

ML4SE + SE4ML

CS454 AI-Based Software Engineering
Shin Yoo

Machine Learning

- “A field of AI that uses statistical techniques to give computer systems the ability to **learn** from data, without being explicitly programmed” (Wikipedia)
- Supervised Learning: examples + desired outcome
- Unsupervised Learning: find inherent structure in data
- Reinforcement Learning: learn from feedback given to the program’s action in a changing environment

ML4SE

- Broadly speaking, this course is AI4SE: we borrow from specific sub-domains of AI to solve SE problems.
 - At the highest level, GP can be thought of as supervised learning
 - Some applications of hyper-heuristic can be thought of as reinforcement learning

Clustering

- Clustering is one of the representative form of unsupervised learning
- Whenever you suspect there are internal patterns in a problem, you can attempt clustering to reveal and exploit the pattern

Maintenance & Reverse Engineering

- Module Clustering: assign modules to clusters based on their relationships
 - B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.
 - K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, March-April 2010.

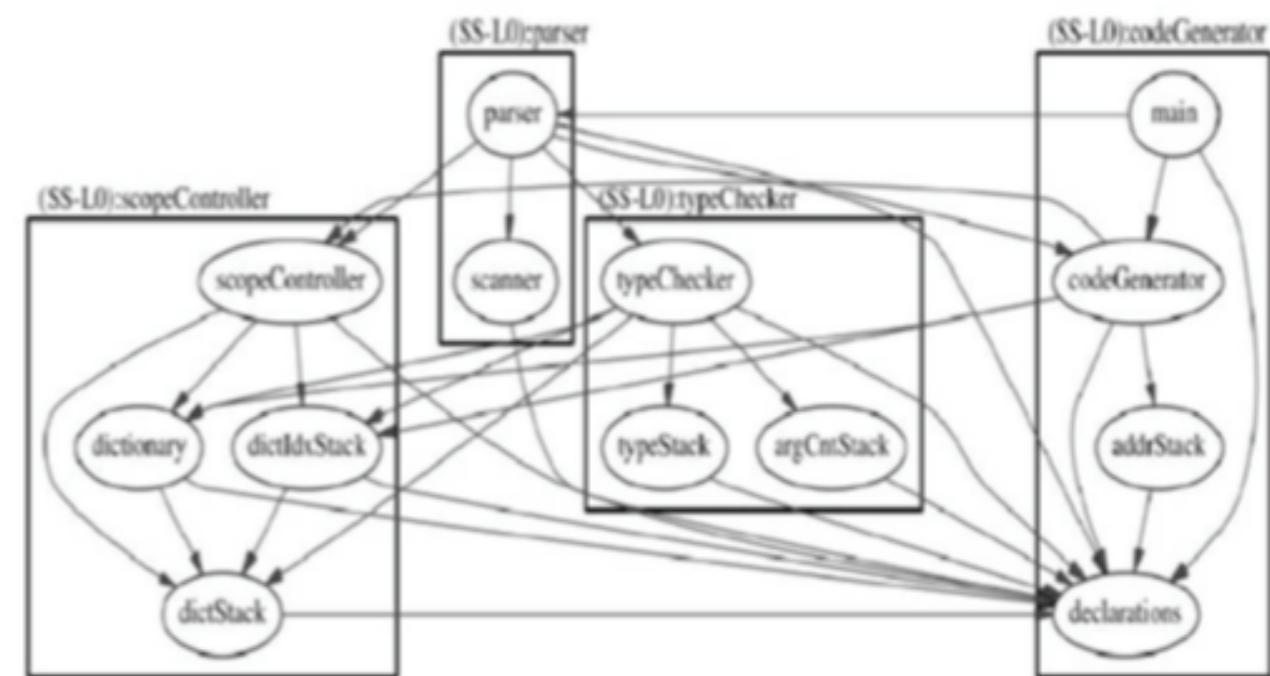
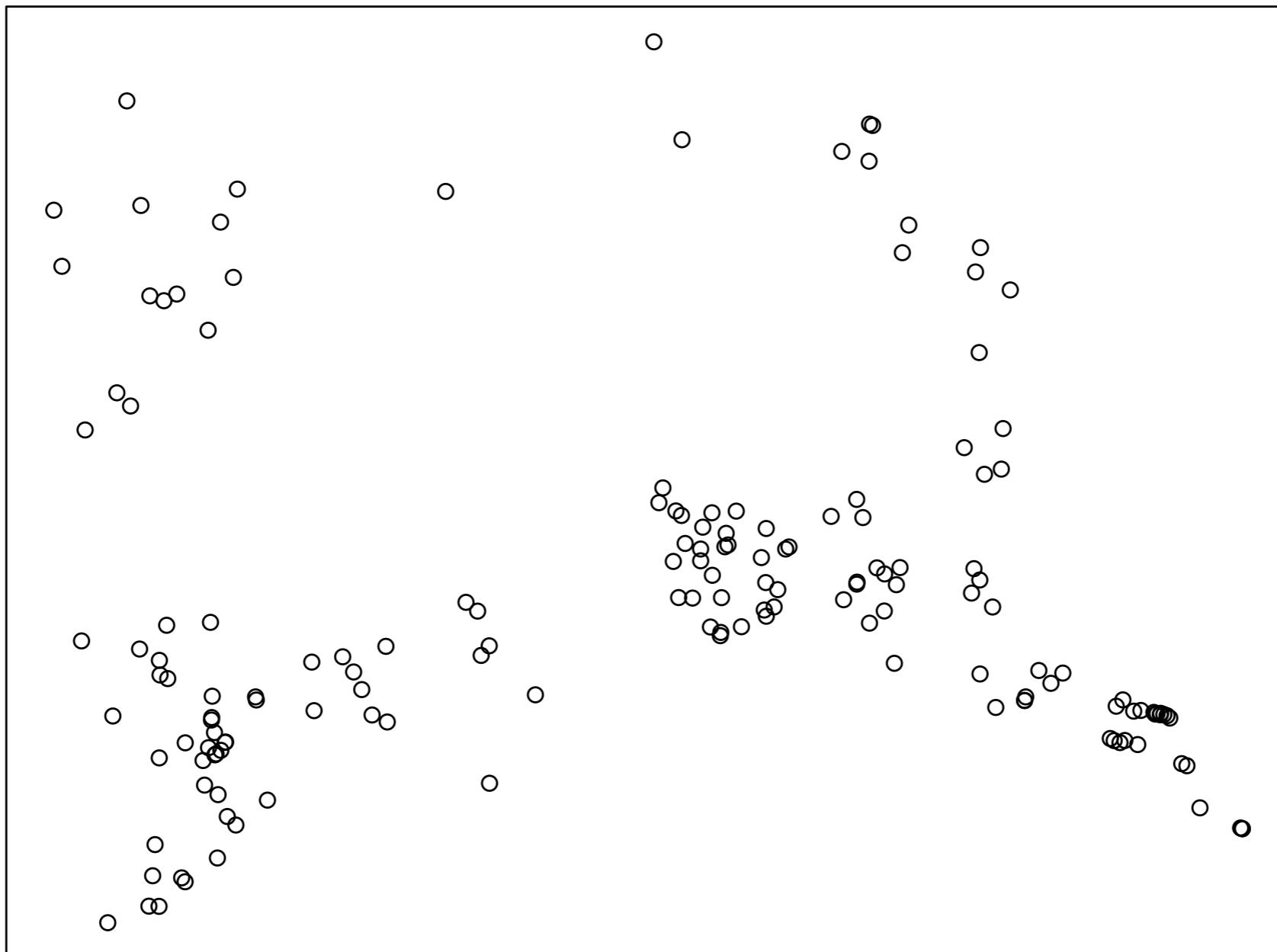


