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

$$0.5\% \div 0.2\% * 1\% = 2.5\%$$

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

Source: https://en.wikipedia.org/wiki/Bayes%27_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 Ck for all features:

Posterior

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

Prior

The maximum a posteriori or MAP decision rule:

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

Source: https://en.wikipedia.org/wiki/Naive_Bayes_classifier

Example: Fruit classification

Type	Long	Not Long	Sweet	Not Sweet	Yellow	Not Yellow	Total
Banana	400	100	350	150	450	50	500
Orange	0	300	150	150	300	0	300
Other Fruit	100	100	150	50	50	150	200
Total	500	500	650	350	800	200	1000

Prior probabilities:

- P(Banana) = 0.5 (500/1000)
- P(Orange) = 0.3
- P(Other Fruit) = 0.2

Evidence:

- P(Long) = 0.5
- P(Sweet) = 0.65
- P(Yellow) = 0.8

Likelihood:

- P(Long/Banana) = 0.8
- P(Long/Orange) = 0.0
- …
- P(Yellow/Other Fruit) = 50/200 = 0.25
- P(Not Yellow/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.

$$P(\text{Banana/Long, Sweet and Yellow}) = \frac{P(\text{Long/Banana})\,P(\text{Sweet/Banana})\,P(\text{Yellow/Banana})\,P(\text{banana})}{P(\text{Long})\,P(\text{Sweet})\,P(\text{Yellow})}$$

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

$= 0.252 / P(\text{evidence})$

P(Orange/Long, Sweet and Yellow) = 0

P(Other Fruit/Long, Sweet and Yellow) = 0.01875/P(evidence)

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

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

MSR
2014

Example Bug Reports

Attribute	Bug 21196	Bug 20161
Submitted	oct 25 2011 08:22:51	sep 19 2011 13:05:15
Status	Duplicate	Duplicate
MergeID	7402	7402
Summary	support urdu in android	urdu language support
Description	i just see many description where people continuously requesting google for support urdu in andriod ...	hello i'm unable to read any type of urdu language text messages. please add urdu language in future updates of android ...
Component	Null	Null
Type	Defect	Defect
Priority	Medium	Medium

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.

Attributes for Pairs of Bug Reports

lenWordDiffSum *lenWordDiffDes*	Difference in the number of words in the summaries or descriptions
simSumControl *simDesControl*	Number of shared words in the summaries or descriptions after stemming and stop-word removal, controlled by their lengths
sameTopicSum *sameTopicDes* *sameTopicTot*	First shared identical topic between the sorted distribution given by LDA to each summary, description, or combined summary and description
topicSimSum *topicSimDes* *topicSimTot*	Hellinger distance between the topic distributions given by LDA to each summary, description, or combined summary and description
priorityDiff	{same-priority, not-same}
timeDiff	Difference in minutes between the times the bugs were submitted
sameComponent	Four-category attribute: {both-null, one-null, no-null-same, no-null-not-same}
sameType	{same-type, not-same}
class	{dup, not-dup}

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

Classification results

Algorithm	Accuracy %	AUC	Kappa
ZeroR	80.00%	0.500	0.00
Naive Bayes	92.990%	0.958	0.778
Logistic Regression	94.585%	0.972	0.824
C4.5	94.780%	0.941	0.832
K-NN	94.785	0.955	0.830
Bagging: REPTree	**95.170%**	**0.977**	**0.845**

Gain of metrics

$sameTopicSum$	0.330
$sameTopicTot$	0.321
$topicSimSum$	0.256
$simSumControl$	0.252
$topicSimTot$	0.209
$sameTopicDes$	0.203
$topicSimDes$	0.170
$simDesControl$	0.109

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.

