Machine learning with Naive Bayes: MSR applications

Ralf Lämmel
Software Languages Team
Computer Science Faculty
University of Koblenz-Landau
Hidden agenda

• Motivate students to use machine learning in their MSR projects while using Naive Bayes as a simple baseline.

• Provide deeper understanding of some MSR projects including details of setting up Naive Bayes in nontrivial situations.

• Provide examples of how studies were using tool support such as Weka for machine learning in their projects.
Style of MSR-related RQs

• How to detect duplicate bug?
• How to detect blocking bugs?
• How to make static analysis (FindBugs) more accurate?
• How to detect causes of performance regression?
Naive Bayes
Towards a definition

In machine learning, naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features.

Source: https://en.wikipedia.org/wiki/Naive_Bayes_classifier
Bayes’ theorem

\[ P(A|B) = \frac{P(B|A) P(A)}{P(B)} , \]

where A and B are events.

• P(A) and P(B) are the probabilities of A and B without regard to each other.
• P(A | B), a conditional probability, is the probability of A given that B is true.
• P(B | A), is the probability of B given that A is true.

Source: https://en.wikipedia.org/wiki/Bayes%27_theorem
Bayes’ theorem

• What is Addison's probability of having cancer?
• „Prior“ probability (general population): 1 %
• „Posterior“ probability for a person of age 65:
  • Probability of being 65 years old: 0.2 %
  • Probability of person with cancer being 65: 0.5 %

Thus, a person (such as Addison) who is age 65 has a probability of having cancer equal to

\[
0.5\% \div 0.2\% \times 1\% = 2.5\%
\]

\[
P(A|B) = \frac{P(B|A)P(A)}{P(B)},
\]

Source: https://en.wikipedia.org/wiki/Bayes’_theorem
Multiple features $x_i$

Multiple classes $C_j$

The event for „age“ is a feature.
The event for „having cancer“ is a class.
In independent probability of a $C_k$ for all features:

$$p(C_k|x_1, \ldots, x_n) = \frac{1}{Z} p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

The maximum a posteriori or MAP decision rule:

$$\hat{y} = \arg\max_{k\in\{1,\ldots,K\}} p(C_k) \prod_{i=1}^{n} p(x_i|C_k).$$

Source: https://en.wikipedia.org/wiki/Naive_Bayes_classifier
Example: Fruit classification

<table>
<thead>
<tr>
<th>Type</th>
<th>Long</th>
<th>Not Long</th>
<th>Sweet</th>
<th>Not Sweet</th>
<th>Yellow</th>
<th>Not Yellow</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>400</td>
<td>100</td>
<td>350</td>
<td>150</td>
<td>450</td>
<td>50</td>
<td>500</td>
</tr>
<tr>
<td>Orange</td>
<td>0</td>
<td>300</td>
<td>150</td>
<td>150</td>
<td>300</td>
<td>0</td>
<td>300</td>
</tr>
<tr>
<td>Other Fruit</td>
<td>100</td>
<td>100</td>
<td>150</td>
<td>50</td>
<td>50</td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>500</td>
<td>500</td>
<td>650</td>
<td>350</td>
<td>800</td>
<td>200</td>
<td>1000</td>
</tr>
</tbody>
</table>

Prior probabilities:
- \( P(\text{Banana}) = 0.5 \) (500/1000)
- \( P(\text{Orange}) = 0.3 \)
- \( P(\text{Other Fruit}) = 0.2 \)

Evidence:
- \( P(\text{Long}) = 0.5 \)
- \( P(\text{Sweet}) = 0.65 \)
- \( P(\text{Yellow}) = 0.8 \)

Likelihood:
- \( P(\text{Long}/\text{Banana}) = 0.8 \)
- \( P(\text{Long}/\text{Orange}) = 0.0 \)
- ... 
- \( P(\text{Yellow}/\text{Other Fruit}) = 50/200 = 0.25 \)
- \( P(\text{Not Yellow}/\text{Other Fruit}) = 0.75 \)

Source: http://stackoverflow.com/questions/10059594/a-simple-explanation-of-naive-bayes-classification
Example: Fruit classification

A fruit is *Long, Sweet and Yellow.*

Is it a Banana? Is it an Orange? Or Is it some Other Fruit?

We compute all possible posterior probabilities and pick max.

\[
P(\text{Banana}/\text{Long, Sweet and Yellow})
\]

\[
\frac{P(\text{Long}/\text{Banana}) \cdot P(\text{Sweet}/\text{Banana}) \cdot P(\text{Yellow}/\text{Banana}) \cdot P(\text{banana})}{P(\text{Long}) \cdot P(\text{Sweet}) \cdot P(\text{Yellow})}
\]

\[
= \frac{0.8 \times 0.7 \times 0.9 \times 0.5}{P(\text{evidence})}
\]

\[
= 0.252 / P(\text{evidence})
\]

\[
P(\text{Orange}/\text{Long, Sweet and Yellow}) = 0
\]

\[
P(\text{Other Fruit}/\text{Long, Sweet and Yellow}) = 0.01875 / P(\text{evidence})
\]

Papers
New Features for Duplicate Bug Detection

Nathan Klein
Department of Computer Science
Oberlin College
Oberlin, Ohio, USA
nklein@oberlin.edu