Figure 3. A Module Dependency Graph and its Modularisation using Bunch, taken from [65]

Test Case Prioritisation

Multi-dimensional Scaling of Test Case Profiles: space



Case-Based Reasoning

- P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization, ICSME 2006
- Human testers make pairwise comparison between test cases
- CBR learns to put priority scores to test cases, based on human examples
- Effective, but human comparison is extremely expensive

Using the Case-Based Ranking Methodology for Test Case Prioritization

Paolo Tonella, Paolo Avesani, Angelo Susi
ITC-irst, Trento, Italy
{tonella, avesani, susi}@itc.it

Abstract

The test case execution order affects the time at which the objectives of testing are met. If the objective is fault detection, an inappropriate execution order might reveal most faults late, thus delaying the bug fixing activity and eventually the delivery of the software. Prioritizing the test cases so as to optimize the achievement of the testing goal has potentially a positive impact on the testing costs, especially when the test execution time is long.

Test engineers often possess relevant knowledge about the relative priority of the test cases. However, this knowledge can be hardly expressed in the form of a global ranking or scoring. In this paper, we propose a test case prioritization technique that takes advantage of user knowledge through a machine learning algorithm, Case-Based Ranking (CBR). CBR elicits just relative priority information from the user, in the form of pairwise test case comparisons. User input is integrated with multiple prioritization indexes, in an iterative process that successively refines the test case ordering. Preliminary results on a case study indicate that CBR overcomes previous approaches and, for moderate suite size, gets very close to the optimal solution.

1. Introduction

Testing amounts for a large proportion of the software development and evolution effort. This is especially true for the system level testing, that typically occurs before each major release of the software. During system testing the whole application is exercised in a realistic setting. Correspondingly, the opportunities for automation are often inferior with respect to the previous testing phases (unit and integration). In fact, it might be hard to run the whole application unattended and to simulate any asynchronous input (e.g., interactive inputs) the application may receive. In such cases, system testing can last days or weeks and can involve substantial human effort.

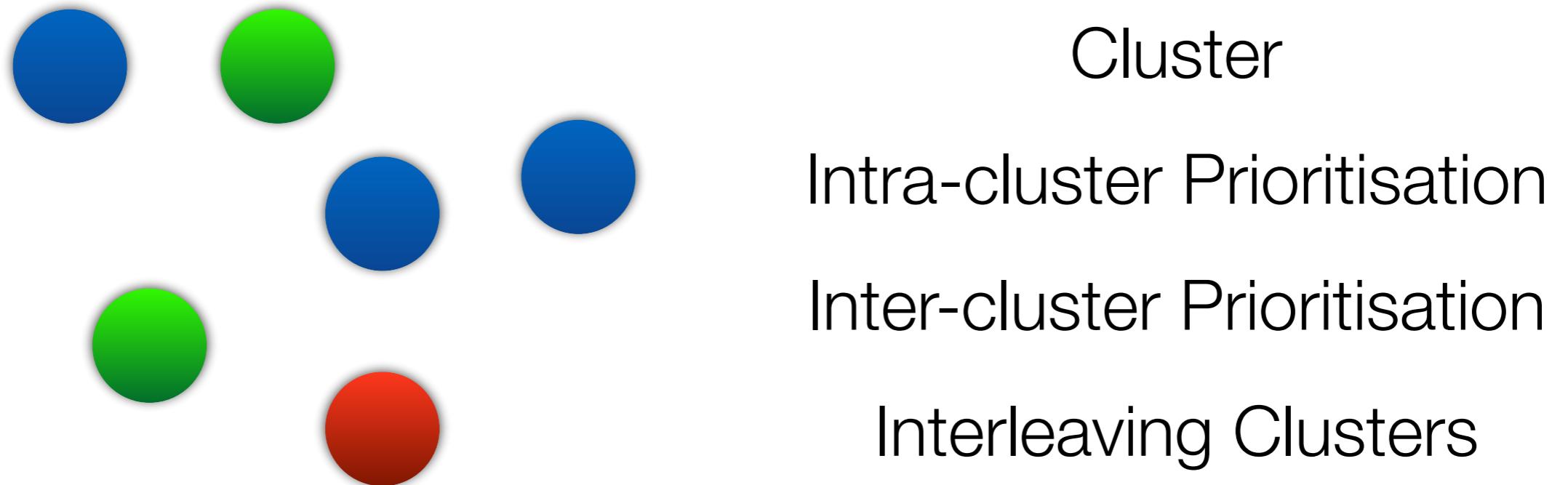
Test case prioritization aims at finding an execution order for the test cases which maximizes a given objective function. Among the others, the most important prioritization objective is probably discovering faults as early as possible, that is, maximizing the *rate of fault detection*. In fact, early feedback about faults allows anticipating the costly activities of debugging and corrective maintenance, with a related economical return. When the time necessary to execute all test cases is long, prioritizing them so as to discover most faults early might save substantial time, since bug fixing can start earlier.

Previous work on test case prioritization [6, 11, 13, 14, 15] is based on the computation of a prioritization index, which determines the ordering of the test cases (e.g., by decreasing values of the index). For example, the coverage level achieved by each test case was used as a prioritization index [13]. Another example is a fault proneness index computed from a set of software metrics for the functions exercised by each test case [6].

