Mining Software Engineering Data

Ahmed E. Hassan
Queen’s University
www.cs.queensu.ca/~ahmed
ahmed@cs.queensu.ca

Tao Xie
North Carolina State University
www.csc.ncsu.edu/faculty/xie
xie@csc.ncsu.edu

Some slides are adapted from tutorial slides co-prepared by Jian Pei from Simon Fraser University, Canada
An up-to-date version of this tutorial is available at http://ase.csc.ncsu.edu/dmse/
Mining SE Data

- SE data can be used to:
  - Gain empirically-based understanding of software development
  - Predict, plan, and understand various aspects of a project
  - Support future development and project management activities

Overview of Mining SE Data

- SE data can be used to:
  - Support future development and project management activities
  -获得 empirically-based understanding of software development
  - Predict, plan, and understand various aspects of a project

Tutorial Outline

- **Part I**: What can you learn from SE data?
  - A sample of notable recent findings for different SE data types

- **Part II**: How can you mine SE data?
  - Overview of data mining techniques
  - Overview of SE data processing tools and techniques
Types of SE Data

- **Historical data**
  - Version or source control: cvs, subversion, perforce
  - Bug systems: bugzilla, GNATS, JIRA
  - Mailing lists: mbox
- **Multi-run and multi-site data**
  - Execution traces
  - Deployment logs
- **Source code data**
  - Source code repositories: sourceforge.net, google code

Historical Data

“History is a guide to navigation in perilous times. History is who we are and why we are the way we are.”

- David C. McCullough

Historical Data

- Track the evolution of a software project:
  - *source control systems* store changes to the code
  - *defect tracking systems* follow the resolution of defects
  - *archived project communications* record rationale for decisions throughout the life of a project
- Used primarily for record-keeping activities:
  - checking the status of a bug
  - retrieving old code

Survey of Software Maintenance Activities

- **Perfective:** add new functionality
- **Corrective:** fix faults
- **Adaptive:** new file formats, refactoring

Percentage of Project Costs Devoted to Maintenance

![Graph showing percentage of project costs devoted to maintenance over time](image)

- Lientz & Swanson [1978]
- Nosek, Palvia [1990]
- Schach, Jin, Yu, Heller, Offutt [2003]
- Mining ChangeLogs (Linux, GCC, RTP)

Source Control Repositories

![Diagram illustrating source control repositories](image)
Source Control Repositories

• A source control system tracks changes to ChangeUnits
• Example of ChangeUnits:
  – File (most common)
  – Function
  – Dependency (e.g., Call)
• Each ChangeUnit:
  – Records the developer, change time, change message, co-changing Units

Change Propagation

“How does a change in one source code entity propagate to other entities?”

Determine Initial Entity To Change

Determine Other Entities To Change

Consult Guru for Advice

Measuring Change Propagation

Precision = \frac{\text{predicted entities which changed}}{\text{predicted entities}}

Recall = \frac{\text{predicted entities which changed}}{\text{changed entities}}

• We want:
  – High Precision to avoid wasting time
  – High Recall to avoid bugs

Guiding Change Propagation

• Mine association rules from change history
• Use rules to help propagate changes:
  – Recall as high as 44%
  – Precision around 30%
• High precision and recall reached in < 1mth
• Prediction accuracy improves prior to a release (i.e., during maintenance phase)

Conceptual & Concrete Architecture (NetBSD)

• Traditional dependency graphs and program understanding models usually do not use historical information
• Static dependencies capture only a static view of a system – not enough detail!
• Development history can help understand the current structure (architecture) of a software system

Code Sticky Notes

• Traditional dependency graphs and program understanding models usually do not use historical information
• Static dependencies capture only a static view of a system – not enough detail!
• Development history can help understand the current structure (architecture) of a software system
Investigating Unexpected Dependencies Using Historical Code Changes

- Eight unexpected dependencies
- All except two dependencies existed since day one:
  - Virtual Address Maintenance → Pager
  - Pager → Hardware Translations