Christopher S. Corley, Nicholas A. Kraft
Department of Computer Science
The University of Alabama
Tuscaloosa, Alabama, USA
cscorley@ua.edu, nkraft@cs.ua.edu
Example Bug Reports

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Bug 21196</th>
<th>Bug 20161</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Duplicate</td>
<td>Duplicate</td>
</tr>
<tr>
<td>MergeID</td>
<td>7402</td>
<td>7402</td>
</tr>
<tr>
<td>Summary</td>
<td>support urdu in android</td>
<td>urdu language support</td>
</tr>
<tr>
<td>Description</td>
<td>i just see many description where</td>
<td>hello i’m unable to read any type</td>
</tr>
<tr>
<td></td>
<td>people continuously requesting</td>
<td>of urdu language text messages.</td>
</tr>
<tr>
<td></td>
<td>google for support urdu in android</td>
<td>please add urdu language in future</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>updates of android ...</td>
</tr>
<tr>
<td>Component</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>Type</td>
<td>Defect</td>
<td>Defect</td>
</tr>
<tr>
<td>Priority</td>
<td>Medium</td>
<td>Medium</td>
</tr>
</tbody>
</table>

LDA is the probability that the classifier will rank a positive instance above a negative instance.

The ROC curve plots the true positive rate against its false positive rate as the classification threshold is varied.

The Kappa statistic is a measure of how closely the learned model fits the data.

Table 1: Attributes for Pairs of Bug Reports

Table 2: Example Bug Reports

Example Bug Reports

<table>
<thead>
<tr>
<th>Bug ID</th>
<th>Date</th>
<th>Summary</th>
<th>Description</th>
<th>Component</th>
<th>Type</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug 21196</td>
<td>Oct 25 2011 08:22:51</td>
<td>support urdu in android</td>
<td>i just see many description where people continuously requesting google for support urdu in android ...</td>
<td>Null</td>
<td>Defect</td>
<td>Medium</td>
</tr>
<tr>
<td>Bug 20161</td>
<td>Sep 19 2011 13:05:15</td>
<td>urdu language support</td>
<td>hello i’m unable to read any type of urdu language text messages. please add urdu language in future updates of android ...</td>
<td>Null</td>
<td>Defect</td>
<td>Medium</td>
</tr>
</tbody>
</table>
New features for duplicate bug detection

• The duplicate bug detection problem: given two bug reports, predict whether they are duplicates.

• Reports pulled from the android database between November 2007 and September 2012 with 1,452 bug reports marked as duplicates out of 37,627 total.

• Features of bug report: Bug ID, Date Opened, Status, Merge ID, Summary, Description, Component, Type, Priority, and Version. (Version is ignored because it is not used much.)

• Duplicate bug reports are placed in buckets resulting 2102 unique bug reports in the buckets.
New features for duplicate bug detection

• Calculated the topic-document distribution of each summary, description; combined summary and description for each report using the implementation of latent Dirichlet allocation (LDA) in MALLET with an alpha value of 50.0 and a beta value of 0.01 and a 100-topic model.

• Generated 20,000 pairs of bug reports consisting of 20% duplicate pairs while ensuring that no two pairs contained identical reports.

• Computed 13 attributes for each pair; used the Porter stemmer to stem words for the simSumControl and simDesControl attributes; used the SEO suite stopword; LDA distributions are sorted based on the percentage each topic describes, in decreasing order.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lenWordDiffSum</td>
<td>Difference in the number of words in the summaries or descriptions</td>
</tr>
<tr>
<td>lenWordDiffDes</td>
<td></td>
</tr>
<tr>
<td>simSumControl</td>
<td>Number of shared words in the summaries or descriptions after stemming and stop-word removal, controlled by their lengths</td>
</tr>
<tr>
<td>simDesControl</td>
<td></td>
</tr>
<tr>
<td>sameTopicSum</td>
<td>First shared identical topic between the sorted distribution given by LDA to each summary, description, or combined summary and description</td>
</tr>
<tr>
<td>sameTopicDes</td>
<td></td>
</tr>
<tr>
<td>sameTopicTot</td>
<td></td>
</tr>
<tr>
<td>topicSimSum</td>
<td>Hellinger distance between the topic distributions given by LDA to each summary, description, or combined summary and description</td>
</tr>
<tr>
<td>topicSimDes</td>
<td></td>
</tr>
<tr>
<td>topicSimTot</td>
<td></td>
</tr>
<tr>
<td>priorityDiff</td>
<td>{same-priority, not-same}</td>
</tr>
<tr>
<td>timeDiff</td>
<td>Difference in minutes between the times the bugs were submitted</td>
</tr>
<tr>
<td>sameComponent</td>
<td>Four-category attribute: {both-null, one-null, no-null-same, no-null-not-same}</td>
</tr>
<tr>
<td>sameType</td>
<td>{same-type, not-same}</td>
</tr>
<tr>
<td>class</td>
<td>{dup, not-dup}</td>
</tr>
</tbody>
</table>
New features for duplicate bug detection

- Tested the predictive power of a range of machine learning classifiers using the Weka tool. Tests were conducted using **ten-fold crossvalidation**.