In this paper, we propose to incorporate user knowledge into the prioritization process and to integrate multiple prioritization indexes through the CBR (Case-Based Ranking) machine learning algorithm. CBR learns the target ranking from two inputs: a set of possibly partial indicators of priority and pairwise comparisons elicited from the user (cases). On one hand, all the information that can be gathered automatically about the test cases (coverage levels, fault proneness metrics, etc.) is used by CBR to approximate the target ranking. On the other hand, the user is involved in the prioritization process to resolve the cases where contradictory or insufficient data are available. The contribution required from the user consists of very local information and has the form of a pairwise comparison. Given two test cases, the user is requested to indicate the one that should be given higher priority. No quantification and no global evaluation is required. No consistency, such as transitivity, in the elicitation process is assumed. CBR operates iteratively and it produces a provisional ordering at each iteration. Thus, prioritization can be stopped at any time and CBR provides the user with the last ordering produced. Thus, the human effort dedicated to the prioritization process can be calibrated arbitrarily.

The main contributions of this paper over the state of the

Interleaved Clusters Prioritisation



S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge. In Proceedings of International Symposium on Software Testing and Analysis, ISSTA 2009, pages 201–211. ACM Press, July 2009.

Classification/Prediction

- To identify to which of a set of categories a new example belongs
 - Defect Prediction / Fault Localisation: Is this statement/method/file (likely to be) faulty or not?
 - Hypotheses
 - “If a file goes through an unusually high number of changes, it is more likely to be faulty”
 - “If a file is modified by an unusually high number of developers, it is more likely to be faulty”

Defect Prediction

- Collect past test history as well as various features leading up to the test results
- Train a classification model
- Before a large project moves into the testing stage, feed the collected data to see which file is more likely to be faulty

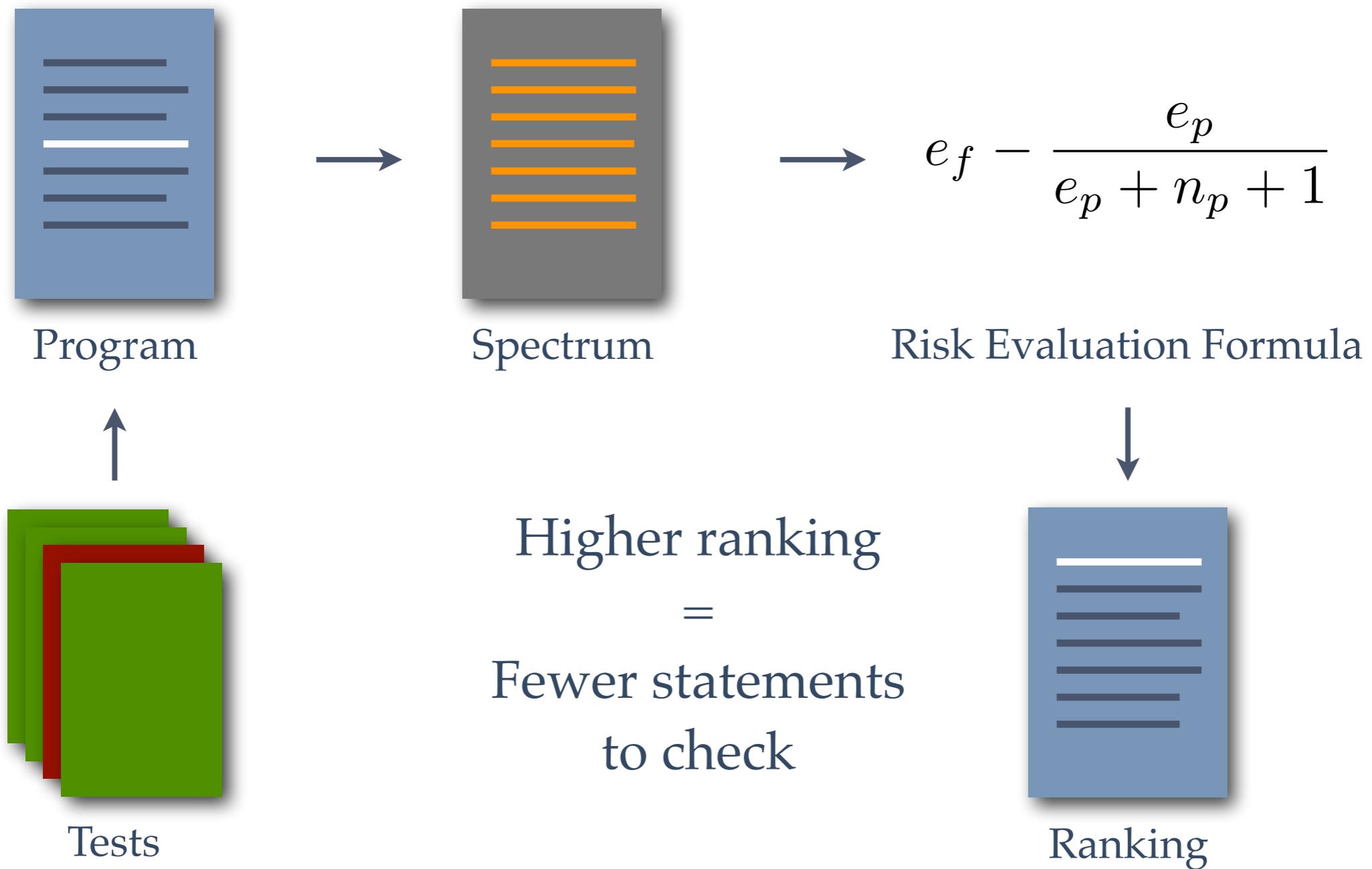
Table 4. List of Change metrics used in the study.