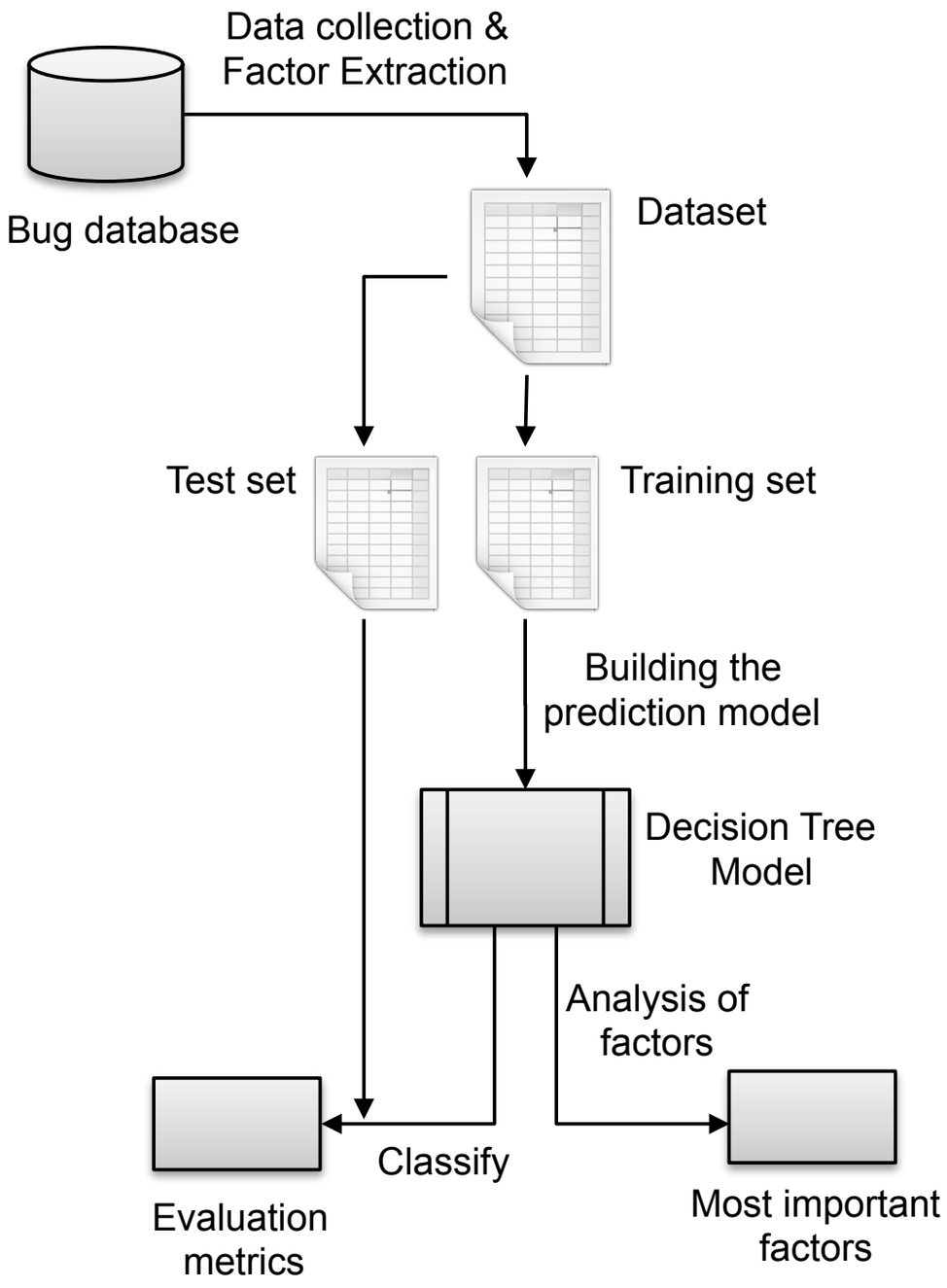
  - **To reduce the impact of blocking bugs, prediction models are build in order to flag the blocking bugs early on for developers.**