- Tested the efficacy of a machine learner using its accuracy, the AUC, or area under the Receiver Operating Characteristic (ROC) curve, and its Kappa statistic. The ROC curve plots the true positive rate of a binary classifier against its false positive rate as the threshold of discrimination changes, and therefore the **AUC is the probability that the classifier will rank a positive instance higher than a negative instance.** The Kappa statistic is a measure of how closely the learned model fits the data given. In this model, it signifies how closely the learned model corresponds to the triagers which classified the bug reports.
Machine learners used

- ZeroR
- Naive Bayes
- Logistic Regression
- C4.5
- K-NN
- REPTree with Bagging
Because our metrics showed a significant improvement over both those used by Alipour et al. and Sun et al., further comparisons are presented for each incoming report, would further improve duplicate bug detection.

Table 4 lists the percentage change over Alipour et al.'s metrics, while Table 3 details the accuracy, AUC, and the Kappa statistic for each of the five classifiers we used. Table 5 lists the percentage change of our accuracy, AUC, and kappa values obtained by our study and a negative value from Sun et al.'s metrics.

### 4. THREATS TO VALIDITY

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A high number of bugs marked as duplicate in the Android database makes them good candidates for futures studies that may be a simple method with which to supplement or replace tf-idf methods in some text retrieval applications.

### 5. CONCLUSION

The effectiveness and simplicity of the metrics proposed by Alipour et al., when combined with ours, could further improve state-of-the-art top-k approaches to bug deduplication. The practices of splitting the sum of the attributes we have proposed. We have shown that they provide an increase in accuracy, AUC, and Kappa statistics over the metrics of Alipour et al. [2] and Sun et al.

### 6. ACKNOWLEDGMENTS

We thank Alipour et al. for sharing the Android bug data used in our study simply extends the analysis by Alipour et al.: a strategy used by Jalbert and Weimar [6], and of using the novel distance measure) are particularly recommended. Finally, the use of Bagging to aid in classification should be considered, as it provided a small increase in accuracy.

### Table 4: Improvement over Alipour et al.'s metrics

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy %</th>
<th>AUC</th>
<th>Kappa</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeroR</td>
<td>80.00%</td>
<td>0.500</td>
<td>0.00</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>92.990%</td>
<td>0.958</td>
<td>0.778</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>94.585%</td>
<td>0.972</td>
<td>0.824</td>
</tr>
<tr>
<td>C4.5</td>
<td>94.780%</td>
<td>0.941</td>
<td>0.832</td>
</tr>
<tr>
<td>K-NN</td>
<td>94.785</td>
<td>0.955</td>
<td>0.830</td>
</tr>
<tr>
<td>Bagging: REPTree</td>
<td>95.170%</td>
<td>0.977</td>
<td>0.845</td>
</tr>
</tbody>
</table>

## Classification results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy %</th>
<th>AUC</th>
<th>Kappa</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeroR</td>
<td>80.00%</td>
<td>0.500</td>
<td>0.00</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>92.990%</td>
<td>0.958</td>
<td>0.778</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>94.585%</td>
<td>0.972</td>
<td>0.824</td>
</tr>
<tr>
<td>C4.5</td>
<td>94.780%</td>
<td>0.941</td>
<td>0.832</td>
</tr>
<tr>
<td>K-NN</td>
<td>94.785</td>
<td>0.955</td>
<td>0.830</td>
</tr>
<tr>
<td>Bagging: REPTree</td>
<td>95.170%</td>
<td>0.977</td>
<td>0.845</td>
</tr>
</tbody>
</table>
Hellinger distance. Therefore, we hypothesize that this measure is significantly higher than that of metrics using Weka. This is detailed in Table 6.

We then evaluated the information gain of each metric, as they provide an increase in accuracy, AUC, and Kappa statistic for each of the five classifiers we used. Table 4 lists the percentage change of our accuracy, AUC, and the Kappa statistic for each of the five classifiers we obtained by Alipour et al. Table 3 details the accuracy, AUC, and the Kappa statistic for each of the five classifiers we used in their study. This material is based upon work supported by the National Science Foundation under Grant No. 1156563.

6. ACKNOWLEDGMENTS

3. CASE STUDY

The results of this paper demonstrate the descriptive power of the metrics proposed. Our results suggest that this suite of metrics could significantly improve state-of-the-art duplicate bug detection. In addition, some 10,000 reports in this time period are represented by large buckets of duplicates, there is a threat to the validity of the study. Because more repositories are duplicates or invalid [3], yet of the reports in the data, only 1,452 of 37,627 reports were marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

