Defect Prediction and Software Risk
Promise’14 Keynote/Tutorial
Torino

Audris Mockus
Department of Electrical Engineering and Computer Science
University of Tennesse, Knoxville
and Avaya Labs Research
audris@utk.edu

[2014-09-17]
Defining Features of Software Repositories

Not experiment data

Definition (Operational Data (OD))

Digital traces produced in the regular course of work or play (i.e., data generated or managed by operational support (OS) tools)

- no carefully designed measurement system
Challenges of OD

- ML/Statistics assume experiment data
- Treacherous - unlike experimental data
  - Multiple contexts: no two events have the same context
    - Observables represent a mix of platonic concepts
  - Missing events: not everything is observed
  - Incorrect, filtered, or tampered with
- Continuously changing
  - OS systems and practices are evolving
  - New OS tools are being introduced in SE and beyond
Outline

- Fascination with defects
- Core issues in common approaches
- Assumptions used in defect models
- Domains and dimensions
- Costs and benefits
- Recommendations
Fascination with defects in SE

- How to not introduce defects?
  - Requirements and other process work
  - Modularity, high-level languages, type-checking and other LINT-type heuristics, garbage collection, . . .
  - Verification of software models

- How to find/eliminate defects?
  - Inspections
  - Testing
  - Debugging

- How to predict defects?
  - When to stop testing and release?
  - What files, changes will have defects?
  - How customers will be affected?
Some applications of defect models

- Faults remaining, e.g., [6]
- Repair effort, e.g., [11, 19]
- Focus QA on [where in the code] faults will occur, e.g., [23, 7, 10, 24, 1, 22, 28]
- Will a change/patch result in any faults [18, 9]
  - such data are rare, may require identification of changes that caused faults
- Impact of technology on defects, e.g., [3, 2]
- Tools, e.g., [5, 27], benchmarking, e.g., [15], availability/reliability, e.g., [8, 21, 12]
State of defect prediction

- **No context**
  - User reported issues are not separated
  - Stable releases are rarely identified
  - Users and usage [20, 17] not taken into account
  - Project domain not considered

- **Missing**
  - In FLOSS commits are rarely linked to defect IDs
  - Not all defects are ever identified
    - Better quality software has more defects
    - Static analysis discovered defects don’t appear to overlap with end-user-detected defects

- **Wrong**
  - Fixes not defects are known
  - Fixes may not fix
  - May fix a different defect
Practice: a real SE problem?

- Easy to overfit: past changes suffice [7, 10]
- Not all defects are created equal:
  - high-impact defects [25]
- Can’t expect to act on $>1\%$ of the code
- Prediction not enough:
  - tell what and why to do on $<1\%$ of code that has 60+\% fixes to customer-reported defects [16]
- Domain matters
Post release defects for two releases

Defect prediction has to account for software use!
Tip of the iceberg

Service Requests
- alarms from systems
- by phone
- through website
- chat

96% Handled by Services Technicians

Technical Escalations to Backbone Engineers (Tier III)

Product Escalations (Tier IV)

< 1% Customer Found Defects

Figure by Ravi Sethi
Defect prediction — perpetum mobile

- Why predictors do not work in practice?
  - Customer-reported defects have little to do with code or development process
  - Overfitting [26]
  - Prediction effort not considered [28]
  - Different projects have different needs

- Why people engage in irrational behavior, e.g., defect prediction?
  - The promise to see the future is irresistible.
  - The promise is phrased in a way the absurdity is well concealed.
How the deception is perpetrated (1/2)?

- By not comparing to naive methods, e.g., locations with most changes
- By not verifying that it provides benefits to actual developers/testers — “we test features not files” or “we need to have at least some clues what the defect may be, not where”
- By selecting misleading evaluation criteria, e.g., focusing on 20% of the code that may represent more than a release-worth of effort
How the deception is perpetrated (2/2)?

- By comparing Type I,II errors of a product with 40% defect rate to a product with 0.5% rate.
- By suggesting an impractical solution, e.g., how many SW project managers can competently apply an involved AI technique?
- By selecting complicated hard-to-understand prediction method, e.g., BN models with hundreds of (mostly implicit) parameters.
Then why do it?!?//1111one/
Then why do it?!?

