

Training on Errors Experiment to Detect Fault-Prone Software Modules by Spam Filter



Osaka Univ.

Osamu Mizuno, Tohru Kikuno

Graduate School of Information Science and Technology
Osaka University, JAPAN

ESEC/FSE2007 presentation

Idea: Fault-prone filtering

- Detection of fault-prone modules using a generic text discriminator such as a spam filter

Experiment

- SPAM filter: CRIMI4 (generic text discriminator)
- Data of fault-proneness from an OSS project (Eclipse)
- Training Only Errors(TOE) procedure

Result

- Achieved high recall (despite of low precision)



Preliminary

Fault-Prone Filtering

Experiments

-  Training Only Errors (TOE) procedure

-  Results

Conclusions

Preliminary: Fault-Prone Modules

- ❶ Fault-prone modules are:
 - ❶ Software modules (a certain unit of source code) which may include faults.
- ❷ In this study:
 - ❶ Source code of Java methods which seems to include faults from the information of a bug tracking system.

Preliminary: Spam E-mail Filtering (I)

- Spam e-mail increases year by year.
 - About 94% of entire e-mail messages are Spam.
- Various spam filters have been developed.
 - Pattern matching based approach causes a rat race between spammers and developers.
 - Bayesian classification based approach has been recognized effective[1].



[1] P. Graham, Hackers and Painters: Big Ideas from the Computer Age, chapter 8, pp. 121-129, 2004.

Preliminary: Spam E-mail Filtering (2)

- All e-mail messages can be classified into
 - Spam: undesired e-mail
 - Ham: desired e-mail
- Tokenize and learn both spam and ham e-mail messages as text data and construct corpora.

Existing e-mail



Learning (Training)

SPAM corpus

HAM corpus



Classify

SPAM Filter

SPAM

HAM

- Incoming e-mail messages are classified into spam or ham by spam filter.

Incoming e-mail



- Preliminary

- **Fault-Prone Filtering**

- Experiments

 - Training Only Errors (TOE) procedure

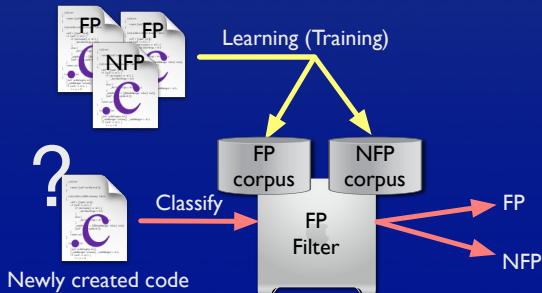
 - Results

- Conclusions

Fault-Prone Filtering

- All software modules can be classified into
 - bug-detected (fault-prone: FP)
 - not-bug-detected (not-fault-prone: NFP)
- Tokenize and learn both FP and NFP modules text data and construct corpora

Existing code modules



- Newly developed modules are classified into FP or NFP by the FP filter.

Fault-Prone Filtering: Spam Filter: CRM I I 4

- Spam filter: **CRM I I 4** (<http://crm114.sourceforge.net/>)
- Generic text discriminator for various purpose
- Implements several classifiers: Markov, OSB, kNN, ...
- Characteristic: generation of tokens.
 - Tokens are generated by combination of words (not a single word)

```
return (x + 2 * y );
```



with the OSB tokenizer

token

return	x	x +	+ 2	2 y
return	+	x 2	+ *	* y
return	2	x *	+ y	
return	*	x y	2 *	

Fault-Prone Filtering:

Example of Fault-Prone Filtering (CRM114, OSB)

Source code (m_{FP})

```
public int fact(int x) {
    return (x<=1?1:x*fact(++x));
}
```

Source code (m_{NFP})

```
public int fact(int x) {
    return (x<=1?1:x*fact(--x));
}
```

Tokens (T_{FP})

```
public int
public fact
public x
int fact
int int
...
...
x ++
x x
* fact
* ++
* x
fact ++
fact x
++ x
```

Tokens (T_{NFP})

```
public int
public fact
public x
int fact
int int
...
...
x --
x x
* fact
* --
* x
fact --
fact x
-- x
```

Training

Empty

Empty

FP
corpus

NFP
corpus

FP
Filter

Fault-Prone Filtering: Example of Fault-Prone Filtering (CRM114, OSB)

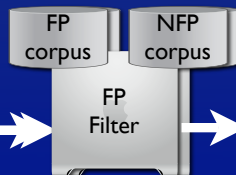
Source code (m_{new})

```
public int sigma(int x) {
    return (x<=0?0:x+sigma(++x));
}
```

Tokens (T_{FP}) Tokens (T_{new}) Tokens (T_{NFP})

public int	public int	public int
public fact	public sigma	public fact
public x	public x	public x
int fact	int sigma	int fact
int int	int int	int int
...
...
x ++	x ++	x --
x x	x x	x x
* fact	+ sigma	* fact
* ++	+ ++	* --
* x	+ x	* x
fact ++	sigma ++	fact --
fact x	sigma x	fact x
++ x	++ x	-- x

m_{new} is predicted as FP
because T_{FP} has more
similarity than T_{NFP} .