Table 4: Improvement over Alipour et al. [2]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy %</th>
<th>AUC</th>
<th>Kappa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive Bayes</td>
<td>+12.13%</td>
<td>0.256</td>
<td>0.00</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>+14.19%</td>
<td>0.215</td>
<td>0.00</td>
</tr>
<tr>
<td>C4.5</td>
<td>+15.06%</td>
<td>0.312</td>
<td>0.00</td>
</tr>
<tr>
<td>K-NN</td>
<td>+16.741%</td>
<td>0.344</td>
<td>0.00</td>
</tr>
<tr>
<td>ZeroR</td>
<td>0%</td>
<td>0.500</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 5: Improvement over Sun et al. [11]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy %</th>
<th>AUC</th>
<th>Kappa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive Bayes</td>
<td>+10%</td>
<td>0.252</td>
<td>0.00</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>+11.76%</td>
<td>0.170</td>
<td>0.00</td>
</tr>
<tr>
<td>C4.5</td>
<td>+12.59%</td>
<td>0.203</td>
<td>0.00</td>
</tr>
<tr>
<td>K-NN</td>
<td>+15.06%</td>
<td>0.209</td>
<td>0.00</td>
</tr>
<tr>
<td>ZeroR</td>
<td>0%</td>
<td>0.109</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 3: Classification Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Accuracy %</th>
<th>AUC</th>
<th>Kappa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive Bayes</td>
<td>94.780%</td>
<td>0.958</td>
<td>0.845</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>94.585%</td>
<td>0.941</td>
<td>0.824</td>
</tr>
<tr>
<td>C4.5</td>
<td>92.990%</td>
<td>0.972</td>
<td>0.778</td>
</tr>
<tr>
<td>K-NN</td>
<td>80.00%</td>
<td>0.500</td>
<td>0.00</td>
</tr>
<tr>
<td>ZeroR</td>
<td>0%</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

The information gain of the metrics using the first shared topic distance measure (which is, we believe, a novel distance measure) are particularly recommended. Finally, the use of Bagging to aid in classification should be considered, as it provided a small increase in accuracy. Many of these could actually be duplicates. A threat to internal validity concerns the selection of pairs. Because more duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The results of this paper demonstrate the descriptive power of the metrics proposed. Our results suggest that this suite of metrics could significantly improve state-of-the-art duplicate bug detection. In addition, some 10,000 reports in this time period are represented by large buckets of duplicates, there is a threat to the validity of the study. Because more repositories are duplicates or invalid [3], yet of the reports in the data, only 1,452 of 37,627 reports were marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.

The primary threat to construct validity concerns the number of bugs marked as duplicate in the Android database. A future study which utilized our metrics in order to find duplicates of incoming bugs, using a top-k approach similar to Sun et al. in which a fixed number of possible duplicate reports are presented for each incoming report, would further prove duplicate bug detection. Alipour et al., when combined with ours, could further improve their effectiveness and simplicity of the metrics proposed.
Characterizing and Predicting Blocking Bugs in Open Source Projects

Harold Valdivia Garcia and Emad Shihab
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
{hv1710, emad.shihab}@rit.edu
Characterizing and Predicting Blocking Bugs

• Normal flow: Someone discovers a bug and creates the respective bug report, then the bug is assigned to a developer who is responsible for fixing it and finally, once it is resolved, another developer verifies the fix and closes the bug report.

• Blocking bugs: The fixing process is stalled because of the presence of a blocking bug. Blocking bugs are software defects that prevent other defects from being fixed.

  • Blocking bugs lengthen the overall fixing time of the software bugs and increase the maintenance cost.

  • Blocking bugs take longer to be fixed compared to non-blocked bugs.

  • To reduce the impact of blocking bugs, prediction models are build in order to flag the blocking bugs early on for developers.
RQ1 Can we build highly accurate models to predict whether a new bug will be a blocking bug?

RQ2 Which factors are the best indicators of blocking bugs?
Characterizing and Predicting Blocking Bugs

Data use in the study: Chromium, Eclipse, FreeDesktop, Mozilla, NetBeans and OpenOffice

---

Chromium extraction, we filtered out those bugs with empty fields. Were verified or closed before blocking bugs and 38,695 were non-blocking bugs. Instances with non-empty values. After this preprocessing step, the fields were also filtered out, because our prediction models require the total number of extracted bugs was 206,125 bugs. We removed Chromium, we extracted all bugs published before long-lived open sources projects, with a large amount of bug re-

NetBeans is also another popular IDE written in Java. Al-

hosts and develops products such as Firefox, Thunderbird, Bugzilla, etc. Mozilla is a framework and umbrella project that Window System), Mesa (free implementation of the OpenGL spec-

FreeDesktop is an umbrella project well known for its system of plugins that allows customization of Chrome. Eclipse is a popular multi-language IDE written in Java, developed by Google and used as the development branch of Google sizes blocking bugs. Section 4 presents our case study. We compare the approach used in this work. Section 3 discusses and character-

In order to perform our study, we used the bug reports from six different projects namely: Chromium, Eclipse, FreeDesktop, Chrome, Editor, UI, etc). Some components are more/less critical than others. For example, it might be the case that bugs in critical Core, Editor, UI, etc). Some components are more/less critical than in Mozilla (30 bugs).

The only two exceptions were FreeDesktop and Mozilla with 8.9% of the bugs are blocking bugs. Section 2.2 Factors Used to Predict Blocking Bugs and 12.5%, respectively. The percentages of blocking bugs was less that 3% of the set of bugs. The total number of valid bugs for these five projects was 363,343. Since our goal is to be able to predict blocking bugs, we extracted factors best identify blocking bugs. In particular, we would like to determine which factors are the best indicators of blocking bugs. We consider 14 different factors from the bug reports so the blocking bugs can be detected early on. In addition, we would like to determine which factors best identify these blocking bugs. We use 14 different factors extracted from bug databases to classify a new bug will be a blocking bug? Our models achieve F-measure values between 15%-42%.