To summarize the historic data in a way that may be useful for expert developers/testers/managers to make relevant design, QA, and deployment decisions

- E.g., [16]
  - Make Risk Transparent
    - What expertise was lost
    - What parts of code will have customer defects
  - Make Risk Reduction Actionable
    - Why risk is high
    - What is the nature of risk
    - What are cost-effective actions
    - Who can implement them
Some approaches used to model defects

- **Mechanistic:** e.g., a change will cause a fault
- **Invariants:** e.g., ratio of post-SV defects to pre-SV changes is constant
- **Data driven**
  - All possible measures
  - Principal components (measures tend to be strongly correlated),
  - Fitting method
- **Mixed:** a mix of metrics from various areas that each has a reason to affect defects, but a regression or AI method are used to find which do
Mechanism to the extreme

- **Axiom 1**: a change will cause $\mu$ faults after time $\lambda$ [19]
  - Empirical relationship between changes and defects is well established
  - Non fixes can be predicted only by knowing future needs
    - use them to predict fixes
  - The $-\log$(Likelihood) is

$$
\sum_i \mu N_{t_i} \left(1 - e^{-\lambda(t-t_i)}\right) - B_{[0,t]} \log(\mu \lambda) - \\
\sum_{s_k} B_{s_k} \log \left( \sum_{i: t_i < s_k} e^{-\lambda(s_k-t_i)} \right)
$$
Mechanism to the extreme

Weekly normalized MRs

- New feature MRs
- Actual Defect MRs
- Predicted Defect MRs (Jan, 2001)
- Predicted Defect MRs (Nov, 2001)

Calendar Weeks
Invariance to the extreme

- **Axiom 2:** The history of MRs for release $n$ will be a scaled and shifted version of the history of MRs for releases $n - 1$, $n - 2$, ... [11]
  - Anything can be predicted: inflow, resolution, test defects, customer reported defects, number of people on the project, release date, effort ...
Invariance to the extreme

- Old project inflow
- Old project outflow
- New project inflow
- New project outflow

- Prediction Done: March 1, 2003
- Predicted GA: July 15, 2003
- Actual GA: Aug 26, 2003
Most common approach

- **Axiom 3:** $\exists f : \forall L, f(m, L) = d(L)$ that given measures $m$ will produce the number of defects $d(L)$ at location $L$

- **Goal:** discover
  - $\hat{f}(m, L) = \arg \min \sum_L \| f(m, L) - d(L) \|

- **Common measures $m$**
  - Code: structural, OO, call/data flow, [3]
  - Process: change properties, age, practices, tools, [2]
  - People: experience, org-change, location, [13]
Locations \( L \)

- Lines, functions, files, packages/subsystems, entire system
- Functionality (features)
- Chunks — groups of files changed together
- Changes — MRs/work items and their hierarchy
- Geographic locations
- People/groups
- Tools/process/practices
Defects $d$

- Customer reported defects
- Alpha/Beta defects
- Customer requested enhancements
- System test reported
- Found in integration/unit test/development
- Higher severity levels
- Static analysis
- Any changes
What predictors may contribute?

- The value may not be in seeing the future but in understanding the past: gain insights
  - Formulate hypotheses
  - Create theories
  - Suggest ideas for tools or practices
- Domain specific questions/analysis based on the cost-benefit analysis
- Focus QA [16]
  - Instead of telling what files will fail, tools that help experts assess situation and evaluate actions may prove more useful
  - Need to find sufficiently small set and type of locations to match resources that could be devoted for QA
Utility function: value of prevention

- Increases sales/market share [14]
- Reduces costs to repair:
  - Domain: low cost for web service, high cost for embedded, heavy/large consumer products, aerospace
  - Number of customers: few customers can be served by the development group itself
- Reduce cost of outage/malfunction:
  - Domain: low for desktop apps, high for aerospace, medical, or large time-critical business systems (banking, telephony, Amazon, Google)
  - Number/size of customers: fewer/smaller customers → less cost
Utility function: costs of prevention

- Utility of the prediction in prevention
  - Reduce the size of $L$ identified as risky
  - i.e. the cost of test all inputs for all configurations
- Low cost: internal customer (more control over environment), web services (few installations),
- High-cost: components, real-time, multi-vendor, large customer base
- Other considerations
  - Will quick repair of field problems count as prevention?
  - Cost of alpha/beta trials
  - Cost of testing
  - Cost of better requirements/design/inspections
Will prediction reduce prevention costs below the repair costs?
From domains to dimensions

- NASA: single use, limited replacement/update, errors critical, often completed by contractors
- Cloud/Mobile: few installations, many users, costly downtime, easy repair/QA
- Mobile: many distributed installations, many users, easy repair
- Consumer devices: many users, expensive to replace somewhat alleviated by Internet connectivity
- Internal projects: single user, no difference between testing and post-SV
Relevant dimensions

- **Impact of defects**
  - Domain: medical, productivity, cloud
  - Market share

- **Cost of prevention**
  - Scale/complexity of software
  - Complexity of the operating environment: e.g., multi-vendor
  - Resources needed to test/inspect/fix

- **Cost of repair**
  - Few, internal users/installations
  - Easy/inexpensive to upgrade
Which domains are likely to benefit?