<table>
<thead>
<tr>
<th>Which?</th>
<th>vm_map_entry_create (in src/sys/vm/Attic/vm_map.c) depends on pager_map (in /src/sys/uvm/uvm_pager.c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Who?</td>
<td>sean eric fagan</td>
</tr>
<tr>
<td>When?</td>
<td>1993/04/09 15:54:59 Revision 1.2 of src/sys/vm/Attic/vm_map.c from sean eric fagan</td>
</tr>
<tr>
<td>Why?</td>
<td>From sean eric fagan: it seems to keep the vm system from deadlocking the system when it runs out of swap + physical memory. prevents the system from giving the last page(s) to anything but the referenced &quot;processes&quot; (especially important is the pager process, which should never have to wait for a free page).</td>
</tr>
</tbody>
</table>

Studying Conway’s Law

- Conway’s Law:
  “The structure of a software system is a direct reflection of the structure of the development team”

Source Control and Bug Repositories

Predicting Bugs

- Studies have shown that most complexity metrics correlate well with LOC:
  - Graves et al. 2000 on commercial systems
  - Herrera et al. 2007 on open source systems
- Noteworthy findings:
  - Previous bugs are good predictors of future bugs
  - The more a file changes, the more likely it will have bugs in it
  - Recent changes affect more the bug potential of a file over older changes (weighted time damp models)
  - Number of developers is of little help in predicting bugs
  - Hard to generalize bug predictors across projects unless in similar domains [Nagappan, Ball et al. 2006]

Using Imports in Eclipse to Predict Bugs

- 71% of files that import compiler packages, had to be fixed later on.
  import org.eclipse.jdt.internal.compiler.lookup.*;
  import org.eclipse.jdt.internal.compiler.*;
  import org.eclipse.jdt.internal.compiler.ast.*;
  import org.eclipse.jdt.internal.compiler.util.*;
  …
  import org.eclipse.pde.core.*;
  import org.eclipse.jface.wizard.*;
  import org.eclipse.ui.*;

- 14% of all files that import ui packages, had to be fixed later on.
  [Schröter et al. 06]
Don’t program on Fridays ;-)  

![Bar chart showing percentage of bug-introducing changes for eclipse](Zimmermann et al. 06)

Classifying Changes as Buggy or Clean

- Given a change can we warn a developer that there is a bug in it?
  - Recall/Precision in 50-60% range

![Image of software development tool](Sung et al. 06)

Project Communication (Mailinglists)

- Most open source projects communicate through mailing lists or IRC channels
- Rich source of information about the inner workings of large projects
- Discussions cover topics such as future plans, design decisions, project policies, code or patch reviews
- Social network analysis could be performed on discussion threads

Social Network Analysis

- Mailing list activity:
  - strongly correlates with code change activity
  - moderately correlates with document change activity
- Social network measures (in-degree, out-degree, betweenness) indicate that committers play a more significant role in the mailing list community than non-committers

![Social network diagram](Bird et al. 06)

Immigration Rate of Developers

- When will a developer be invited to join a project?
  - Expertise vs. interest

![Graph of smoothed hazard estimate](Bird et al. 07)
The Patch Review Process

- Two review styles
  - RTC: Review-then-commit
  - CTR: Commit-then-review
- 80% patches reviewed within 3.5 days and 50% reviewed in <19 hrs

Measure a team’s morale around release time?

<table>
<thead>
<tr>
<th>Dimension</th>
<th>1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimistic</td>
<td>0.37</td>
<td></td>
</tr>
<tr>
<td>Tentative</td>
<td>-1.3</td>
<td>*</td>
</tr>
<tr>
<td>References to Time</td>
<td>1.1</td>
<td>*</td>
</tr>
<tr>
<td>Future tense verbs</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td>Social Processes</td>
<td>0.74</td>
<td></td>
</tr>
<tr>
<td>Inclusive</td>
<td>-4.64</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. Mean differences for Apache 1.3 and 2.0 releases. (* p < 0.05, otherwise p ≤ 0.05)