Probability:
0.52
Predicted:
FP



- 📌 Preliminary

- 📌 Fault-Prone Filtering

- 📌 **Experiments**

- 📌 Training Only Errors (TOE) procedure

- 📌 Results

- 📌 Conclusions

- Target: **Eclipse** project
- Written in Java
 - “Methods” in Java classes are considered as modules
- Date of snapshots of cvs repository and bugzilla database
 - January 30, 2007.
- Large CVS repository (about 14GB)
- Faults are recorded precisely

Experiment: Collecting FP & NFP Modules

- Track FP modules from CVS log based on an algorithm by Sliwerski, et. al[2].





[2] J. Sliwerski, et. al., When do changes induce fixes? (on fridays.). In Proc. of MSR2005, pp. 24-28, 2005.

- Search terms such as “issue”, “problem”, “#”, and bug id as well as “fixed”, “resolved”, or “removed” from CVS log, then identify a revision the bug is removed.
- Get difference from the previous revision and identify modified modules.
- Track back repository and identify modules that have not been modified since the bug is reported.
 - They are FP modules.

Experiment: Result of Module Collection

Extracted bugs from bugzilla database of Eclipse

Conditions:

-  Type of faults: Bugs
-  Status of faults: Resolved, Verified, or Closed
-  Resolution of faults: Fixed
-  Severity: Blocker, Critical, Major, or Normal

 Total # of faults: **40,627**

Result of collection

 # of faults found in CVS log: **21,761 (52% of total)**

 # of fault-prone(FP) modules: **65,782**

 # of not-fault-prone(NFP) modules: **1,113,063**



- Preliminary

- Fault-Prone Filtering

- Experiments

 - **Training Only Errors (TOE) procedure**

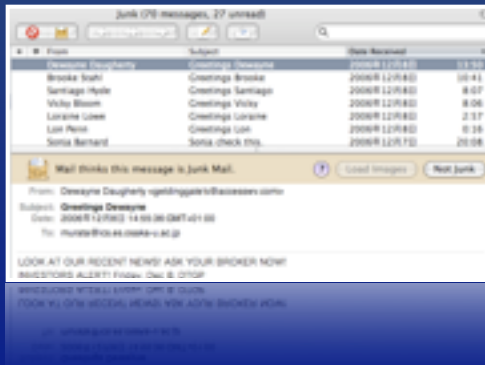
 - Results

- Conclusions



Training Only Errors Procedure

- In Spam filtering:
 - Apply e-mail messages to spam filter in order of arrival.
 - Only misclassified e-mail messages are trained in corpuses.
 - You may do this procedure in daily e-mail assorting.



- In Fault-prone filtering:
 - Apply software modules to fault-prone filter **in order of construction and modification.**
 - Only misclassified modules are trained in corpuses.



- 📌 Preliminary
- 📌 Fault-Prone Filtering
- 📌 Experiments
 - 📌 Training Only Errors (TOE) procedure
 - 📌 **Results**
- 📌 Conclusions

Evaluation Measurements

Accuracy

Overall accuracy of prediction

$$(N1 + N4) / (N1 + N2 + N3 + N4)$$

Recall

How much actual FP modules are predicted as FP.

$$N3 / (N3 + N4)$$

Precision

How much predicted FP modules include actual FP modules

$$N2 / (N2 + N4)$$

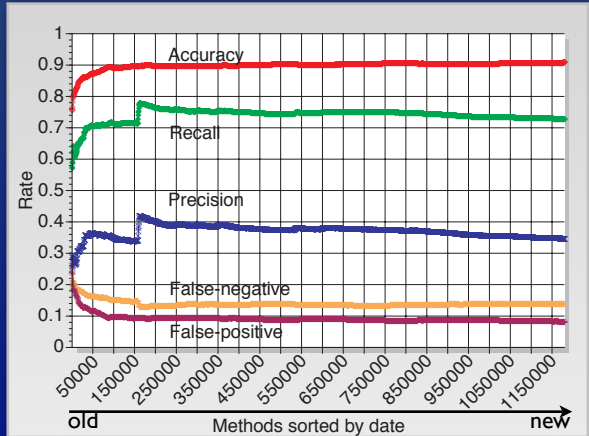
Result of prediction		Predicted	
		NFP	FP
Actual	NFP	N1	N2
	FP	N3	N4

Result of Experiment (Transition of Rates)

• All extracted modules are sorted by date, and applied FP filter one by one from the oldest one.

• Observation

• The prediction result become stable after 50,000 modules classification.



Result of Experiment (Final Accuracy)

 Cumulative prediction result at the end of TOE.