Metric name	Definition
REVISIONS	Number of revisions of a file
REFACTORINGS	Number of times a file has been refactored ¹
BUGFIXES	Number of times a file was involved in bug-fixing ²
AUTHORS	Number of distinct authors that checked a file into the repository
LOC_ADDED	Sum over all revisions of the lines of code added to a file
MAX_LOC_ADDED	Maximum number of lines of code added for all revisions
AVE_LOC_ADDED	Average lines of code added per revision
LOC_DELETED	Sum over all revisions of the lines of code deleted from a file
MAX_LOC_DELETED	Maximum number of lines of code deleted for all revisions
AVE_LOC_DELETED	Average lines of code deleted per revision
CODECHURN	Sum of (added lines of code – deleted lines of code) over all revisions
MAX_CODECHURN	Maximum CODECHURN for all revisions
AVE_CODECHURN	Average CODECHURN per revision
MAX_CHANGESET	Maximum number of files committed together to the repository
AVE_CHANGESET	Average number of files committed together to the repository
AGE	Age of a file in weeks (counting backwards from a specific release)
WEIGHTED_AGE	See equation (1)

Spectrum Based Fault Localisation

- Given some failing test case executions, fault localisation aims to automatically identify the program elements that are responsible for the observed failure
- One of the most widely studied technique is Spectrum-Based Fault Localisation