- Study the content of messages before and after a release
- Use dimensions from a psychometric text analysis tool:
  - After Apache 1.3 release there was a drop in optimism
  - After Apache 2.0 release there was an increase in sociability

A. E. Hassan and T. Xie: Mining Software Engineering Data 37

Code Entities

<table>
<thead>
<tr>
<th>Source data</th>
<th>Mined info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable names and function names</td>
<td>Software categories</td>
</tr>
<tr>
<td>Statement seq in a basic block</td>
<td>Copy-paste code</td>
</tr>
<tr>
<td>Set of functions, variables, and data types within a C function</td>
<td>Programming rules</td>
</tr>
<tr>
<td>Sequence of methods within a Java method</td>
<td>API usages</td>
</tr>
<tr>
<td>API method signatures</td>
<td>API Jungloids</td>
</tr>
</tbody>
</table>

A. E. Hassan and T. Xie: Mining Software Engineering Data 38

Mining API Usage Patterns

- How should an API be used correctly?
  - An API may serve multiple functionalities
  - Different styles of API usage
- “I know what type of object I need, but I don’t know how to write the code to get the object” [Mandelin et al. 05]
  - Can we synthesize jungloid code fragments automatically?
  - Given a simple query describing the desired code in terms of input and output types, return a code segment
- “I know what method call I need, but I don’t know how to write code before and after this method call” [Xie&Pei 06]

A. E. Hassan and T. Xie: Mining Software Engineering Data 41

Relationships btw Code Entities

- Mine framework reuse patterns [Michail 00]
  - Membership relationships
    - A class contains membership functions
  - Reuse relationships
    - Class inheritance/ instantiation
    - Function invocations/overriding
- Mine software plagiarism [Liu et al. 06]
  - Program dependence graphs

Program Execution Traces

Method-Entry/Exit States
- Goal: mine specifications (pre/post conditions) or object behavior (object transition diagrams)
- State of an object
  - Values of transitively reachable fields
- Method-entry state
  - Receiver-object state, method argument values
- Method-exit state
  - Receiver-object state, updated method argument values, method return value

Other Profiled Program States
- Goal: detect or locate bugs
- Values of variables at certain code locations [Hangal&Lam 02]
  - Object/static field read/write
  - Method-call arguments
  - Method returns
- Sampled predicates on values of variables [Liblit et al. 03/05][Liu et al. 05]

Executed Structural Entities
- Goal: locate bugs
- Executed branches/paths, def-use pairs
- Executed function/method calls
  - Group methods invoked on the same object
- Profiling options
  - Execution hit vs. count
  - Execution order (sequences)

Part I Review
- We presented notable results based on mining SE data such as:
  - Historical data:
    - Source control: predict co-changes
    - Bug databases: predict bug likelihood
    - Mailing lists: gauge team morale around release time
  - Other data:
    - Program source code: mine API usage patterns
    - Program execution traces: mine specs, detect or locate bugs

Q&A and break
Part II: How can you mine SE data?
– Overview of data mining techniques
– Overview of SE data processing tools and techniques

Frequent Itemsets
• Itemset: a set of items
  – E.g., acm={a, c, m}
• Support of itemsets
  – Sup(acm)=3
• Given min_sup = 3, acm is a frequent pattern
• Frequent pattern mining: find all frequent patterns in a database

Transaction database TDB

<table>
<thead>
<tr>
<th>TID</th>
<th>Items bought</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>f, a, c, d, g, l, m, p</td>
</tr>
<tr>
<td>200</td>
<td>a, b, c, f, l, m, o</td>
</tr>
<tr>
<td>300</td>
<td>b, f, h, j, o</td>
</tr>
<tr>
<td>400</td>
<td>b, c, k, s, p</td>
</tr>
<tr>
<td>500</td>
<td>a, f, c, e, l, p, m, n</td>
</tr>
</tbody>
</table>

Association Rules
• (Time ∈ {Fri, Sat}) ∧ buy(X, diaper) → buy(X, beer)
  – Dads taking care of babies in weekends drink beer
• Itemsets should be frequent
  – It can be applied extensively
• Rules should be confident
  – With strong prediction capability

