COT6930: Data Mining
Combining Multiple Models To Improve Accuracy In Software Quality Modeling

Erik Geleyn

Empirical Software Engineering Laboratory
Dept. of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
(561)297-2512
egeleyn@cse.fau.edu
http://www.cse.fau.edu/esel.html

Overview

- Introduction to Software Quality Modeling
- Introduction to Case Studies
- Description of the System Studied
- Empirical Results
- Conclusions

Introduction to Software Quality Modeling

- Motivations
- Techniques
- Software Metrics
- Classification Specificities

Introduction to Software Quality Modeling

Motivations

Need for increased reliability in software systems.

Dramatic costs may be associated with failures of software systems.

Be able to have early insights of the quality of software systems.

Introduction to Software Quality Modeling

Techniques

Techniques applied to software quality modeling aim to predict numeric values or classes.

Prediction: determine the numeric value of a dependent variable such as the number of faults.

Classification: determine the class of an instance according to a quality factor (fault-prone, not fault-prone).

Introduction to Software Quality Modeling

Techniques

Techniques include:

- Statistical methods
- Analytical methods
- Machine Learning methods
- Data mining techniques
- ...

Introduction to Software Quality Modeling

Software Metrics

Measure the past to be able to predict the future.

Software metrics are attributes of the software product.

Software Quality Modeling uses software metrics collected early in the lifecycle to predict a quality factor that will only be known much later.

Introduction to Software Quality Modeling

Classification specificities

The prior probabilities of class membership are usually unbalanced.

Not fault-prone class usually much larger than the fault-prone class.

Costs associated with the different misclassification type vary.

Introduction to Software Quality Modeling

Classification specificities

Type I error: a not fault-prone module classified as fault-prone.

Type II error: a fault-prone module classified as not fault-prone.

Type II errors are more serious than Type I errors.

Introduction to Case Studies

- Definitions
- Model Evaluation

Introduction to Case Studies

Definitions

Case study: Research technique where we identify key factors that may affect the outcome of an activity and document the activity.

Projects for a case study should:

1. Be developed by a group, rather than an individual programmer.

2. Be developed by professionals, rather than students.

3. Be developed in an industrial environment rather than an artificial setting.

4. Be large enough to be comparable to real industry projects.

Introduction to Case Studies

Model Evaluation

- Resubstitution
- Data splitting
- Cross-validation
- Multiple releases
- Multiple projects

Description of the System Studied

- LLTS
- Inspection

Description of the System Studied

LLTS

System Description:

The LLTS software system is a Large Legacy Telecommunication System. LLTS was developed in a large organization by professional programmers using a proprietary high level procedural language, PROTEL.

Description of the System Studied

LLTS

Software Metrics:

48 raw software metrics were collected over four consecutive releases.

Only new or changed modules are listed in the subsequent releases.

Modules with one or more faults are considered fault-prone.

Description of the System Studied

Inspection

System description:

The two data sets were obtained from the industry. The project consisted of two large Windows-based applications used primarily for customizing the configuration of wireless products. The data sets were obtained from the initial release of these applications. The applications are written in C++, and they provide similar functionality.

Description of the System Studied

Inspection

System description:

Applications	Service Configuration Software
Language	C++
Application 1: AENSCL*	320 million
Application 1: Actual Lines of Code	28 million
Application 2: AENSCL*	300 million
Application 2: Actual Lines of Code	27.5 million
Number of source files	1406

* AENCSL is Assembly Equivalent Non-Commented Source Lines of Code

Description of the System Studied

Inspection

Software Metrics:

Symbol	Description
Product Metrics, Statement metrics	
BASE_LOC	Number of lines of code for the source file version just prior to the coding phase. This represents auto-generated code.
SYST_LOC	Number of lines of code for the source file version delivered to system test.
BASE_COM	Number of lines of commented code for the source file version just prior to the coding phase. This represents auto-generated code.
SYST_COM	Number of lines of commented code for the source file version delivered to system test.
Process Metrics	
INSP	Number of times the source file was inspected prior to system test.

Description of the System Studied

Inspection

Classification rule: If a module as 2 or more faults then it is considered fault-prone.

Empirical Results

LLTS

Determining the preferred model for a modeling technique:

The preferred model is the one with the best balance between Type I and Type II misclassification rates with Type II as low as possible.

Empirical Results

LLTS

Determining the preferred model for a modeling technique:

Empirical Results

LLTS

Overall comparision

Empirical Results

Inspection

Determining the preferred model for a modeling technique:

CII	Fit (cross validation)			Test		
	Type I	Type II	Overall	Type I	Type II	Overall
1	10.28	27.43	14.00	8.52	18.39	10.64
5	18.51	19.43	18.71	14.20	9.20	13.12
5.5	**18.67**	**18.29**	**18.59**	**15.14**	**9.20**	**13.86**
6	19.62	15.43	18.71	17.98	8.05	15.84
10	24.68	14.29	22.43	17.67	8.05	15.59

Empirical Results

Inspection

Determining the preferred model for a modeling technique:

C4.5	CII	Release 1 (cross validation)			Release 2		
		Type I	Type II	Overall	Type I	Type II	Overall
Alone	5.5	18.67	18.29	18.59	15.14	9.20	13.86
Bagging	5.5	18.67	17.71	18.46	11.99	12.64	12.13
Boosting	200	18.67	18.86	18.71	12.62	8.05	11.63
CostBoosting	24.5	17.88	15.43	17.35	13.56	12.64	13.37

Decision Stump	CII	Release 1 (cross validation)			Release 2		
		Type I	Type II	Overall	Type I	Type II	Overall
Alone	6.8	54.43	28.00	48.70	11.36	42.53	18.07
Bagging	5.8	28.16	37.14	30.11	9.46	42.53	16.58
Boosting	4.5	24.53	24.00	24.41	27.76	12.64	24.50
Logitboost	3.4	19.46	18.86	19.33	23.03	8.05	19.80
CostBoosting	7.4	26.42	22.29	25.53	31.55	4.60	25.74

Conclusions

- Systematically improves the weak learner
- Strong learner, harder to improve
- Boosting methods usually yield more substantial improvements than bagging
- Cost-Boosting yields more realistic cost ratios with the strong learner
- No tuning required