We find that the bug comments, the number of developers who reported the bug. We determine the ground truth (i.e., infor-

The approach used in this work. Section 3 discusses and character-

1. RQ1:

Which factors are the best indicators of blocking bugs?

2. RQ2:

Which factors best identify blocking bugs? In particular, we would like to find blocking/non-blocking bugs for specific platforms. Note that we were not able to have this factor for Chromium because its issue tracking system does not support it.

We use 14 different factors extracted from bug databases to determine whether a bug is blocking or non-blocking ) from the bug factors extracted from the bugs reports. Second, we describe the factors best extract from the bugs reports. After this preprocessing step, the fields were also filtered out, because our prediction models require prediction models and the performance metrics used in our study. First, we discuss the data used in our case study and we list the factors extracted from the bugs reports. Second, we describe the factors best identify blocking bugs. In particular, we would like to answer the following research questions:

RQ1:

Which factors are the best indicators of blocking bugs?

RQ2:

Which factors best identify these blocking bugs. We consider 14 different factors to help us discriminate between blocking and non-blocking bugs. We consider 14 different factors from the bug reports so the blocking bugs can be detected early on. In addition, we would like to determine which factors best identify these blocking bugs. We use 14 different factors extracted from bug databases to classify a new bug will be a blocking bug? Our models achieve F-measure values between 15%-42%.
Factors Used to Predict Blocking Bugs

- Product
- Component
- Platform
- Severity
- Priority
- Number in the CC list
- Description size
- Description text
- Comment size
- Comment text
- Priority has increased
- Reporter name
- Reporter experience
- Reporter blocking experience
Converting textual factor into Bayesian-score

- Two data sets based on stratified sampling to avoid bias of classifiers
- Corpus\textsubscript{1} comment/description for blocking bugs
- Corpus\textsubscript{0} comment/description for nonblocking bugs
- Compute probability of a word to be indicator of blocking bug
- Bayesian score simply combines all word-level probabilities
- Filtered out all the words with less than five occurrences in the corpora.
- Bayesian-score of a description/comment is based on the combined probability of the fifteen most important words of the description/comment.

---

**Conversion Process:**

1. **First Training Set**
   - Corpus\textsubscript{1} (blocking)
   - Corpus\textsubscript{0} (nonblocking)
   - Word Frequency Tables
   - Training the classifier
   - Naïve Bayes Classifier
   - Apply on Bayesian Score

2. **Second Training Set**
   - Corpus\textsubscript{1} (blocking)
   - Corpus\textsubscript{0} (nonblocking)
   - Word Frequency Tables
   - Training the classifier
   - Naïve Bayes Classifier
   - Apply on Bayesian Score
Prediction models

- Primary: Decision tree classifier
- Also:
  - Naive Bayes
  - kNN
  - Random Forests
  - Zero-R
Confusion matrix: true/false positive/negatives

<table>
<thead>
<tr>
<th>Classified as</th>
<th>True class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Blocking</td>
</tr>
<tr>
<td>Blocking</td>
<td>TP</td>
</tr>
<tr>
<td>Non-blocking</td>
<td>FN</td>
</tr>
</tbody>
</table>

Table 2: Confusion matrix:

<table>
<thead>
<tr>
<th>True class</th>
<th>Blocking</th>
<th>Non-blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking</td>
<td>TP</td>
<td>FP</td>
</tr>
<tr>
<td>Non-blocking</td>
<td>FN</td>
<td>TN</td>
</tr>
</tbody>
</table>
Performance evaluation

1. **Precision:** The ratio of correctly classified blocking bugs over all the bugs classified as blocking. It is calculated as $Pr = \frac{TP}{TP+FP}$.

2. **Recall:** The ratio of correctly classified blocking bugs over all of the actually blocking bugs. It is calculated as $Re = \frac{TP}{TP+FN}$.

3. **F-measure:** Measures the weighted harmonic mean of the precision and recall. It is calculated as $F-measure = \frac{2 \times Pr \times Re}{Pr + Re}$.

4. **Accuracy:** The ratio between the number of correctly classified bugs (both the blocking and the non-blocking) over the total number of bugs. It is calculated as $Acc = \frac{TP+TN}{TP+FP+TN+FN}$.

A blocking precision value of 100% would indicate that every bug we classified as blocking bug was actually a blocking bug. A blocking recall value of 100% would indicate that every actual blocking bug was classified as blocking bug.
Estimation of accuracy by 10-fold cross-validation

• Split data set

• 10 parts of same size

• Preserve original distribution of classes

• Perform $i = 1, \ldots, 10$ iterations (folds)