A Simple Case
• Finding highly correlated method call pairs
• Confidence of pairs helps
  – Conf(<a,b>)=support(<a,b>)/support(<a,a>)
• Check the revisions (fixes to bugs), find the pairs of method calls whose confidences have improved dramatically by frequent added fixes
  – Those are the matching method call pairs that may often be violated by programmers

Conflicting Patterns
• 999 out of 1000 times spin_lock is followed by spin_unlock
  – The single time that spin_unlock does not follow may likely be an error
• We can detect an error without knowing the correctness rules

[Livshits&Zimmermann 05] [Li&Zhou 05, Livshits&Zimmermann 05, Yang et al. 06]
Detect Copy-Paste Code

• Apply closed sequential pattern mining techniques
• Customizing the techniques
  – A copy-paste segment typically does not have big gaps
  – Output the instances of patterns (i.e., the copy-pasted code segments) instead of the patterns
  – Use small copy-pasted segments to form larger ones
  – Prune false positives: tiny segments, unmappable segments, overlapping segments, and segments with large gaps

[Li et al. 04]

Find Bugs in Copy-Pasted Segments

• For two copy-pasted segments, are the modifications consistent?
  – Identifier $a$ in segment $S_1$ is changed to $b$ in segment $S_2$ 3 times, but remains unchanged once – likely a bug
  – The heuristic may not be correct all the time
• The lower the unchanged rate of an identifier, the more likely there is a bug

[Li et al. 04]

Mining Rules in Traces

• Mine association rules or sequential patterns $S \rightarrow F$, where $S$ is a statement and $F$ is the status of program failure
• The higher the confidence, the more likely $S$ is faulty or related to a fault
• Using only one statement at the left side of the rule can be misleading, since a fault may be led by a combination of statements
  – Frequent patterns can be used to improve

[Denmat et al. 05]

Mining Emerging Patterns in Traces

• A method executed only in failing runs is likely to point to the defect
  – Comparing the coverage of passing and failing program runs helps
• Mining patterns frequent in failing program runs but infrequent in passing program runs
  – Sequential patterns may be used

[Dallmeier et al. 05, Denmat et al. 05]

Types of Frequent Pattern Mining

• Association rules
  – open $\rightarrow$ close
• Frequent itemset mining
  – \{open, close\}
• Frequent subsequence mining
  – open $\rightarrow$ close
• Frequent partial order mining
• Frequent graph mining
• Finite automaton mining

Data Mining Techniques in SE

• Association rules and frequent patterns
  • Classification
  • Clustering
  • Misc.
Classification: A 2-step Process

- Model construction: describe a set of predetermined classes
  - Training dataset: tuples for model construction
    - Each tuple/sample belongs to a predefined class
  - Classification rules, decision trees, or math formulae
- Model application: classify unseen objects
  - Estimate accuracy of the model using an independent test set
  - Acceptable accuracy → apply the model to classify tuples with unknown class labels

Model Construction

Model Construction

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
<th>Years</th>
<th>Tenured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike</td>
<td>Ass. Prof</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>Mary</td>
<td>Ass. Prof</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>Bill</td>
<td>Prof</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Jim</td>
<td>Asso. Prof</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>Dave</td>
<td>Ass. Prof</td>
<td>6</td>
<td>No</td>
</tr>
<tr>
<td>Anne</td>
<td>Asso. Prof</td>
<td>3</td>
<td>No</td>
</tr>
</tbody>
</table>

IF rank = ‘professor’ OR years > 6 THEN tenured = ‘yes’

Model Application

<table>
<thead>
<tr>
<th>Name</th>
<th>Rank</th>
<th>Years</th>
<th>Tenured</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom</td>
<td>Ass. Prof</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>Merlisa</td>
<td>Asso. Prof</td>
<td>7</td>
<td>No</td>
</tr>
<tr>
<td>George</td>
<td>Prof</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>Joseph</td>
<td>Asso. Prof</td>
<td>7</td>
<td>Yes</td>
</tr>
</tbody>
</table>

