Position Paper: Bayesian Reasoning for Software Testing

Akbar Siami Namin
Advanced Empirical Software Testing and Analysis Research Group
Department of Computer Science
Texas Tech University
Lubbock, TX, USA
akbar.namin@ttu.edu

Mohan Sridharan
Stochastic Estimation and Autonomous Robotics Laboratory
Department of Computer Science
Texas Tech University
Lubbock, TX, USA
mohan.sridharan@ttu.edu

ABSTRACT

Despite significant advances in software testing research, the ability to produce reliable software products for a variety of critical applications remains an open problem. The key challenge has been the fact that each program or software product is unique, and existing methods are predominantly not capable of adapting to the observations made during program analysis. This paper makes the following claim: *Bayesian reasoning methods provide an ideal research paradigm for achieving reliable and efficient software testing and program analysis.* A brief overview of some popular Bayesian reasoning methods is provided, along with a justification of why they are applicable to software testing. Furthermore, some practical challenges to the widespread use of Bayesian methods are discussed, along with possible solutions to these challenges.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: [Testing and Debugging, Diagnostics, Monitors, Tracing]

General Terms

Reliability, Experimentation, Measurement

Keywords

Bayesian data analysis, probabilistic reasoning, stochastic methods, software testing, program analysis

1. MOTIVATION

The software industry is a multi-billion dollar contributor to the economic growth and technological advancement of modern society [15]. Bohem and Sullivan attributed the importance of software economics to three reasons: the alteration of the dynamics of technology innovation; the increased impact of software-enabled change in organizations; and the identification of value creation as the key to success [3]. The quality and reliability of software products are hence of paramount importance, and thus software testing is a key component of the software development life-cycle.

Despite significant advances in testing methods, the increased use of software has only increased the cost of debugging defective software. Harrold even stated that the use of systematic testing techniques is not widespread in industry [18]. Existing software testing methods are still unable to provide high-quality software products.

Software testing research has frequently utilized methods from other computer science disciplines such as compilers, programming languages, logical reasoning, data mining and graph theory. However, many software testing challenges are essentially NP-hard problems. Hence, a critical concern is whether the solutions proposed for software testing challenges are appropriate and sufficient. In his paper, Notkin [26] states: "*we may need to approach testing and analysis more like theoreticians pursue NP-hard problems: in the absence of efficient, precise algorithms, the theoreticians pursue probabilistic and epsilon-approximate algorithms.*" In other words, approximation algorithms [20] may constitute a more appropriate approach for software testing problems.

Software programs are typically developed for specific tasks, and they possess certain unique features (e.g., mathematical computation) that result in unique behaviors during testing. Though these unique features define the complexity of the program, there is little agreement on a comprehensive list of features that characterize all programs. The unique behaviors introduce uncertainties during program analysis, and the existing software testing methods are incapable of adapting to these behaviors due to the following reasons:

- **Specificity.** Software testing methods are often developed to encode the observations obtained through case studies or experiments. Though these methods may perform well for certain subject programs, they may not provide good performance when used for other programs.
- **Intractability.** Effective test cases need to be developed to create reliable software products. Generating a large number of test cases for any reasonable-sized program and evaluating the thoroughness of these test cases is intractable. Furthermore, maintaining these test cases in the context of regression testing and software changes is a challenge.
- **Inability to Adapt.** Current testing methods typically do not *adapt* based on the data observed while analyzing a particular program. As a result, the debugging data is not fully utilized and these methods are unable to account for the uncertainties associated with each program.

Though the above-mentioned problems are posed in the context of software testing, they are also observed in many other research areas in computer science and engineering. Data mining methods, for instance, automatically recognize complex patterns in a noisy data stream, and use these patterns to make adaptive intelligent decisions. The pattern recognition methods developed in data mining research can therefore be utilized to automate some software testing problems [22]. However, not all such methods can address the challenging issues (e.g., specificity, intractability) listed above.