Characterizing and Predicting Blocking Bugs

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

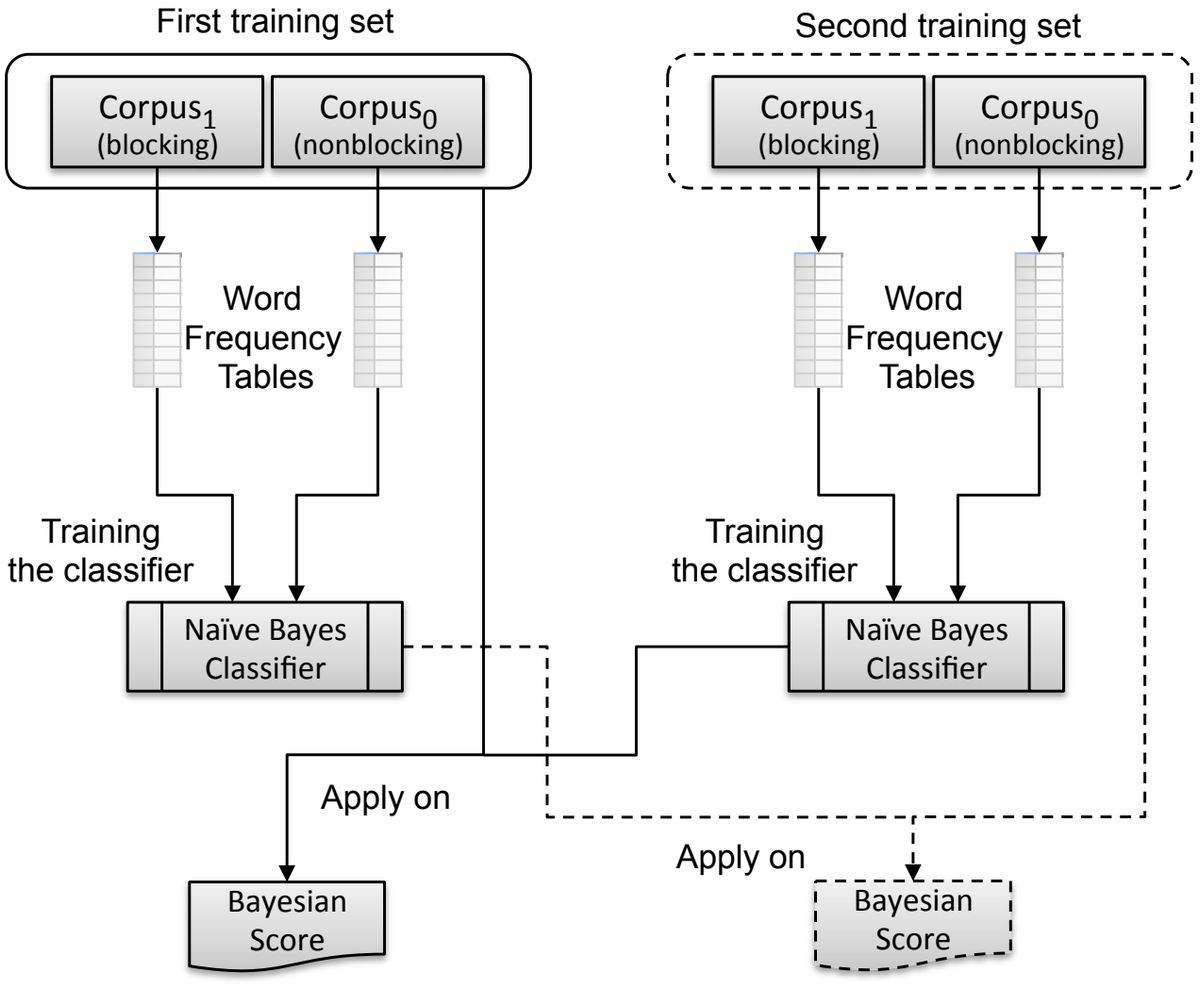


Data use in the study: Chromium, Eclipse, FreeDesktop, Mozilla, NetBeans and OpenOffice

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
- $Corpus_1$ comment/description for blocking bugs
- $Corpus_0$ 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.