 Supervised vs. Unsupervised Learning

- Supervised learning (classification)
  - Supervision: objects in the training data set have labels
  - New data is classified based on the training set
- Unsupervised learning (clustering)
  - The class labels of training data are unknown
  - Given a set of measurements, observations, etc. with the aim of establishing the existence of classes or clusters in the data

GUI-Application Stabilizer

- Given a program state \( S \) and an event \( e \), predict whether \( e \) likely results in a bug
  - Positive samples: past bugs
  - Negative samples: “not bug” reports
- A k-NN based approach
  - Consider the k closest cases reported before
  - Compare \( \sum 1/d \) for bug cases and not-bug cases, where \( d \) is the similarity between the current state and the reported states
  - If the current state is more similar to bugs, predict a bug

Data Mining Techniques in SE

- Association rules and frequent patterns
- Classification
- Clustering
- Misc.
What is Clustering?
• Group data into clusters
  – Similar to one another within the same cluster
  – Dissimilar to the objects in other clusters
  – Unsupervised learning: no predefined classes

Clustering and Categorization
• Software categorization
  – Partitioning software systems into categories
• Categories predefined – a classification problem
• Categories discovered automatically – a clustering problem

Software Categorization - MUDABlue
• Understanding source code
  – Use Latent Semantic Analysis (LSA) to find similarity between software systems
  – Use identifiers (e.g., variable names, function names) as features
    • “gtk_window” represents some window
    • The source code near “gtk_window” contains some GUI operation on the window
• Extracting categories using frequent identifiers
  – “gtk_window”, “gtk_main”, and “gpointer” → GTK related software system
  – Use LSA to find relationships between identifiers
  [Kawaguchi et al. 04]

Data Mining Techniques in SE
• Association rules and frequent patterns
• Classification
• Clustering
• Misc.

Other Mining Techniques
• Automaton/grammar/regular expression learning
• Searching/matching
• Concept analysis
• Template-based analysis
• Abstraction-based analysis

How to Do Research in Mining SE Data
http://sites.google.com/site/asergrp/dmse/miningalgs
How to do research in mining SE data

• We discussed results derived from:
  – Historical data:
    • Source control
    • Bug databases
    • Mailing lists
  – Program data:
    • Program source code
    • Program execution traces
• We discussed several mining techniques
• We now discuss how to:
  – Get access to a particular type of SE data
  – Process the SE data for further mining and analysis

Source Control Repositories

Concurrent Versions System (CVS) Comments

• cvs log – displays for all revisions and its comments for each file
• cvs diff – shows differences between different versions of a file
• Used for program understanding

Code Version Histories

• CVS provides file versioning
  – Group individual per-file changes into **individual transactions**: checked in by the same author with the same check-in comment within a short time window
• CVS manages only files and line numbers
  – Associate **syntactic entities** with line ranges
• Filter out long transactions not corresponding to meaningful atomic changes
  – E.g., features and bug fixes vs. branch and merging
• Used to mine co-changed entities

Getting Access to Source Control

• These tools are commonly used
  – **Email**: ask for a local copy to avoid taxing the project’s servers during your analysis and development
  – **CVSup**: mirrors a repository if supported by the particular project
  – **rsync**: a protocol used to mirror data repositories
  – **CVSsuck**:
    • Uses the CVS protocol itself to mirror a CVS repository
    • The CVS protocol is not designed for mirroring; therefore, CVSsuck is not efficient
    • Use as a last resort to acquire a repository due to its inefficiency
    • Used primarily for dead projects
Recovering Information from CVS

Challenges in recovering information from CVS

CVS Limitations

- CVS has limited query functionality and is slow
- CVS does not track co-changes
- CVS tracks only changes at the file level

Inferring Transactions in CVS

- Sliding Window:
  - Time window: [3-5mins on average]
    - min 3mins
    - as high as 21 mins for merges
- Commit Mails