Spectrum Based Fault Localisation

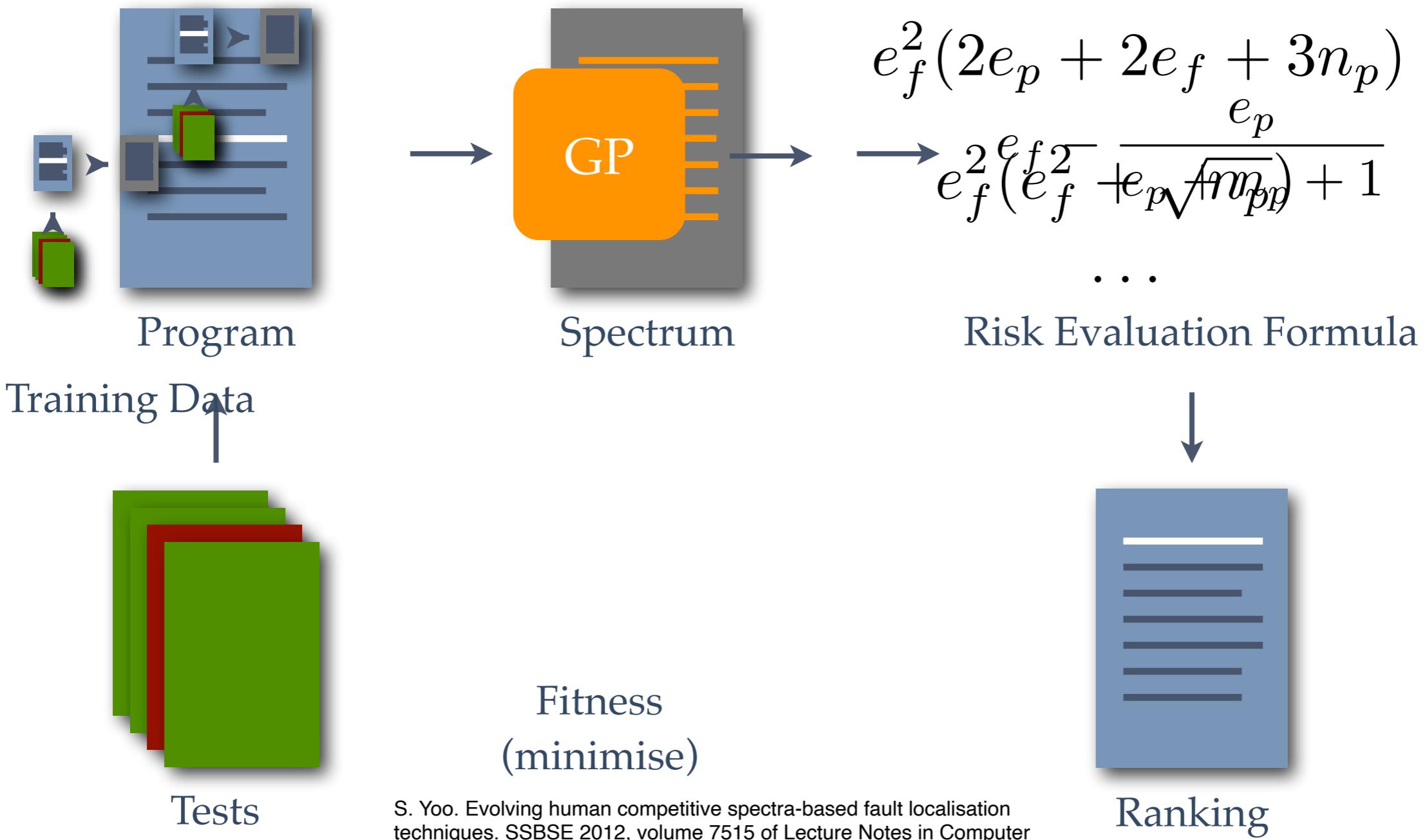


Spectrum Based Fault Localisation

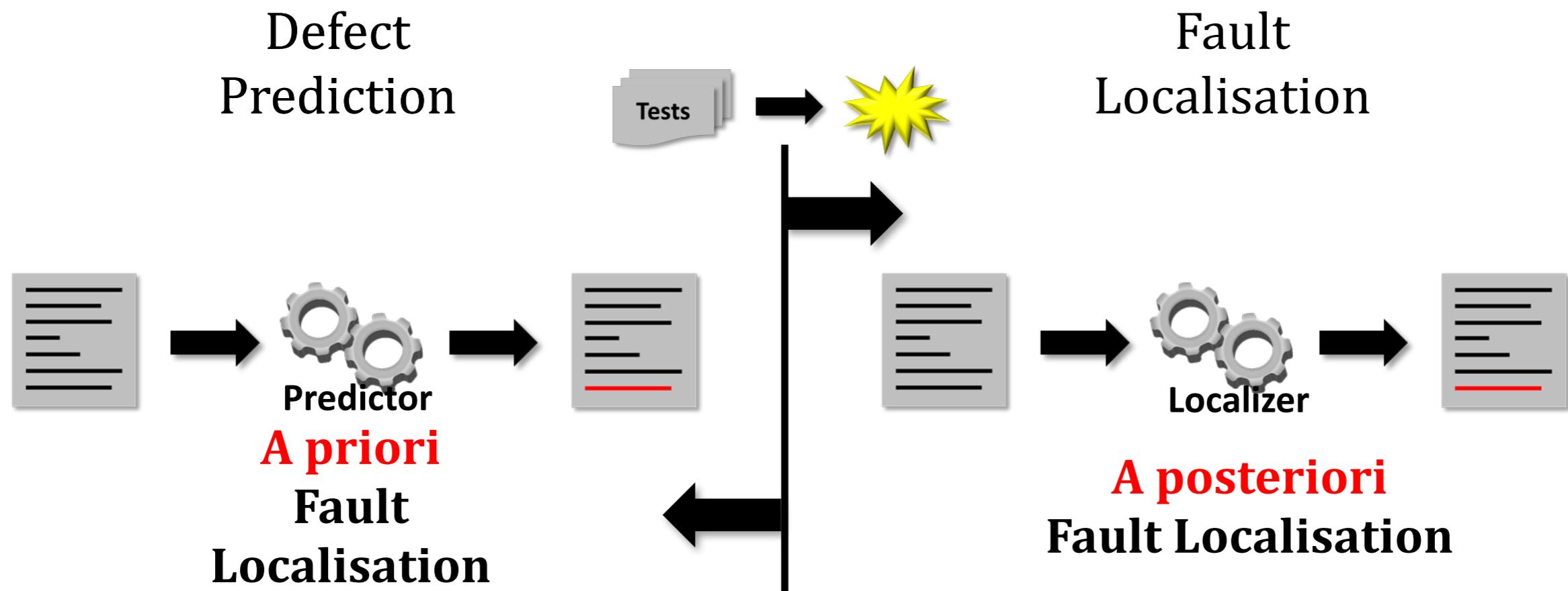
Structural Elements	Test t_1	Test t_2	Test t_3	Spectrum				Tarantula	Rank
	e_p	e_f	n_p	n_f					
s_1	•			1	0	0	2	0.00	9
s_2		•		1	0	0	2	0.00	9
s_3		•		1	0	$\frac{e_f}{e_f+n_f}$	2	0.00	9
s_4		•		1	0	$\frac{e_f}{e_f+n_f}$	2	0.00	9
s_5		•		1	$\frac{e_p}{e_p+n_p}$	0	$\frac{e_f}{e_f+n_f}$	0.00	9
s_6		•		1	$\frac{e_p}{e_p+n_p}$	0	$\frac{e_f}{e_f+n_f}$	0.33	4
s_7 (faulty)				0	2	1	0	1.00	1
s_8	•	•		1	1	0	1	0.33	4
s_9	•	•	•	1	2	0	0	0.50	2
Result	P	F	F						

$$\text{Tarantula} = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_p}{e_p+n_p} + \frac{e_f}{e_f+n_f}}$$

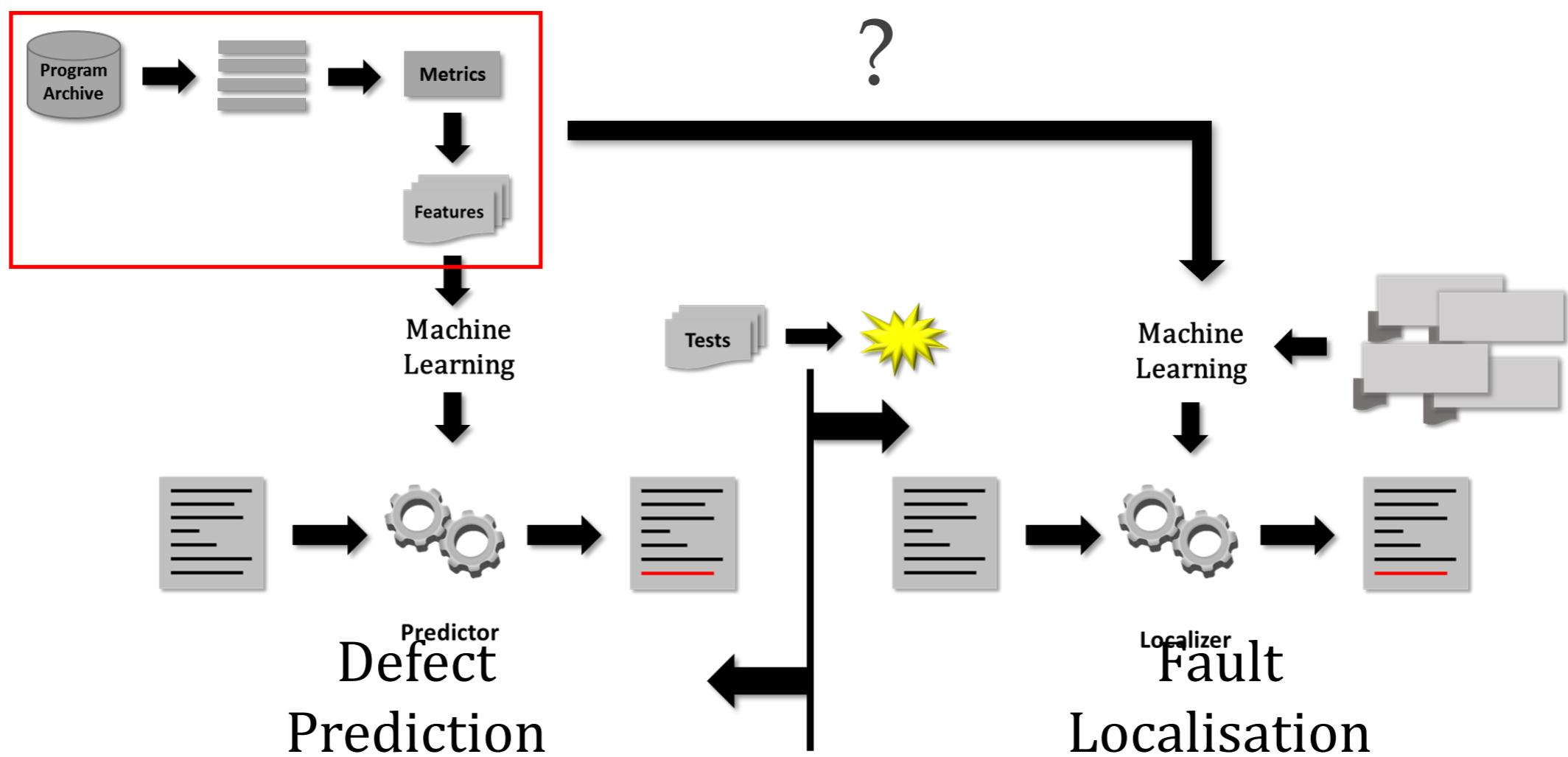
Evolving Formulae



Defect Prediction and Fault Localisation: the ML perspective



Defect prediction and Fault Localisation



Ranking Suspicious Program Elements

- Genetic Programming: we know the real faulty element in the training data. We evolve a score formula that takes input features and returns suspiciousness scores: rank the elements, and take the ranking of the real faulty element as the fitness to minimise
- Linear RankSVM: learns the ranking score function, which is linear sum of individual features
- Gaussian Process / Random Forrest: use the faulty/non-faulty labels in the training data to train classifiers - take the classification scores to rank program elements

Information Retrieval

- IR is also used to perform fault localisation
- Given a bug report, the program element responsible for the observed failure is the part of the source code that is lexically the most similar to the bug report
- See for example: R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In Automated Software Engineering (ASE), 2013.