Prediction models

- Primary: Decision tree classifier
- Also:
    - Naive Bayes
    - kNN
    - Random Forests
    - Zero-R

Confusion matrix: true/false positive/negatives

		True class	
		Blocking	Non-blocking
Classified as	Blocking	TP	FP
	Non-blocking	FN	TN

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 $\textit{F-measure} = \frac{2*Pr*Re}{Pr+Re}$.
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, …, 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

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

Other measures

- Time to identify a blocking bug
- Degree of blockiness

Characterizing and Predicting Blocking Bugs

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

MSR
2014

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

```
1 static final SimpleDateFormat cDateFormat
2   = 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

```
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   ...
```

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.tr- ace() 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

```
1 try { socket.close (); }
2 catch (Exception ignore) {}
3 try { reader.close (); }
4 catch (Exception ignore) {}
```

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

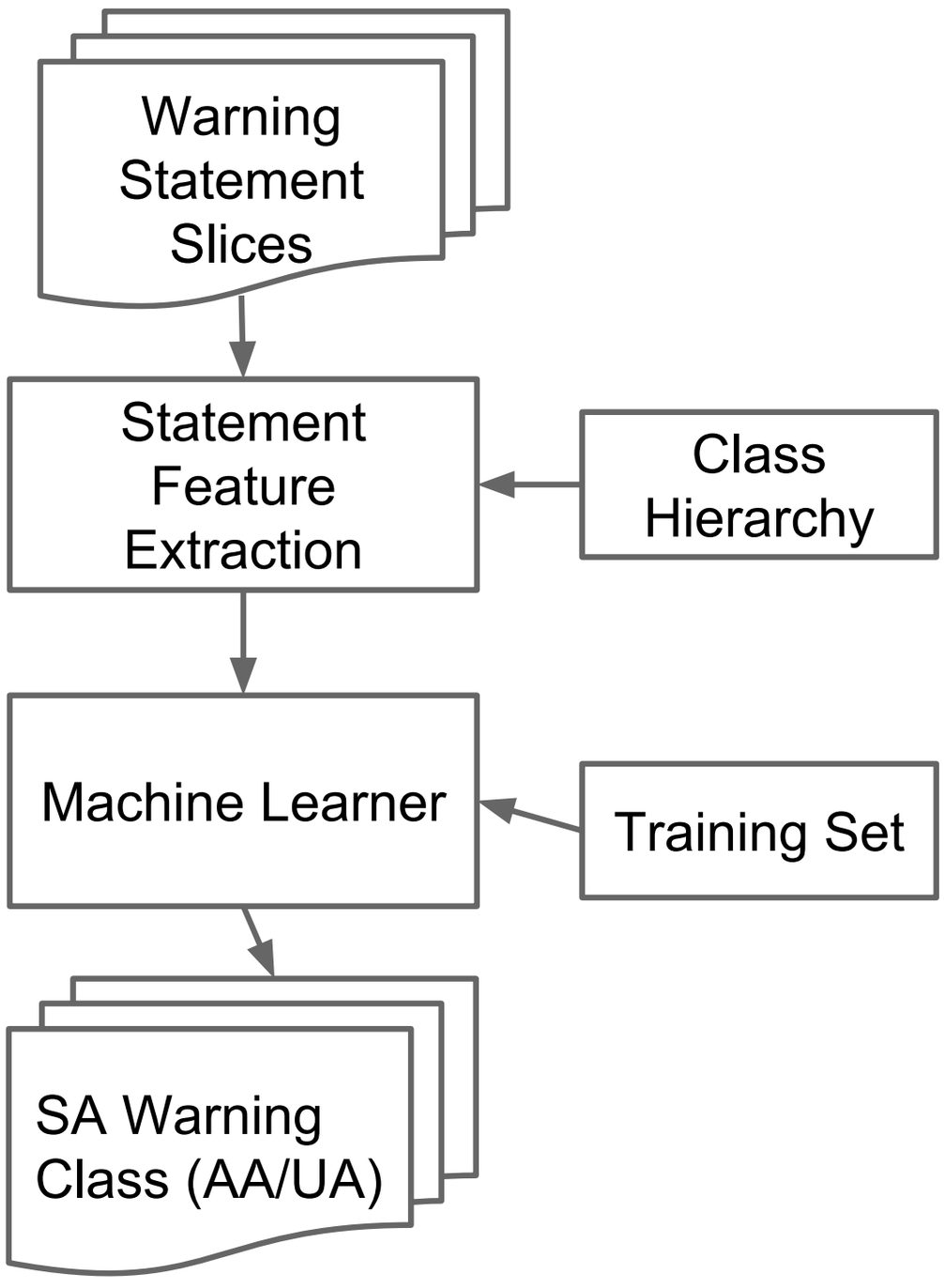
- TP, FP, TN, FN — not applicable
- Use AA (actionable alert) and UA (unactionable altert)
- SA tools use patterns for alert generation
- Tool developers must strive to minimize UA
- Discovering and adding new UA patterns is time consuming
- Thus, add machine learning with alert characteristics (AC)
  - 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?*

Classify SA alerts as AA / UA



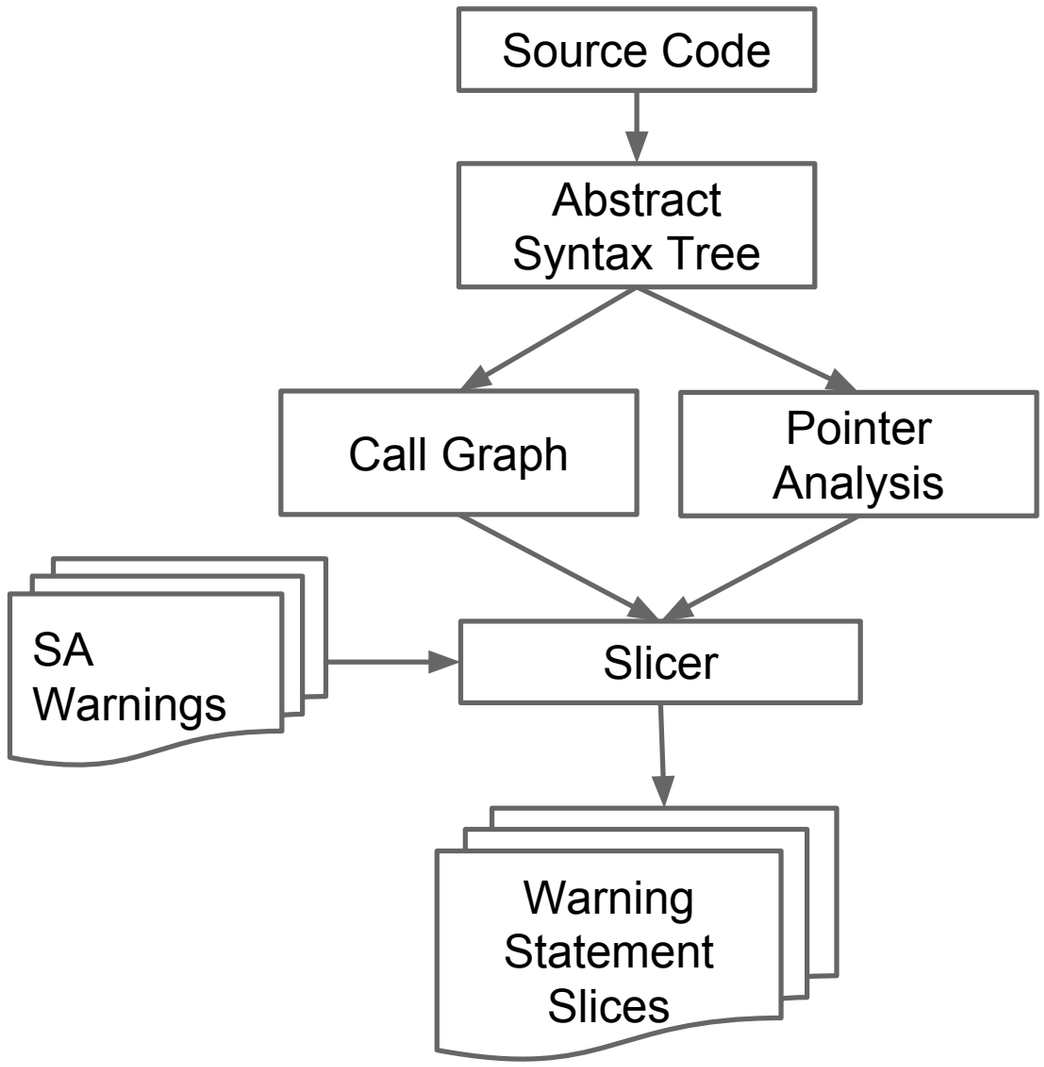
1. We calculate a set of related statements by slicing the program at the site of the alert.

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

Seed Statements	
Statement Type	**Alert Characteristic**
Call	Call Name
	Call Class
	Call Parameter Signature