Noise in CVS Transactions

- Drop all transactions above a large threshold
  - “Change $include filenames from <foo.h> [sigh] to <openssl.h>” (255 files, OPENSSL)
  - “Change function: to ANSI C.” (101 files, OPENSSL)
- For Branch merges either look at CVS comments or use heuristic algorithm proposed by Fischer et al. 2003

A Note about large commits
Noise in detecting developers

- Few developers are given commit privileges
- Actual developer is usually mentioned in the change message
- One must study project commit policies before reaching any conclusions

Source Control and Bug Repositories

Acquiring Bugzilla data

- Download bug reports using the XML export feature (in chunks of 100 reports)
- Download attachments (one request per attachment)
- Download activities for each bug report (one request per bug report)

Using Bugzilla Data

- Depending on the analysis, you might need to rollback the fields of each bug report using the stored changes and activities
- Linking changes to bug reports is more or less straightforward:
  - Any number in a log message could refer to a bug report
  - Usually good to ignore numbers less than 1000. Some issue tracking systems (such as JIRA) have identifiers that are easy to recognize (e.g., JIRA-4223)

Bugzilla

Sample Bugzilla Bug Report

Bugzilla: open source bug tracking tool
http://www.bugzilla.org/
So far: Focus on fixes

Fixes give only the location of a defect, not when it was introduced.

Bug-introducing changes

Bug-introducing changes are changes that lead to problems as indicated by later fixes.

The SZZ algorithm

The SZZ algorithm
Project Communication – Mailing lists

Acquiring Mailing lists

• Usually archived and available from the project’s webpage
• Stored in mbox format:
  – The mbox file format sequentially lists every message of a mail folder

Challenges using Mailing lists data I

• Unstructured nature of email makes extracting information difficult
  – Written English
• Multiple email addresses
  – Must resolve emails to individuals
• Broken discussion threads
  – Many email clients do not include “In-Reply-To” field

Challenges using Mailing lists data II

• Country information is not accurate
  – Many sites are hosted in the US:
    • Yahoo.com.ar is hosted in the US
• Tools to process mailbox files rarely scale to handle such large amount of data (years of mailing list information)
  – Will need to write your own

Program Source Code

Acquiring Source Code

• Ahead-of-time download directly from code repositories (e.g., Sourceforge.net)
  – Advantage: offline perform slow data processing and mining
  – Some tools (Prospector and Strathcona) focus on framework API code such as Eclipse framework APIs
• On-demand search through code search engines:
  – E.g., http://www.google.com/codesearch
  – Advantage: not limited on a small number of downloaded code repositories

Prospector: http://snobol.cs.berkeley.edu/prospector
Processing Source Code

- Use one of various static analysis/compiler tools (McGill Soot, BCEL, Berkeley CIL, GCC, etc.)
- But sometimes downloaded code may not be compilable
- May use simple heuristics/analysis to deal with some language features [Xie&Pei 06, Mandelin et al. 05]
  - Conditional, loops, inter-procedural, downcast, etc.

Program Execution Traces

- Acquiring Execution Traces
  - Code instrumentation or VM instrumentation
    - Java: ASM, BCEL, SERP, Soot, Java Debug Interface
    - C/C++/Binary: Valgrind, Fjalar, Dyninst
- Processing Execution Traces
  - Processing types: online (as data is encountered) vs. offline (write data to file)
  - May need to group relevant traces together
    - e.g., based on receiver-object references
    - e.g., based on corresponding method entry/exit

Repositories Available Online

- Promise repository: [http://promisedata.org](http://promisedata.org)
- iBug: [http://www.st.cs.uni-sb.de/ibugs/](http://www.st.cs.uni-sb.de/ibugs/)
  - http://msr.uwaterloo.ca/msr2008/challenge/)
- Software-artifact infrastructure repository: [http://sir.uni.edu/portal/index.html](http://sir.uni.edu/portal/index.html)

Tools and Repositories
Eclipse Bug Data

• Defect counts are listed as counts at the plug-in, package and compilation unit levels.
• The value field contains the actual number of pre- ("pre") and post-release defects ("post").
• The average ("avg") and maximum ("max") values refer to the defects found in the compilation units ("compilationunits").