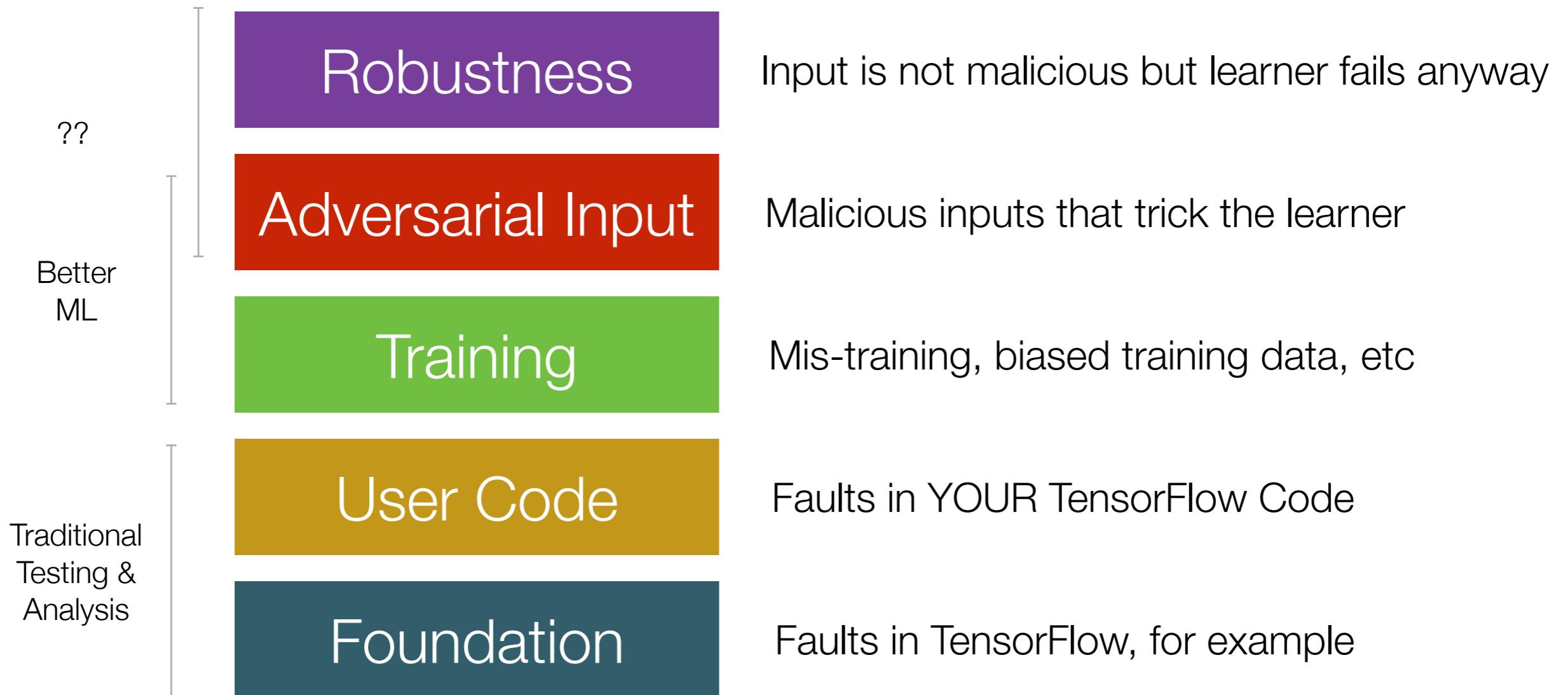
Recommendation System

- You buy X from Amazon, and give it five star. Amazon gives you “people who bought (and liked) this also bought...”
- Similarly, we can think of bug-triaging (i.e., the question of who should handle the new bug report) as:
 - You fix bug X from Project Z, and does the job well. The project gives you “people who successfully fixed bugs like this may also do well on...”

SE4ML

- We will see ML components as part of larger systems
 - Autonomous driving, photography manager, machine translation...
- ML research may deal with the core precision issues, but SE research has to deal with the larger system, especially in terms of quality assurance.
- But what is a fault in machine learning?

What are the faults?



What is the test oracle?

- For many practical ML/Deep Learning systems, inputs are raw, real-world perceptions (such as photography/video, voice, etc)
- Currently human judgement (a.k.a. data labelling) is often the only effective test oracle, but this is extremely expensive

Metamorphic Testing

- In testing, there is a widely known technique that focuses on metamorphic relationships between inputs and outputs
- Given an IO pair for program P , $y=P(x)$, if metamorphic relation f and g hold for input and output, it has to follow that $g(y) = P(f(i))$
 - For example: if P is the sine function, $f(x) = \pi - x$, $g(y) = y$. That is, if $y_0 = P(x_0)$, $y_1 = y_0 = P(\pi - x_0) = P(x_1)$
 - MT cannot replace a full oracle, but if a program violates its own metamorphic properties, **something** is wrong

Metamorphic Testing for DL

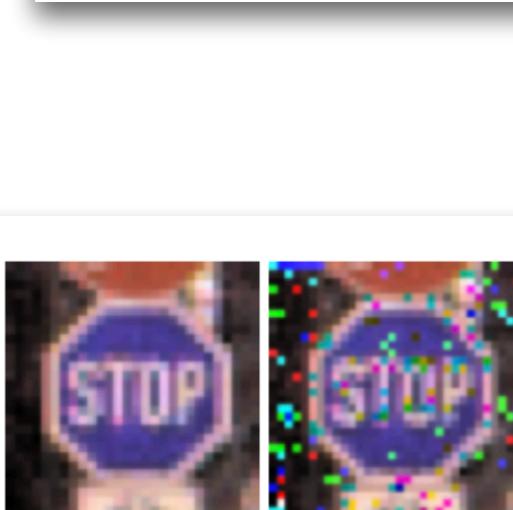
- MT has been applied to test DL robustness
 - If we apply negligible (i.e., bearable by humans) perturbation to the input, the output of the DL system should be the same
- Again, if a DL system violates this, **something** is wrong