• Use all but $i$-th part for training

• Use $i$-th part for testing

• Report average performance for the folds
Predictions different algorithms

<table>
<thead>
<tr>
<th>Project</th>
<th>Classif.</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
<th>Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium</td>
<td>Zero-R</td>
<td>NA</td>
<td>0%</td>
<td>0%</td>
<td>97.6%</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>10.0%</td>
<td>54.2%</td>
<td>16.9%</td>
<td>81.5%</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>9.0%</td>
<td>47.1%</td>
<td>15.1%</td>
<td>81.5%</td>
</tr>
<tr>
<td></td>
<td>Rand. Forest</td>
<td><strong>18.6%</strong></td>
<td>29.6%</td>
<td><strong>22.8%</strong></td>
<td>93.0%</td>
</tr>
<tr>
<td></td>
<td>C4.5</td>
<td>9.1%</td>
<td><strong>49.9%</strong></td>
<td>15.3%</td>
<td>80.7%</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Zero-R</td>
<td>NA</td>
<td>0%</td>
<td>0%</td>
<td>97.2%</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>8.8%</td>
<td><strong>66.4%</strong></td>
<td>15.5%</td>
<td>79.7%</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>8.1%</td>
<td>53.0%</td>
<td>14.0%</td>
<td>81.8%</td>
</tr>
<tr>
<td></td>
<td>Rand. Forest</td>
<td><strong>16.5%</strong></td>
<td>24.0%</td>
<td><strong>19.5%</strong></td>
<td>94.5%</td>
</tr>
<tr>
<td></td>
<td>C4.5</td>
<td>9.2%</td>
<td>47.0%</td>
<td>15.4%</td>
<td>85.5%</td>
</tr>
<tr>
<td>FreeDesktop</td>
<td>Zero-R</td>
<td>NA</td>
<td>0%</td>
<td>0%</td>
<td>91.1%</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>18.7%</td>
<td><strong>74.3%</strong></td>
<td>29.9%</td>
<td>69.0%</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>19.4%</td>
<td>72.6%</td>
<td>30.6%</td>
<td>70.7%</td>
</tr>
<tr>
<td></td>
<td>Rand. Forest</td>
<td><strong>27.8%</strong></td>
<td>60.2%</td>
<td><strong>37.9%</strong></td>
<td>82.4%</td>
</tr>
<tr>
<td></td>
<td>C4.5</td>
<td>20.4%</td>
<td>73.6%</td>
<td>31.9%</td>
<td>72.0%</td>
</tr>
<tr>
<td>Mozilla</td>
<td>Zero-R</td>
<td>NA</td>
<td>0%</td>
<td>0%</td>
<td>87.4%</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>29.5%</td>
<td>68.0%</td>
<td>41.1%</td>
<td>75.6%</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>23.3%</td>
<td>69.3%</td>
<td>34.9%</td>
<td>67.5%</td>
</tr>
<tr>
<td></td>
<td>Rand. Forest</td>
<td><strong>36.1%</strong></td>
<td>53.6%</td>
<td><strong>43.2%</strong></td>
<td>82.3%</td>
</tr>
<tr>
<td></td>
<td>C4.5</td>
<td>29.0%</td>
<td><strong>76.7%</strong></td>
<td>42.1%</td>
<td>73.6%</td>
</tr>
<tr>
<td>NetBeans</td>
<td>Zero-R</td>
<td>NA</td>
<td>0%</td>
<td>0%</td>
<td>96.8%</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>9.9%</td>
<td><strong>73.3%</strong></td>
<td>17.3%</td>
<td>77.1%</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>11.4%</td>
<td>59.3%</td>
<td>19.1%</td>
<td>84.2%</td>
</tr>
<tr>
<td></td>
<td>Rand. Forest</td>
<td><strong>26.3%</strong></td>
<td>37.6%</td>
<td><strong>30.9%</strong></td>
<td>94.7%</td>
</tr>
<tr>
<td></td>
<td>C4.5</td>
<td>12.8%</td>
<td>59.3%</td>
<td>21.1%</td>
<td>86.0%</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>Zero-R</td>
<td>NA</td>
<td>0%</td>
<td>0%</td>
<td>96.9%</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>12.9%</td>
<td><strong>78.1%</strong></td>
<td>22.1%</td>
<td>83.3%</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>14.2%</td>
<td>66.1%</td>
<td>23.4%</td>
<td>87.0%</td>
</tr>
<tr>
<td></td>
<td>Rand. Forest</td>
<td><strong>32.9%</strong></td>
<td>46.7%</td>
<td><strong>38.6%</strong></td>
<td>95.5%</td>
</tr>
<tr>
<td></td>
<td>C4.5</td>
<td>15.9%</td>
<td>65.9%</td>
<td>25.6%</td>
<td>88.4%</td>
</tr>
</tbody>
</table>
Other measures

- Time to identify a blocking bug
- Degree of blockiness
RQ1 Can we build highly accurate models to predict whether a new bug will be a blocking bug?
We use 14 different factors extracted from bug databases to build accurate prediction models that predict whether a bug will be a blocking bug or not. Our models achieve F-measure values between 15%-42%.

RQ2 Which factors are the best indicators of blocking bugs?
We find that the bug comments, the number of developers in the CC list and the bug reporter are the best indicators of whether or not a bug will be blocking bug.
Finding Patterns in Static Analysis Alerts

Improving Actionable Alert Ranking

Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam
University of Waterloo
200 University Ave W
Waterloo, Ontario
qhanam,lintan,rtholmes,patrick.lam@uwaterloo.ca
Finding Patterns in Static Analysis Alerts

• Static analysis (SA) tools find bugs by inferring programmer beliefs

• Tools such as FindBugs are commonplace in industry

• SA tools find a large number of actual defects

• Problem: high rates of alerts that a developer would not act on (unactionable alerts) because they are incorrect, do not significantly affect program execution, etc. High rates of unactionable alerts decrease the utility of SA tools in practice.
Finding Patterns in Static Analysis Alerts

```java
1 static final SimpleDateFormat cDateFormat = new SimpleDateFormat("yyyy-MM-dd");
```

The code defines the member variable `cDateFormat`.