It has been stated that the field of software engineering is a *fertile ground* and that many software engineering problems can be formulated as learning problems that can be addressed using popular machine learning algorithms [35]. In a recent keynote address, Briand [5] stated that machine learning techniques are diverse and that their utilization depends on the underlying assumptions and the preparation of the research community. Machine learning methods can be grouped into two high-level categories. The *offline learning* methods train models based on observed data—the models are then used to identify desired patterns in the test data. Examples include: decision-trees, association rules, support vector machines etc. The *online learning* methods, on the other hand, include adaptive algorithms that learn models and incrementally refine the learned models based on the observed data. One popular means of achieving this adaptive performance is to use a probabilistic representation and a Bayesian reasoning scheme. Examples include: Bayesian classification, Markov decision processes, Bayesian networks, stochastic sampling etc. These probabilistic algorithms have been used extensively in research areas such as computer vision, robotics and human-computer interaction [2, 32].

Bayesian methods have been used to address a few research issues in software engineering [27, 30]. Research in software reliability, for instance, has several examples of the use of Bayesian methods to address research challenges. Examples include the: use of active learning for predicting software behavior [4]; use of Bayesian networks for predicting software maintainability [33]; and the use of Bayesian networks to predict software defects in development life cycles [13, 14]. However, despite the inherent potential of Bayesian methods to adapt to the observed behavior during program analysis, the use of Bayesian reasoning in software testing and program analysis is still in its early stages.

It has been stated that software testing is among the most challenging domains for structured machine learning over the next ten years [11]. Machine learning algorithms, especially the Bayesian reasoning methods, are appropriate for adapting to the behavior of the target program based on the observed (debugging) data. However, though learning algorithms and probabilistic representations have been used in software testing research, the online and incremental adaptation capabilities of Bayesian methods have not been fully utilized. This paper advocates the use of Bayesian methods for achieving adaptive software testing, primarily because these methods can model the inherent stochasticity of software testing and other real-world systems.

2. BAYESIAN REASONING AND APPLICABILITY TO SOFTWARE TESTING

Exact inference is often intractable in complex real-world domains. Bayesian reasoning methods provide a mathematically well-defined mechanism for representation, inference and learning in such situations. They use a probabilistic representation to explicitly model the uncertainty and hence track multiple hypotheses about the state of the system being analyzed. A higher probability represents a higher likelihood that the corresponding hypothesis is true. As additional information is obtained about the state of the system, in the form of a series of noisy observations, Bayesian methods provide an elegant scheme to incrementally update the probabilities associated with the individual hypotheses [2]. As a result, Bayesian inference methods have been used extensively in several real-world domains. This section briefly overviews some Bayesian inference methods and their applicability to software testing challenges. The underlying principle of Bayesian inference is [2]:

$$p(a|b) = \frac{p(b|a)p(a)}{p(b)} = \frac{\text{likelihood} \cdot \text{prior}}{\text{normalizer}} \qquad (1)$$

Equation 1 is the basic form of Bayes rule that computes the *posterior* (conditional) probability of event a given b, i.e., $p(a|b)$, based on the *likelihood* $p(b|a)$, prior probability $p(a)$, and probability $p(b)$ that is a normalizing constant. This simple rule can be used to design many different methods.

$$p(C_i|z) = \frac{p(z|C_i) \cdot p(C_i)}{\sum_{j=1}^{N} p(z|C_j)p(C_j)} \qquad (2)$$

Equation 2 shows the version of Bayes rule for multi-class classification with classes $C_i, i \in [1, N]$, where N is the number of classes. It incrementally updates the probability of class C_i given observation z, i.e., $p(C_i|z)$, based on the prior likelihood of obtaining this observation given class C_i, i.e., $p(z|C_i)$, and the prior probability of this class: $p(C_i)$. Consider the example of detecting the presence or absence of a fault in a specific code segment, denoted by classes C_1 and C_2 respectively. The debugging data from individual test runs provides observations (z). A model of the likelihood of observations given the presence or absence of faults in a code segment, i.e., $p(z|C_1)$ and $p(z|C_2)$, can be modeled based on known program behavior. Then, based on the observations, the posterior probability of the presence or absence of a fault can be incrementally updated. This formulation is robust to a few unreliable observations, and can be used for several different classification and regression tasks in software testing or program analysis.