	Return Type
New	New Type
	New Concrete Type
Binary Operation	Operator
Field Access	Field Access Class
	Field Access Field
Catch	Catch
Non-Seed Statements	
Statement Type	**Alert Characteristic**
Field	Name
	Type
	Visibility
	Is Static/Final
Method	Visibility
	Return Type
	Is Static/Final/Abstract/Protected
Class	Visibility
	Is Abstract/Interface/Array Class

- **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 $|R_{NA}|/|A| * 100$. 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 $\%AA_N = 5/10 * 100 = 50\%$. This formula is shown below.

$$\%AA_N = \frac{|R_{NA}|}{|A|} * 100$$

Ground truth

- 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.**

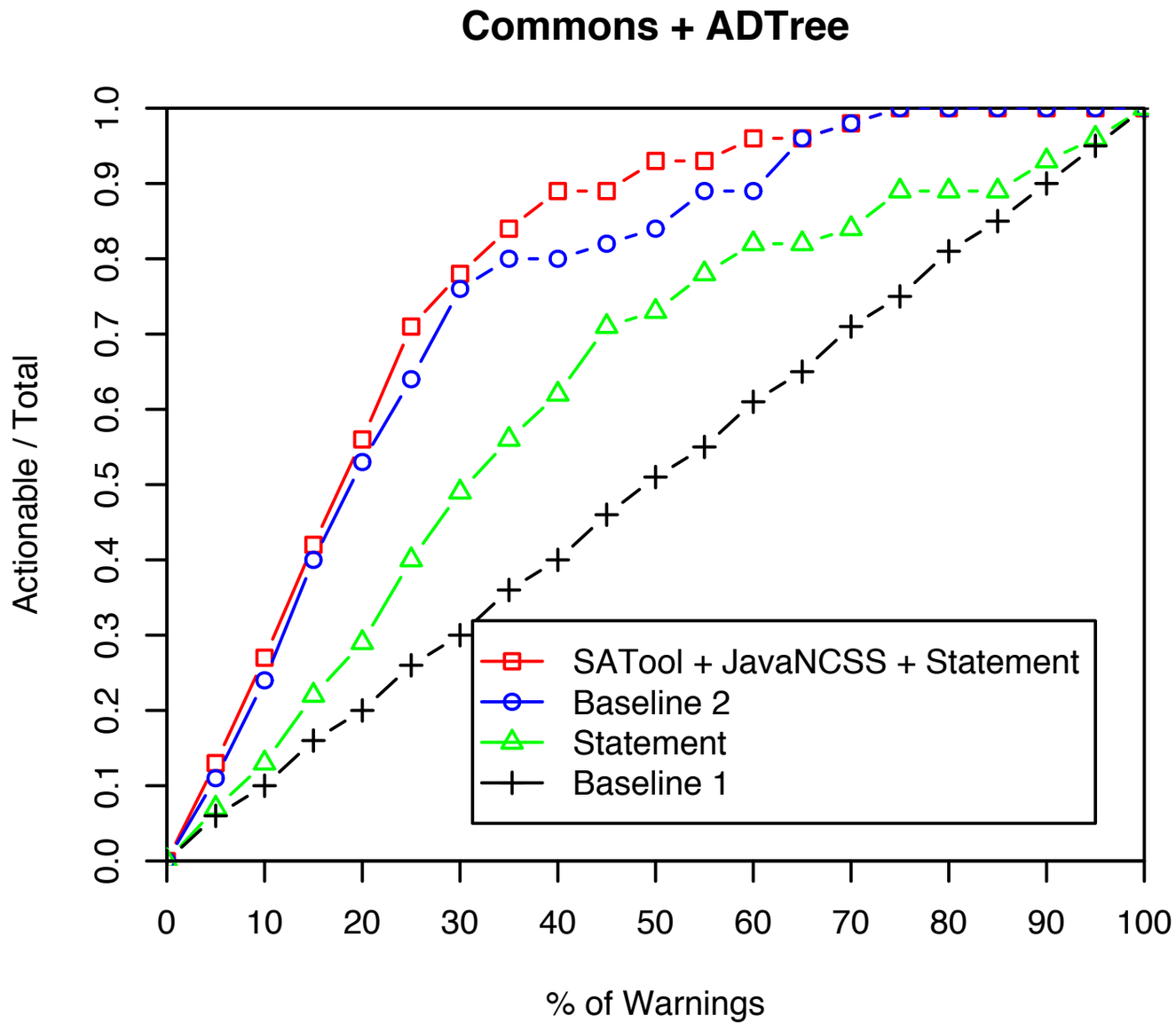
Ground truth

Metrics from 10-fold cross validation using only statement ACs

	$\%AA_N$, $N =$			**Unactionable**			**Actionable**			**Weighted**		
	10	20	30	Precision	Recall	F	Precision	Recall	F	Precision	Recall	F
Tomcat6												
ADTree	0.36	0.42	0.52	0.96	1.00	0.96	1.00	0.31	0.47	0.93	0.93	0.91
Naive Bayes	0.40	0.49	0.61	0.93	0.93	0.93	0.38	0.41	0.39	0.88	0.87	0.87
BayesNet	0.33	0.51	0.60	0.93	0.92	0.93	0.38	0.43	0.41	0.88	0.87	0.88
Commons												
ADTree	0.13	0.29	0.49	0.75	0.73	0.75	0.53	0.58	0.55	0.69	0.68	0.68
Naive Bayes	0.24	0.49	0.64	0.60	0.47	0.60	0.45	0.84	0.59	0.71	0.60	0.60
BayesNet	0.18	0.42	0.58	0.80	0.81	0.80	0.62	0.58	0.60	0.73	0.73	0.73
Logging												
ADTree	0.31	0.46	0.46	0.95	1.00	0.95	0.00	0.00	0.00	0.82	0.91	0.86
Naive Bayes	0.23	0.46	0.54	0.59	0.43	0.59	0.10	0.62	0.17	0.84	0.45	0.55
BayesNet	0.15	0.38	0.54	0.89	0.87	0.89	0.11	0.15	0.13	0.83	0.80	0.82

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

MSR
2014

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

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

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

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

MSR
2014

End of lecture

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