Adversarial Examples



automobile to bird automobile to frog automobile to airplane automobile to horse

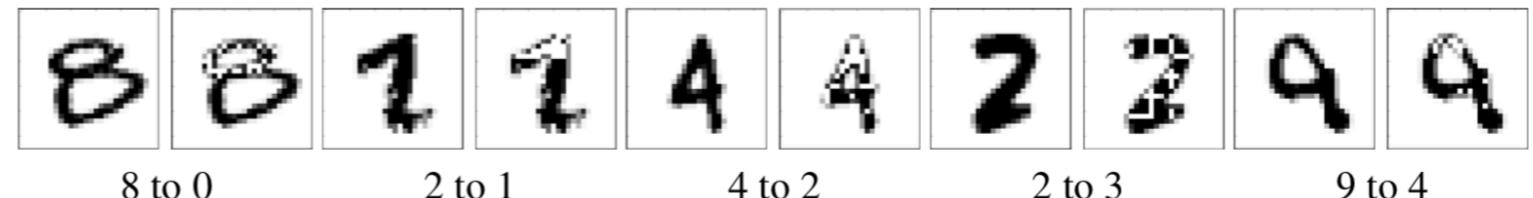
Fig. 1. Automobile images (classified correctly) and their perturbed images (classified wrongly)



“stop”
to “30m speed limit”

“80m speed limit”
to “30m speed limit”

“go right”
to “go straight”



samples for a neural network trained on MNIST

Fig. 11. Adversarial examples for the network trained on the GTSRB dataset by multi-path search

Robustness Testing

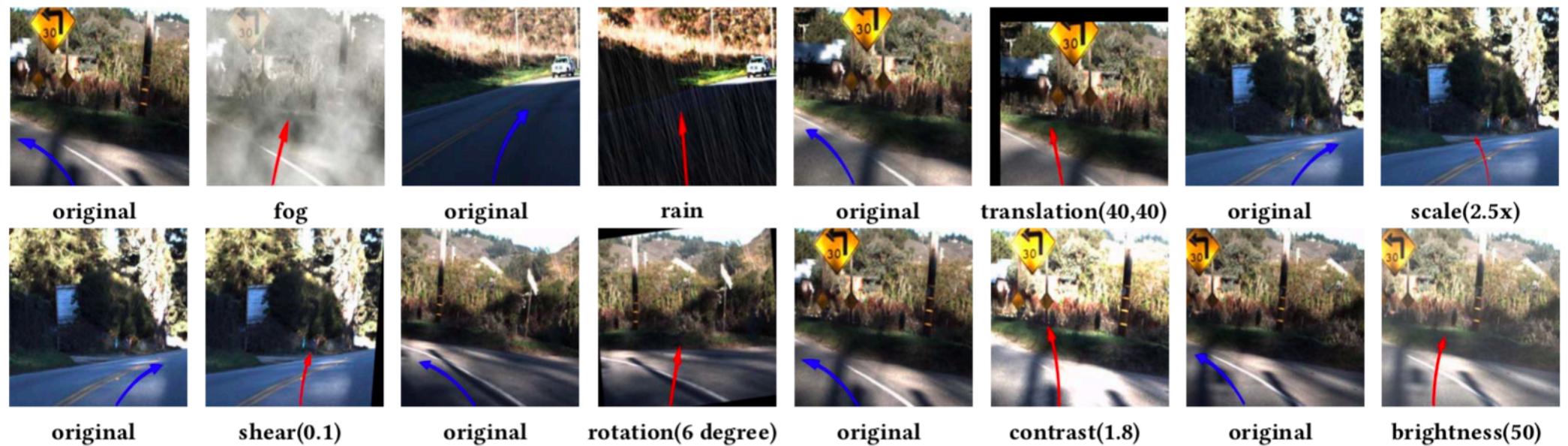


Figure 7: Sample images showing erroneous behaviors detected by DeepTest using synthetic images. For original images the arrows are marked in **blue**, while for the synthetic images they are marked in **red**. More such samples can be viewed at <https://deeplearningtest.github.io/deepTest/>.

Coverage for Deep Learning Systems

- Various coverage criteria have been suggested:
 - Neuron Coverage: given a set of test input, NC measures % of neurons that have been activated above a given threshold (e.g., 0.6)
 - k-Multisection Neuron Coverage: mark the range of neuron activation during training, divide the range into k buckets, and count the number of buckets checked during execution of the given input set
 - Strong Neuron Activation Coverage: mark the range of neuron activation during training, and count the number of neurons that are activated beyond the maximum observed activation value
- All designed to diversify neuron activation patterns

A Big Challenge

- For traditional software, inputs can be either randomly sampled, searched, or synthesised
- For DL systems that interface with the real physical world, inputs have to be collected from the real world
 - You can try random sampling or search, but would it be relevant?