A key principle in Bayesian reasoning is the *Markov* assumption. Consider a system that changes over time, and is observed through a series of observations z obtained through a set of actions u: $\{u_1, z_1, \ldots, u_t, z_t\}$. The goal is to estimate the probabilistic *belief* of system state x at time t: $bel(x_t) = p(x_t|x_0, u_1, z_1, \ldots, x_{t-1}, u_t, z_t)$. The first-order *Markov* assumption for system state can then be stated as:

$$p(x_t|x_0, u_1, z_1, \ldots, x_{t-1}, u_t, z_t) = p(x_t|x_{t-1}, z_t, u_t) \qquad (3)$$

i.e., given x_{t-1} the system state at time $t-1$, the current action and the current observation, the state at time t can be estimated conditionally independent of all prior states, actions and observations. This assumption is applicable to some software testing problems as well when the domain is stationary—the presence or absence of a fault does not change during program analysis. In addition, each test run does provide an independent observation. Furthermore, other Bayesian methods can be utilized to formulate dynamic domains where the state of the system can change over time.

The Markov assumption can be used to derive a two-step iterative process that forms the core of Bayesian inference methods. First, a *prediction* step updates the belief of all possible hypotheses of system state based on the prior belief and the actions taken since then. Second, a *correction* step "corrects" the updated belief based on the correspondence between the expected and actual observations. The key feature is the incremental and iterative update of the likelihood of the hypotheses. Such an adaptive procedure is well-suited to the software testing domain. Consider the problem of determining the location of faults in a program. Here, the first step will predict the likely locations based on prior knowledge (or observations), while the second step will revise the fault location probabilities based on the observations obtained during the current test run. Popular state estimation techniques such as Kalman filters and particle filters (i.e., MonteCarlo sampling) are based on this inference scheme—see [2, 32] for complete details.

The *MonteCarlo* sampling methods have been used successfully in domains with significant noise in the observations and actions [2, 32]. Sampling is applicable to domains with multiple hypotheses about the state of the system being analyzed. Each "sample" is an instance of a hypothesis (e.g., the occurrence of a type of fault) and it is associated with a probability that represents the likelihood that the hypothesis is true. A small set of samples of the hypotheses are selected initially, and each iteration of sampling consists of three steps: (1) each hypothesis is modified to account for any dynamic changes in the system; (2) the probability of each hypothesis is updated based on examining some samples of that hypothesis; and (3) a larger number of samples are drawn corresponding to hypotheses with a larger (relative) probability. Over a few iterations, sampling converges to focus on hypotheses that show more evidence of being correct.

Sampling methods are hence well-suited to model the inherent stochasticity of software testing. Consider, for instance, the mutation testing problem, where the adequacy of a test suite is measured in terms of its ability to expose faulty versions of the target program (i.e., mutants) that are synthetically generated using well-defined mathematical transformations (i.e., mutation operators). The generated mutants will differ based on the target program and the mutation operators, and analyzing all mutants of a reasonable-sized program is intractable. One intuitively appealing approach is to determine the mutation operators whose mutants are more likely to remain unexposed by the existing test suite, so that the test suite can be suitably augmented. The authors have developed an approach based on importance sampling and information theory that significantly improves the mutation testing performance in comparison to the existing approaches [28]. A similar sampling-based approach is also applicable to related challenges such as test case generation, test case minimization, adaptive random testing and static analysis. For instance, a sampling-based selection of appropriate test cases would result in reliable program analysis. Similarly, sampling-based prioritization of the results obtained during static analysis would provide significantly better results than the existing probabilistic approaches for this problem [19].