Running FindBugs with this code results in the following alert:

“STCAL: Sharing a single instance across thread boundaries without proper synchronization will result in erratic behaviour of the application”.

The alert is correct and this statement could potentially result in a concurrency error. However, in practice this `SimpleDateFormat` object is never written to beyond its construction. As long as this is the case, there is no need to provide synchronized access to the object.
Finding Patterns in Static Analysis Alerts

```java
1  public int read(byte[] b, int offset, int len)
2  {
3      if (log.isTraceEnabled()) {
4          log.trace("read() " + b + " "
5              + (b==null ? 0: b.length)
6              + " " + offset + " " + len);
7      }
8  }
9  ...
```

The code reads a message in the form of a byte array. Running FindBugs results in the following alert: “USELESS STRING: This code invokes toString on an array, which will generate a fairly useless result such as [C@16f0472.”.

Indeed, the toString method is being called on byte array b, which emits a memory address. However, the behaviour may be intentional — the output of b.toString() appears in logging code, where it might be useful to disambiguate arrays.

In fact, any call to toString on an array within log.trace() is likely to be an unactionable alert. We can automatically identify this unactionable alert pattern by looking for calls to toString on an array inside the method log.trace(). There are 29 occurrences of this unactionable alert pattern in Tomcat6 r1497967.
Finding Patterns in Static Analysis Alerts

The code closes Socket and ObjectReader objects. Running FindBugs on this code results in the following alert on lines 2 and 4: “This method might ignore an exception.”

This is an unactionable alert. Since both resources socket and reader are being closed, the program is clearly done using them. One can easily see that if either are null or there is an error while closing the resources, the program can ignore the exception and assume the connection is closed with only minor consequences if the connection fails to close (i.e. trying to determine what went wrong is not worth the developer’s effort in this situation). We can automatically identify this unactionable alert pattern by finding calls to Socket.close() or ObjectReader.close() within the preceding try statement of the offending catch block.
Finding Patterns in Static Analysis Alerts

- TP, FP, TN, FN — not applicable
- Use AA (actionable alert) and UA (unactionable alert)
- SA tools use patterns for alert generation
- Tool developers must strive to minimize UA
- Discovering and adding new UA patterns is time consuming
- Thus, add machine learning with alert characteristics (AC)
  - What are the different patterns for AA and UA?
  - SA tool results are prioritized then.
    - Compare new alert with previous alert patterns
Finding Patterns in Static Analysis Alerts

**Research Question 1** Do SA alert patterns exist?

**Research Question 2** Can we use SA alert patterns to improve actionable alert ranking over previous techniques?
1. We calculate a set of related statements by slicing the program at the site of the alert.

2. Using the statements from step 1 and the class hierarchy for the subject program, we extract a set of ACs.

3. A machine learning algorithm pre-computes a model with which to classify new alerts as actionable or unactionable. The model is trained using previously classified alerts. Alerts are classified by the developer or inferred by the version history.

4. Using the model from step 3, the machine learning algorithm ranks each alert, with those more likely to be actionable at the top.
Generate alert slices

- Use the statements flagged with alerts by SA as seeds for (limited!) backward program slice construction.

- A backwards program slice takes a statement in source code (called a seed statement) and determines which statements could have affected the outcome of the seed statement.
Extract alert characteristics

<table>
<thead>
<tr>
<th>Seed Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statement Type</strong></td>
</tr>
<tr>
<td>Call</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>New</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Binary Operation</td>
</tr>
<tr>
<td>Field Access</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Catch</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-Seed Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statement Type</strong></td>
</tr>
<tr>
<td>Field</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

- **Call Name** — The name of the method being called.
- **Call Class** — The name of the class containing the method being called.
- **Call Parameter Signature** — The signature for the method parameters.
- **Return Type** — The signature for the method’s return type.
- **New Type** — The class of the object being created.
- **Concrete Type** — The class of the concrete type of the object being created.
- **Operator** — The operator for the binary operation.
- **Field Access Class** — The class containing the field being accessed.
- **Field Access Field** — The name of the field being accessed.
- **Catch** — Indicates that a catch statement is present.

| Alert ID | [Statement 1 Features] | [Statement 2 Features] | ... | [Statement D Features] |
Speeding up analysis

• Call graphs and points-to analysis are expensive.

• Computation of program slices, too.

• Call graph and slice size are limited:
  • Assumption: code-clone patterns occur close to alert
  • Thin slicing (to limit statements in slice)
  • Only 5 statements prior to seed
Metrics

- **Percent of actionable alerts found:** how many actionable alerts a developer would see if she inspected the top N% of alerts in a ranked list.