Fooling DNNs with Genetic Algorithm

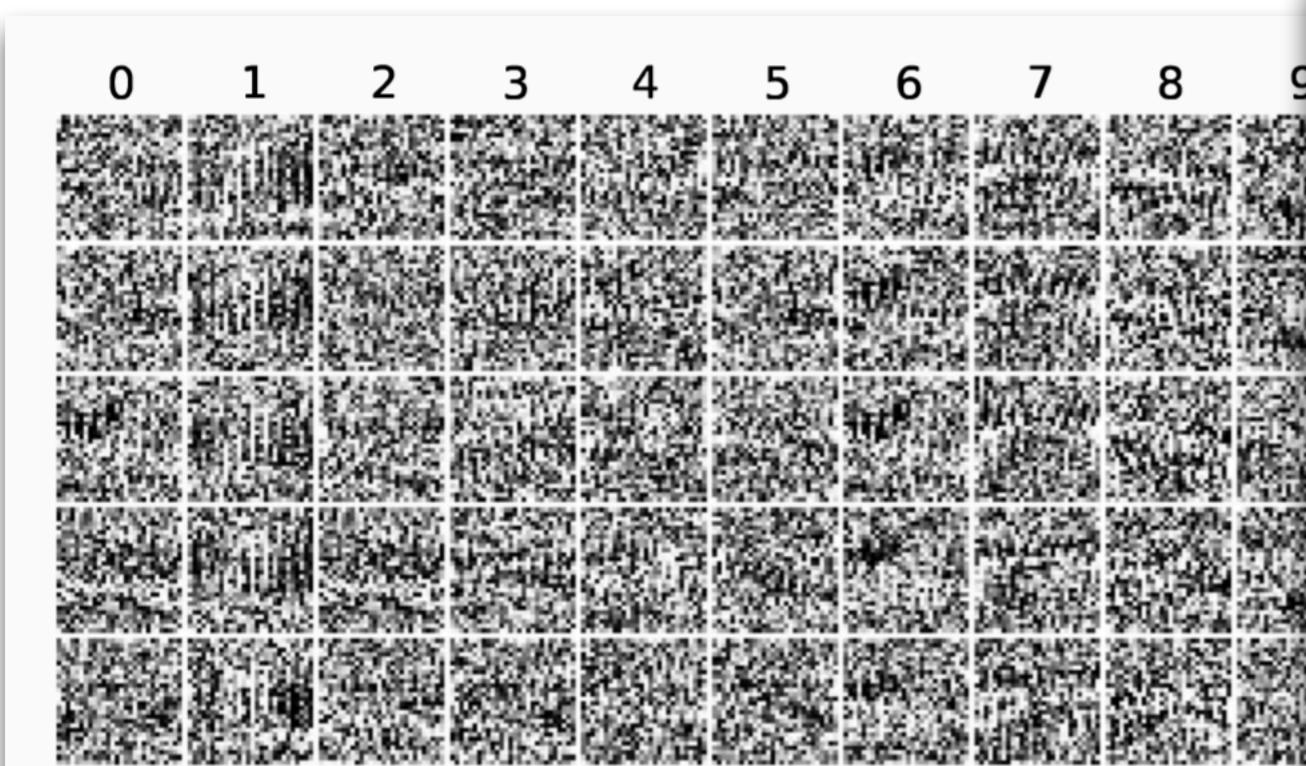


Figure 4. Directly encoded, thus irregular, images that MNIST DNNs believe with 99.99% confidence are digits 0-9. Each column is a digit class, and each row is the result after 200 generations of a randomly selected, independent run of evolution.

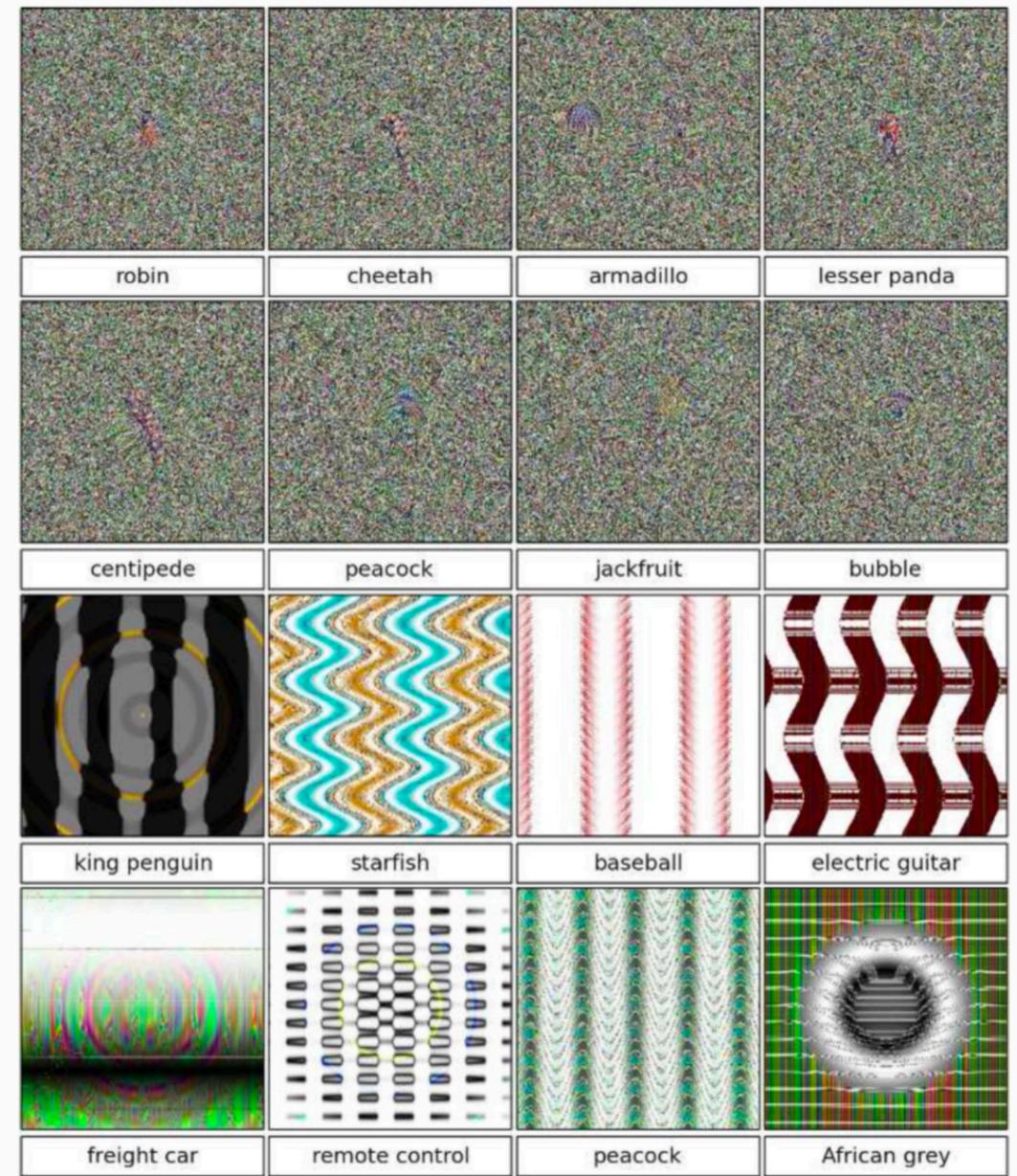


Figure 1. Evolved images that are unrecognizable to humans, but that state-of-the-art DNNs trained on ImageNet believe with $\geq 99.6\%$ certainty to be a familiar object. This result highlights differences between how DNNs and humans recognize objects. Images are either directly (*top*) or indirectly (*bottom*) encoded.

A. M. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 427–436, 2015.

Many Open Questions

- DL systems have continuous quality measure (precision/recall/accuracy); traditional systems have discontinuous quality measure (test pass/fail) - how do we connect these two worlds, when a DL system is part of a larger, conventional SW system?
- How do we automatically generate test inputs for a DL system, without going into the real world?
- How can search help? :)