A Fault-Prone Module Detection Using a Spam Filter
Fault-prone Filtering

- Fault-prone module detection using a spam filtering approach[1][2]
- Uses frequency of terms like spam e-mail filtering
- Constructs both faulty and non-faulty corpuses from past modules
- Classifies an unknown module using two corpuses

How It Works (Training)

1. Training faulty and non-faulty modules using Tokenizer.

Each corpus holds frequency information of each word.
How It Works (Prediction)

2. Calculating probability by the Bayesian filter.

Fault-proneness is determined by the probability and a threshold.
# Experiment: Result of Prediction

<table>
<thead>
<tr>
<th></th>
<th>Predicted</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse (random sample)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual</td>
<td>NFP</td>
<td>FP</td>
<td></td>
</tr>
<tr>
<td>not faulty</td>
<td>12,249</td>
<td>7,093</td>
<td></td>
</tr>
<tr>
<td>faulty</td>
<td>2,972</td>
<td>16,243</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Precision: 0.696</td>
<td>Recall: 0.845</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Predicted</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse BIRT (all modules)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual</td>
<td>NFP</td>
<td>FP</td>
<td></td>
</tr>
<tr>
<td>not faulty</td>
<td>70,349</td>
<td>16,011</td>
<td></td>
</tr>
<tr>
<td>faulty</td>
<td>2,039</td>
<td>7,501</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Precision: 0.319</td>
<td>Recall: 0.786</td>
<td></td>
</tr>
</tbody>
</table>
Apply software modules to fault-prone filter in order of construction and modification. Only misclassified modules are trained in corpuses.

The accuracy improves as the number of modules increases.

Successive Prediction (TOE)
Fault-proneness filtering with static code analysis

Non-faulty

PMD or CheckStyle
(static code analyzer)

Tokenizer

Warning messages

Avoid, variables, with, short, names, like, gd, Local, variable, ', gd, ', could, be, declared, finalLocal, variable, ', cmp, Basic, ', could, be, declared, final, ...

Parameter, ', e, ', is, not, assigned, and, could, be, declared, final, ...

Parameter, ', isSelected, ', is, not, assigned, and, could, be, declared, final, ...

Non-faulty corpus

Faulty

PMD or CheckStyle
(static code analyzer)

Tokenizer

Warning messages

Parameter, ', isSelected, ', is, not, assigned, and, could, be, ...

Faulty corpus

Unknown

PMD or CheckStyle
(static code analyzer)

Tokenizer

Warning messages

Parameter, ', isSelected, ', is, not, assigned, and, could, be, ...

Probability to be faulty

PMD or CheckStyle
(static code analyzer)

Tokenizer

Warning messages

Parameter, ', isSelected, ', is, not, assigned, and, could, be, ...

Probability to be faulty

CRM114 Learner

Non-faulty corpus

CRM114 Classifier

Probability to be faulty
Recall rises rapidly and becomes stable earlier. But, lower precision makes overall result worse.
Analysis of Identifiers
What we did

We investigated the frequency of appearance of identifiers in source code modules from the viewpoint of the length of identifiers in Eclipse and NetBeans.

We modelled the relationship between the length of identifiers and the fault-proneness.

We used the random forest for the modelling.

Data used:

Frequency of identifier’s occurrence for each file grouped by the length of identifier + faulty status by SZZ

Kimiaki Kawamoto and Osamu Mizuno, "Do Long Identifiers Induce Faults in Software? --- a Repository Mining Based Investigation ---," In Proc. of 22nd International Symposium on Software Reliability Engineering (ISSRE2011), Supplemental proceedings, 3-1, November 2011. (Hiroshima, Japan)
Results of analysis

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Evaluation of prediction (average of 10 times)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
</tr>
<tr>
<td>Eclipse</td>
<td>0.888</td>
</tr>
<tr>
<td>Netbeans</td>
<td>0.850</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3</th>
<th>Top 10: Mean Decrease Gini (average of 10 times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>Length</td>
</tr>
<tr>
<td></td>
<td>Mean decrease Gini</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
</tr>
</tbody>
</table>

Lengths of 3, 4, and 7 have impact for fault-proneness.
[FYI] The longest identifiers

In Eclipse: 237 characters, appears 26 times

WorkingDirectoryStatusHandler_Eclipse_is_not_able_to_set_the_working_directory_specified_by_the_program_beingLauncher_as_the_current_runtime_does_not_support_working_directories__nContinue_launch_without_setting_the_working_directory__2

In Netbeans: 210 characters, appears 76 times

Do code review activities become more productive in Gerrit-based projects as a result of evolution?
Data collection tool (Developed)

Integrated crawling and mining tool [1].

Developed by Junwei Liang (a master course student)

Ratio of activities in review process

![Bar chart showing the ratio of activities in different projects.](chart.png)

- **Activity**
  - Create
  - Inline Comment
  - LGTM/Label
  - Message

- **Projects**
  - (Rietveld) Chromium
  - (Rietveld) GWT
  - (Gerrit) Android
  - (Gerrit) Qt

The ratio of Label activities increases in Gerrit-based projects.
Discussion time and approval

✦ In Rietveld, the number of LGTM is proportional to the discussion time.

✦ In Gerrit, +2 labeled issues quickly finish discussion.