It is frequently necessary to plan a sequence of actions to perform a given task reliably and efficiently. This planning is particularly challenging when the action outcomes are not reliable. In addition, the state may not be directly observable, and can only be estimated based on action outcomes. Planning in real-world domains characterized by non-determinism and partial observability can be elegantly modeled as Markov decision processes (MDPs) or partially observable MDPs (POMDPs) [2]. MDPs and POMDPs are common formulations used in probabilistic, i.e., decision-theoretic planning methods. These planning methods are instances of Bayesian reasoning that have been used successfully in many application domains. Another related problem is to use a sequence of observations to estimate the most likely sequence of states that generated the observations. Methods such as hidden Markov models can be used to formulate such problems [2, 21]. The key feature, once again, is the ability to explicitly model the inherent uncertainty in order to provide a sequence of actions most likely to achieve a desired goal, or to estimate the sequence of states that is most likely to have produced the observations.

Consider fault localization, a software testing challenge where the objective is to identify faulty statements in the program being analyzed. However, the result of evaluating individual test cases on the program can be noisy, i.e., non-deterministic. There have been some instances of the application of machine learning methods to fault localization [6, 7]. In the last few years, fault localization has been addressed using Bayesian reasoning techniques such as dependency graphs, Bayesian networks, universal models and causal inferences [1, 8, 12, 24]. These graphical methods have been successful because they have explicitly modeled this uncertainty in program analysis [1, 12]. The performance can be further improved by developing a fully Bayesian treatment of such challenges, where the required (probabilistic) models are learned and updated online. Furthermore, stochastic sampling methods such as importance sampling and Gibbs samplings can be used to effectively model the uncertainties associated with the cause of failure during fault localization.

A probabilistic representation can also be used in the software reliability domain to represent and measure the reliability of program components. Factors such as defect density, time to failure and growth model have already been modeled using Bayesian networks [4, 33, 13]. These approaches can be enhanced significantly using methods such as Markov decision processes and (Bayesian) regression. Software economics and metrics research also provides an application domain for Bayesian analysis—software cost models can, for instance, be learned using Bayesian reasoning methods [9].

Test case generation is another major software testing challenge that can be formulated using Bayesian reasoning methods. Efficiency and effectiveness are the main concerns in this domain. Prior research has already resulted in approaches that prioritize test cases in the course of regression testing [25], and generating test strategies based on Bayesian networks [17, 34]. Adaptive random test case generation is a feasible approach for generating a large number of test cases. The effectiveness and thoroughness of these randomly generated test cases can be significantly enhanced by using Bayesian sampling and probabilistic reasoning to generate test cases that are more useful, i.e., more likely to identify faults in the target program.

In many real-world tasks, noisy observations are obtained over several trials, and an approach is required to use these observations to learn a reliable model of the system being analyzed. In other instances, it may be necessary to estimate the best possible action to take in any given state. The *reinforcement learning* literature provides several methods for learning such models and estimating the "value" of taking each possible action in each possible state [31]. The problem is essentially modeled as an MDP, and several solution techniques have been proposed and used extensively to tackle research problems in industry and medicine.

In addition to the core Bayesian reasoning methods, principles drawn from the associated fields such as information theory can also be applied to software testing challenges. Consider the example of *entropy*, a key concept in information theory [10]. Entropy is a measure of the uncertainty associated with a random variable. A large value of entropy indicates a higher uncertainty, while a small value implies a greater confidence in the estimated value of the random variable. One possible use of entropy is to measure whether successive iterations in a stochastic sampling process are providing useful information. For instance, the authors used entropy to automatically terminate the importance sampling process while estimating the mutation operators whose application to the target program is more likely to generate mutants that will remain undetected by the corresponding test suites [28].

Only a small representative set of Bayesian methods have been summarized above—see [2] for a detailed description of a wide range of Bayesian methods and their applicability to several machine learning problems. However, the above-mentioned description does establish the applicability of Bayesian methods to software testing challenges. Recent papers in the software testing literature have shown that such Bayesian formulations lead to significantly better performance than the existing methods. However, the utilization of Bayesian methods as a research paradigm for formulating software testing challenges needs further investigation. The goal of this paper is to stimulate interest in the software testing community, and to promote the use of Bayesian methods for formulating challenges in the field of adaptive program analysis.

