Edinburgh, Scotland. doi:10.1109/MS.2005.6

Nichols, W. R. (2012). Plan for success, model the cost of quality. *Software Quality Professional, 14*(2), 4–11.

Pearl, J., Glymour, M., & Jewell, N. P. (2016). *Causal inference in statistics : A primer.* Hoboken, NJ: Wiley.

Perry, D. E., Porter, A. A., & Votta, L. G. (2000). Empirical studies of software engineering: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 345–355). doi:10.1145/336512.336586

Rombach, D., Münch, J., Ocampo, A., Humphrey, W. S., & Burton, D. (2008). Teaching disciplined software development. *Journal of Systems and Software, 81*(5), 747–763. doi:10.1016/j.jss.2007.06.004

Sackman, H., Erikson, W. J., & Grant, E. E. (1968). Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM, 11*(1), 3–11.

Sanchez-Romero, R., Ramsey, J. D., Zhang, K., Glymour, M. R. K., Huang, B., & Glymour, C. (2018). Causal discovery of feedback networks with functional magnetic resonance imaging. *Preprint*, 1–54.

Sheil, B. A. (1981). The psychological study of programming. *Computing Surveys, 13*(1).

Shull, F., Singer, J., & Sjøberg, D. I. K. (Eds.). (2008). *Guide to advanced empirical software engineering.* London, England: Springer. doi:10.1007/978-1-84800-044-5

Spirtes, P. (2010). Introduction to causal inference. *Journal of Machine Learning Research, 11,* 1643–1662. Retrieved from https://dl.acm.org/citation.cfm?id=1859905

Valett, J., & McGarry, F. (1989). A summary of software measurement experiences in the software engineering laboratory. *Journal of Systems and Software, 148*(2), 137–148. Retrieved from http://www.sciencedirect.com/science/article/pii/0164121289900162

Vallespir, D., & Nichols, W. (2011). Analysis of design defect injection and removal in PSP. In *TSP Symposium 2011 Proceedings* (pp. 19–25). Retreived from https://www.fing.edu.uy/sites/default/files/biblio/22573/designdefectspsp.pdf

Vallespir, D., & Nichols, W. (2012). An analysis of code defect injection and removal in PSP. In *TSP Symposium 2012 Proceedings* (CMU/SEI-2012-SR-015; pp. 3–19). Retrieved from https://resources.sei.cmu.edu/asset_files/SpecialReport/2012_003_001_34121.pdf

Acknowledgments

This material is based upon work supported in part by Cyber Security and Information Systems Information Analysis Center (CSIAC). We would also like to thank David Zubrow and Robert Stoddard of the SEI for encouragement, support, and sharing insights for the work in this paper. Additionally, we thank David Danks, Kun Zhang, Madelyn Glymour, and Joe Ramsey for their help in understanding causal discovery, the algorithms, and the tools.

Disclaimer and Distribution Statement

Copyright 2018 Carnegie Mellon University. All Rights Reserved.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

No warranty. This Carnegie Mellon University and Software Engineering Institute material is furnished on an “as-is” basis. Carnegie Mellon University makes no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material. Carnegie Mellon University does not make any warranty of any kind with respect to freedom from patent, trademark, or copyright infringement.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

Internal use:* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use:* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

* These restrictions do not apply to U.S. government entities.

Personal Software Process^{SM}, PSP^{SM} and TSP^{SM} are service marks of Carnegie Mellon University.

DM18-0425

ACQUISITION RESEARCH PROGRAM
GRADUATE SCHOOL OF BUSINESS & PUBLIC POLICY
NAVAL POSTGRADUATE SCHOOL
555 DYER ROAD, INGERSOLL HALL
MONTEREY, CA 93943

www.acquisitionresearch.net