Cost of repair

Cost to prevent

Impact of outage

Productivity

Cloud services

Safety

Internal

Communications

Mainframe

Medical

NASA

Bulky consumer electronics
Resist the urge to be astrologer (Method)

- Method [14]
  - Understand practices of using operational systems
    - Establish Data Laws
    - Use other sources, experiment, . . .
  - Use Data Laws to
    - Recover the context, correct data, impute missing
  - Don’t confuse defects with quality
    - A tiny fraction of user-observed issues are ever identified as defects
    - A tiny fraction of defects would ever affect end-users
Resist the urge to be astrologer (Practice)

- It's about engineering quality software
  - Not all defects matter:
    - "for one release we tested new features — customers hated it, we instrumented it in the field and found no one using new features. We then tested basic functionality and customers were happy.”
  - Consider relevant dimensions and the utility
  - Prediction effort matters
    - Compare to naive/simple predictors
    - Make historic data actionable: leave it expert developers/testers/managers to make relevant design, QA, and deployment decisions
Data Analysis Tutorial

- Fundamentals of Digital Archeology
  - https://github.com/fdac/syllabus
  - Data Analysis tutorial:
    http://ec2-54-164-167-251.compute-1.amazonaws.com:8899/
Bibliography I

Predicting fault-prone components in a java legacy system. 

Using version control data to evaluate the impact of software tools: A case study of the version editor. 

A validation of object-oriented design metrics as quality indicators. 

Software dependencies, the structure of work dependencies and their impact on failures. 

Hipikat: A project memory for software development. 
TSE, 31(6), 2005.

When should one stop testing software? 

Predicting fault incidence using software change history. 

Software reliability research. 

A large-scale empirical study of just-in-time quality assurance. 
Comparative assessment of software quality classification techniques: An empirical case study. 

Analogy based prediction of work item flow in software projects: a case study. 

Empirical estimates of software availability of deployed systems. 

Organizational volatility and its effects on software defects. 
In *ACM SIGSOFT / FSE*, pages 117–126, Santa Fe, New Mexico, November 7–11 2010.

Engineering big data solutions. 
In *ICSE'14 FOSE*, 2014.

Two case studies of open source software development: Apache and Mozilla. 

Risky files: An approach to focus quality improvement effort. 

Interval quality: Relating customer-perceived quality to process quality. 
Predicting risk of software changes.

Understanding and predicting effort in software projects.

Drivers for customer perceived software quality.

ISBN: 0-07-044093-X.

[22] N. Nagappan, B. Murphy, and V. R. Basili.
The influence of organizational structure on software quality: an empirical case study.

Predicting fault-prone software modules in telephone switches.

Predicting the location and number of faults in large software systems.

High-impact defects: a study of breakage and surprise defects.
On measuring and correcting the effects of data mining and model selection.

[27] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll.
Predicting source code changes by mining change history.

Towards building a universal defect prediction model.
PostDoc Opening at UTK:

Who: An incurably curious person deeply interested in understanding the world through the observations recorded as data of every size or shape. Passion for hacking the data analysis to describe, understand, model, and present complex and dynamic interrelationships, and discover insidious data quality problems. An uncompromising striving to obtain reproducible and practically relevant results.

What: You will develop techniques to explore, understand, and model various phenomena based on very large operational data from software and related domains to shape the future of this rapidly evolving domain. You will collaborate with a multidisciplinary team of engineers, qualitative and quantitative scientists on a wide range of problems of practical significance. This position will bring analytical rigor and statistical methods to the challenges of understanding the accuracy, completeness, and relevance of data, and how it reflect people’s behavior.

Requirements:

- PhD preferred in statistics, applied mathematics, operation research, computer science or related field;
- Substantial real-world experience, especially in areas of data analysis.
- Familiarity statistical software (R, S-Plus, or similar).
- Familiarity with machine learning and/or experimental design principles.
- Proficiency with databases and scripting or programming languages (particularly Python or Java).
- Ability to draw real-world conclusions from data and recommend actions.
- Demonstrated willingness to both teach others and learn new techniques.
Abstract

Defect prediction has always fascinated researchers and practitioners. The promise of being able to predict the future and act to improve it is hard to resist. However, the operational data used in predictions are treacherous and the prediction is usually done outside the context of the actual development project, making it impossible to employ it for software quality measurement or improvement. Contextualizing, imputing missing observations, and correcting operational data related to defects is essential to gauge software quality. Such augmented data can then be used with domain- and project-specific considerations to assess risk posed by code, organization, or activities and to suggest risk-specific remediation activities.
Audris Mockus wants to know how and why software development and other complicated systems work. He combines approaches from many disciplines to reconstruct reality from the prolific and varied digital traces these systems leave in the course of operation. Audris Mockus received a B.S. and an M.S. in Applied Mathematics from Moscow Institute of Physics and Technology in 1988. In 1991 he received an M.S. and in 1994 he received a Ph.D. in Statistics from Carnegie Mellon University. He is Harlan Mills Chair Professor in the Department of Electrical Engineering and Computer Science of the University of Tennessee and is a consulting research scientist at Avaya Labs Research. Previously he worked at Software Production Research Department of Bell Labs.