3. PRACTICAL CHALLENGES

Bayesian reasoning methods have the capability to transform the field of software testing, by offering probabilistic solutions to major challenges in the field. However, some practical challenges need to be overcome in order to enable the widespread use Bayesian methods.

Generalization Issues. One criticism of Bayesian reasoning (and even empirical) approaches is the generalizability of these approaches. Logical reasoning plays an important role in many areas of computer science, and there are well-established ways to move from specific observations to broader generalizations using inductive reasoning. However, it is extremely challenging to generalize the results of empirical or probabilistic analysis. Research initiatives based on empirical work typically discuss the *threats to validity*, especially the *external threats* to the proposed method. It is also common to question whether the results obtained would generalize to other programs or programming languages. Such an argument can also be made against empirical work based on Bayesian reasoning methods. However, many researchers may not realize that the underlying probabilistic representation makes Bayesian methods more robust to such threats to validity. Furthermore, Bayesian methods provide the highly desirable ability to incrementally and automatically tune the performance based on the observations (i.e., debugging data) obtained during the analysis of the target program.

Sensitivity to Priors. Another criticism of Bayesian approaches is that their performance is highly dependent on the choice of prior probabilities. In addition, it is a challenge to effectively estimate the conditional likelihoods and joint probability densities required for the Bayesian update. However, it can be argued that the same prior knowledge that is encoded to design the existing techniques, can be better utilized by the Bayesian methods. The underlying probabilistic representation provides an elegant scheme to represent prior information. In addition, the incremental update scheme ensures a significant amount of robustness to unreliable observations (e.g., spurious recognition of faults), a feature that the existing non-probabilistic techniques lack. For instance, the authors recently developed a method based on importance sampling for mutation testing [28]. The corresponding experimental results showed that the performance of the stochastic sampling technique was mostly independent of the choice of the prior probabilities. One way to address this prevalent criticism is to perform extensive experimental analysis on different subject programs, and to state the possible threats to validity along with all reported results. At the same time, the research community needs to recognize the contributions made by Bayesian reasoning to several real-world applications.

Steep Learning Curve. Ghezzi and Mandrioli [16] emphasized that all engineers, including software engineers, must have a solid background in statistics and probability theory. In reality, however, the time-consuming nature of statistical analysis deters researchers from investing the time required to master even the basic concepts such as *p-values* and hypothesis testing [23]. As a result, contradictory research findings are frequently published. In addition, there is this perception that "statistics can be used to prove anything". The situation is all the more formidable with regard to Bayesian reasoning methods. Unlike other applied research fields (e.g., machine learning and computer vision), not all software engineering researchers are likely to possess the required background in Bayesian reasoning. The core concepts of Bayesian and probabilistic reasoning are typically not covered in a standard software engineering course curriculum. As a result, there is a significant "inertia" that has to be overcome in order to enable researchers in the field to utilize Bayesian methods to formulate their research challenges. One solution to this problem is to expose software engineering researchers to the core principles and benefits of Bayesian reasoning, through coursework and tutorials that include extensive case studies. Motivated by this thought, the authors recently offered a tutorial on "Bayesian Methods for Data Analysis in Software Engineering" at the International Conference on Software Engineering (ICSE-2010) [29].

To summarize, this paper advocates the use of Bayesian reasoning methods as a widespread research paradigm, in order to address the representation, inference and learning challenges in the field of software testing. The ability of Bayesian methods to explicitly account for the unique characteristics of each software product will enable researchers to make significant inroads on the hard challenges in the domain. As a result, robust software products will be created, which will have a major impact on the economic growth and technological advancement of modern society.

4. REFERENCES

[1] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *International Symposium on Software Testing and Analysis*, Trento, Italy, July 2010.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 2008.

[3] B. W. Bohem and K. Sullivan. Software economics: A roadmap. In *International Conference on Software Engineering*, pages 319–343, Limerick, Ireland, 2000.

[4] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA*, pages 195–205, 2004.

[5] L. Briand. Novel applications of machine learning in software engineering. In *International Conference on Quality Software*, pages 3–10, 2008. Keynote Speaker.