TOE - final OSB		Predicted	
		NFP	FP
Actual	NFP	1,022,895	90,168
	FP	17,890	47,892

✖ Precision: 0.347

✖ Recall: 0.728

✖ Accuracy: 0.908

 In other words,

 72% of actual FP modules are predicted as FP.




 34% of predicted FP modules include faults.



- 📌 Preliminary
- 📌 Fault-Prone Filtering
- 📌 Experiments
 - 📌 Training Only Errors (TOE) procedure
 - 📌 Results
- 📌 **Conclusions**

Threats to Validity

Threats to construction validity

-  Collection of fault-prone modules from OSS projects.
-  We could not cover all faults in bugzilla database.
-  We have to collect more reliable data in future work.

Threats to external validity

-  Generalizability of the results
-  We have to apply Fault-prone Filtering to many projects including industrial ones.





Related Works




- Much research has been done so far.
 - Logistic regression
 - CART
 - Bayesian classification
 - and more.
- Most of them use software metrics
 - McCabe, Halstead, Object-oriented, and so on.
- Intuitively speaking, our approach uses a new metric, “frequency of tokens”.



Summary

-  We proposed the new approach to detect fault prone modules using spam filter.
-  The case study showed that our approach can predict fault prone modules with high accuracy.

Future works

-  Using semantic parsing information instead of raw code
-  Using differences between revisions as an input of Fault-prone filtering
 -  Seems more reasonable...



Thank you!

Any questions?



Result of Cross Validation (From slide of MSR2007)

Result for Eclipse BIRT plugin

10-fold cross validation

Cross Validation OSB		Predicted	
		NFP	FP
Actual	NFP	70,369	16,011
	FP	2,039	7,501

Precision: 0.319

Recall: 0.786

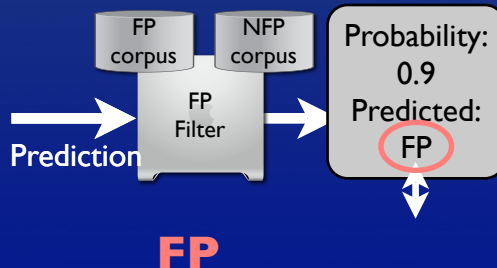
Accuracy: 0.811

- Recall is important for quality assurance.
- Precision implies the cost for finding FP modules.

Recall is rather high, and precision is rather low.

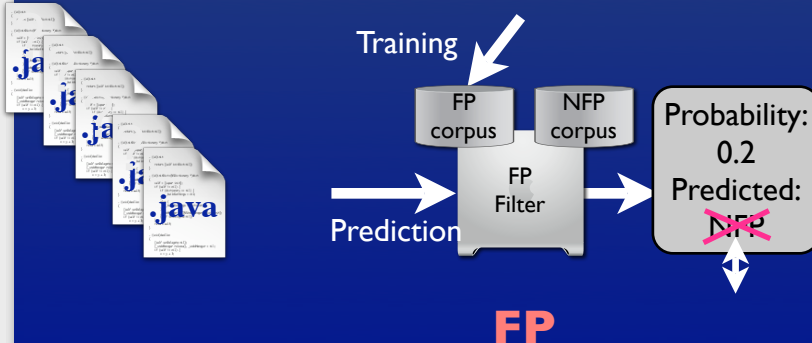
Training Only Errors Procedure

Case 2: Prediction matches to actual status



Training Only Errors Procedure

Case I: Prediction does not match to actual status



Procedure of Experiment

- Two experiments with different thresholds of probability (t_{FP}) to determine FP and NFP.
 - Changing t_{FP} may achieve higher recall
- Experiment 1:
 - TOE with OSB classifier, $t_{FP}=0.5$
- Experiment 2:
 - TOE with OSB classifier, $t_{FP}=0.25$
 - Predict more modules as FP than Experiment 1



Result of Experiment (OSB, $t_{FP} = 0.25$)

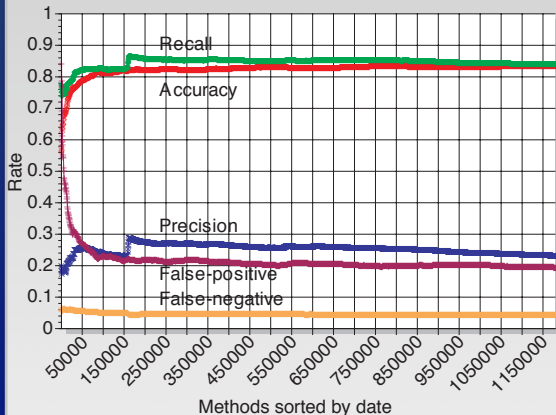
Comparison with threshold = 0.50

Precision becomes lower.

Only 1/4 of FP predicted modules hits actual faulty modules.

Recall becomes much higher.

83% of actual faulty modules can be detected.



TOE - final OSB		Predicted	
		NFP	FP
Actual	NFP	930,218	182,845
	FP	10,592	55,190

✖ Precision: 0.232

✖ Recall: 0.839

✖ Accuracy: 0.835