- Precision

- Recall

- F-measure

Given a set of ranked alerts $R$, a set of actionable alerts $A$ (where $A \subseteq R$) and integer $N$ where $0 \leq N \leq 100$, let $\%AA_N$ be the percent of actionable alerts found if we inspect the top $N\%$ of alerts in $R$. To get $\%AA_N$, we select the top $N\%$ of alerts in $R$ and call this set $R_N$. We then extract all actionable alerts from $R_N$ into a new set called $R_{NA}$. $\%AA_N$ is then $\frac{|R_{NA}|}{|A|} * 100$. For example, consider a situation where $A$ contains 10 actionable alerts ($|A| = 10$) and $R$ contains 200 alerts ($|R| = 200$). If $N=10$ then we inspect 20 alerts ($|R_{10}| = 20$). If there are five actionable alerts within $R_{10}$ ($|R_{10A}| = 5$), then $\%AA_N = \frac{5}{10} * 100 = 50\%$. This formula is shown below.

$$\%AA_N = \frac{|R_{NA}|}{|A|} * 100$$
• Use the source code history of a project to determine if alerts are actionable or unactionable.

• 1. Select a number of revisions across a subject project’s history.

• 2. Run a static analysis tool (FindBugs) on each revision to generate a list of alerts for each revision.

• 3. **Find alerts that are closed** over the course of the project history:
   • An alert is opened in the first revision it appears.
   • An alert is closed in the first revision after the open revision where the alert is not present (except in the case where it is not present because the file containing it is deleted).

• 4. **Alerts that are closed are classified as actionable, while alerts that are open following the last revision analysed are classified as unactionable.**
Bugs) ACs and JavaNCSS ACs, our technique dis-
tool to enhance alert ranking. FindBugs priority ranking and that it could be an e
This result shows that our technique by itself out-performs
57 AAs in the top 5% of alerts and Baseline 1 discovers 19.
Across all three subject programs, statement ACs discover
Bugs priority ranking discovers 10% (or 15/153) of all AAs.
ment ACs discover 33% (or 51/153) of all AAs while Find-
example, observe the graph for Tomcat6. When x=5, state-
FindBugs priority ranking is labelled
shows the value of N (% of warnings inspected). Our tech-
shows the %
programs.
our method discovers 38 more AAs than Baseline 1
warning as unactionable). For RQ2, we rank alerts by the
We answer RQ2 in two parts: First we compare our method
Because there is a high number of UAs compared to AAs, it

<table>
<thead>
<tr>
<th>Subject</th>
<th>%AA_N, N =</th>
<th>Unactionable</th>
<th>Actionable</th>
<th>Weighted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 20 30</td>
<td>Precision</td>
<td>Recall</td>
<td>F</td>
</tr>
<tr>
<td>Tomcat6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADTree</td>
<td>0.36 0.42 0.52</td>
<td>0.96 1.00</td>
<td>0.96</td>
<td>1.00</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.40 0.49 0.61</td>
<td>0.93 0.93</td>
<td>0.93</td>
<td>0.38</td>
</tr>
<tr>
<td>BayesNet</td>
<td>0.33 0.51 0.60</td>
<td>0.93 0.92</td>
<td>0.93</td>
<td>0.38</td>
</tr>
<tr>
<td>Commons</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADTree</td>
<td>0.13 0.29 0.49</td>
<td>0.75 0.73</td>
<td>0.75</td>
<td>0.53</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.24 0.49 0.64</td>
<td>0.60 0.47</td>
<td>0.60</td>
<td>0.45</td>
</tr>
<tr>
<td>BayesNet</td>
<td>0.18 0.42 0.58</td>
<td>0.80 0.81</td>
<td>0.80</td>
<td>0.62</td>
</tr>
<tr>
<td>Logging</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADTree</td>
<td>0.31 0.46 0.46</td>
<td>0.95 1.00</td>
<td>0.95</td>
<td>0.00</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.23 0.46 0.54</td>
<td>0.59 0.43</td>
<td>0.59</td>
<td>0.10</td>
</tr>
<tr>
<td>BayesNet</td>
<td>0.15 0.38 0.54</td>
<td>0.89 0.87</td>
<td>0.89</td>
<td>0.11</td>
</tr>
</tbody>
</table>
Baseline

- Baseline 1: Order returned by FindBugs
- Baseline 2: Other approaches for finding AA or ranking alerts
An Industrial Case Study of Automatically Identifying Performance Regression-Causes

Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan
Queen’s University, Kingston, Ontario, Canada
{thanhnguyen, mei, ahmed}@cs.queensu.ca

Mohamed Nasser, Parminder Flora
Performance Engineering, Blackberry, Canada
Improving the Accuracy of Duplicate Bug Report Detection using Textual Similarity Measures

Alina Lazar, Sarah Ritchey, Bonita Sharif
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio USA 44555
alazar@ysu.edu, sritchey@student.ysu.edu, bsharif@ysu.edu
Towards Building a Universal Defect Prediction Model

Feng Zhang  
School of Computing  
Queen’s University  
Kingston, Ontario, Canada  
feng@cs.queensu.ca

Iman Keivanloo  
Department of Electrical and Computer Engineering  
Queen’s University  
Kingston, Ontario, Canada  
iman.keivanloo@queensu.ca

Audris Mockus  
Department of Software Engineering  
Avaya Labs Research  
Basking Ridge, NJ 07920, USA  
audris@avaya.com

Ying Zou  
Department of Electrical and Computer Engineering  
Queen’s University  
Kingston, Ontario, Canada  
ying.zou@queensu.ca
End of lecture

• Quick round: How could you use Naive Bayes et al. in your project?