[6] L. C. Briand, Y. Labiche, and X. Liu. Using machine learning to support debugging with tarantula. In *ISSRE*, pages 137–146, Los Alamitos, CA, USA, 2007.

[7] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, 2004.

[8] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *ICSE*, pages 144–154, 2009.

[9] S. W. Chulani, B. W. Bohem, and B. Steecs. Bayesian analysis of empirical software engineering cost models. *IEEE Transactions on Software Engineering*, 25(4):573–583, 1999.

[10] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Publishing House, 1991.

[11] T. G. Dietterich, P. Domingos, L. Getoor, S. Muggleton, and P. Tadepalli. Structured Machine Learning: The Next Ten Years. *Machine Learning*, 73:3–23, 2008.

[12] M. Feng and R. Gupta. Learning universal probabilistic models for fault localization. In *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, Toronto, Canada, June 2010.

[13] N. E. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra. Predicting software defects in varying development life cycles using bayesian nets. *Information & Software Technology*, 49(1):32–43, 2007.

[14] N. E. Fenton, M. Neil, W. Marsh, P. Hearty, L. Radlinski, and P. Krause. On the effectiveness of early life cycle defect prediction with bayesian nets. *Empirical Software Engineering*, 13(5):499–537, 2008.

[15] M. Gallaher and B. Kropp. Economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002.

[16] C. Ghezzi and D. Mandrioli. The Challenges of Software Engineering Education. In *International Conference on Software Engineering*, pages 637–638, 2005.

[17] J.-J. Gras, R. Gupta, and E. Perez-Minana. Generating a test strategy with bayesian networks and common sense. *Practice And Research Techniques, Testing: Academic and Industrial Conference on*, 0:29–40, 2006.

[18] M. J. Harrold. Testing: a roadmap. In *International Conference On Software Engineering - Future of SE Track*, pages 61–72, 2000.

[19] S. S. Heckman and L. A. Williams. A model building process for identifying actionable static analysis alerts. In *International Conference on Software Testing*, pages 161–170, 2009.

[20] D. S. Hochbaum. *Approximation Algorithms for NP-hard Problems*. PWS Publishing Co., Boston, MA, USA, 1997.

[21] L. Kaelbling, M. Littman, and A. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101:99–134, 1998.

[22] M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 388–396. ACM, 2003.

[23] E. Lehmann and J. P. Romano. *Testing Statistical Hypotheses*. Springer Texts in Statistics, 2005.

[24] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, 2005.

[25] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *FASE*, pages 276–290, 2007.

[26] D. Notkin. Software, software engineering and software engineering research: Some unconventional thoughts. *Journal of Computer and Science Technology*, 24(2):189–197, 2009.

[27] P. C. Pendharkar, G. H. Subramanian, and J. A. Rodger. A probabilistic model for predicting software development effort. *IEEE Transactions on Software Engineering*, 31(7):615–624, 2005.

[28] M. Sridharan and A. S. Namin. Prioritizing mutation operators based on probabilistic sampling. In *ISSRE*, November 1-4 2010.

[29] M. Sridharan and A. S. Namin. Tutorial on bayesian methods for data analysis in software engineering. In *International Conference on Software Engineering*, pages 477–478, 2010.

[30] J. Stamelos, L. Angelis, P. Dimou, and E. Sakellaris. On the use of bayesian belief networks for the prediction of software productivity. *Information and Software Technology*, 45(1):51–60, 2003.

[31] R. L. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.

[32] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, Cambridge, USA, 2005.

[33] C. van Koten and A. R. Gray. An application of bayesian network for predicting object-oriented software maintainability. *Information and Software Technology*, 48(1):59–67, 2006.

[34] D. A. Wooff, M. Goldstein, and F. P. A. Coolen. Bayesian graphical models for software testing. *IEEE Trans. Softw. Eng.*, 28(5):510–525, 2002.

[35] D. Zhang and J. J. P. Tsai. Machine learning and software engineering. *Software Quality Journal*, 11(2):87–119, 2003.