Abstract Syntax Tree Nodes in Eclipse Bug Data

• The AST node information can be used to calculate various metrics

Example Graphs from FlossMole


FLOSSmole

• FLOSSmole
  – provides raw data about open source projects
  – provides summary reports about open source projects
  – integrates donated data from other research teams
  – provides tools so you can gather your own data
• Data sources
  – Sourceforge
  – Freshmeat
  – Rubyforge
  – ObjectWeb
  – Free Software Foundation (FSF)
  – SourceKibitzer
    http://flossmole.org/

Analysis Tools

• R
  – http://www.r-project.org/
  – R is a free software environment for statistical computing and graphics
• Aisee
  – http://www.aisee.com/
  – Aisee is a graph layout software for very large graphs
• WEKA
  – http://www.cs.waikato.ac.nz/ml/weka/
  – WEKA contains a collection of machine learning algorithms for data mining tasks
• RapidMiner (YALE)
  – http://rapidminer.com/
• More tools: http://ase.csc.ncsu.edu/site/asergrp/dmse/resources
Data Extraction/Processing Tools

- **Kenyon**
  - http://dforge.cse.ucsc.edu/projects/kenyon/
- **MyIn/Mylar** (comes with API for Bugzilla and JIRA)
  - http://www.eclipse.org/myln/
- **Libresoft toolset**
  - Tools (cvsanaly/mlstats/detras) for recovering data from cvs/svn and mailinglists
  - http://forge.morfeo-project.org/projects/libresoft-tools/

Publishing Advice

- Report the statistical significance of your results:
  - Get a statistics book (one for social scientist, not for mathematicians)
- Discuss any limitations of your findings based on the characteristics of the studied repositories:
  - Make sure you manually examine the repositories. Do not fully automate the process!
  - Use random sampling to resolve issues about data noise
- Relevant conferences/workshops:
  - main SE conferences, ICSM, ISSTA, MSR, WODA, …

Mining Software Repositories

- Very active research area in SE:
  - MSR is the most attended ICSE event in last 7 yrs
  - Special Issue of IEEE TSE 2005 on MSR:
    - 15% of all submissions of TSE in 2004
    - Fastest review cycle in TSE history: 8 months
  - Special Issue Empirical Software Engineering 2009
  - MSR 2011!

Example Tools

- **MAPO**: mining API usages from open source repositories [Xie&Pei 06]
- **DynaMine**: mining error/usage patterns from code revision histories [Livshits&Zimmermann 05]
- **BugTriage**: learning bug assignments from historical bug reports [Anvik et al. 06]
### Demand-Driven Or Not

<table>
<thead>
<tr>
<th></th>
<th>Any-gold mining</th>
<th>Demand-driven mining</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Examples</strong></td>
<td>DynaMine, …</td>
<td>MAPO, BugTriage, …</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>Surface up only cases that are applicable</td>
<td>Exploit demands to filter out irrelevant information</td>
</tr>
<tr>
<td><strong>Issues</strong></td>
<td>How much gold is good enough given the amount of data to be mined?</td>
<td>How high percentage of cases would work well?</td>
</tr>
</tbody>
</table>

### Code vs. Non-Code

<table>
<thead>
<tr>
<th></th>
<th>Code/ Programming Langs</th>
<th>Non-Code/ Natural Langs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Examples</strong></td>
<td>MAPO, DynaMine, …</td>
<td>BugTriage, CVS/Code comments, emails, docs</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>Relatively stable and consistent representation</td>
<td>Common source of capturing programmers’ intentions</td>
</tr>
<tr>
<td><strong>Issues</strong></td>
<td>What project/context-specific heuristics to use?</td>
<td></td>
</tr>
</tbody>
</table>

### Static vs. Dynamic

<table>
<thead>
<tr>
<th></th>
<th>Static Data: code bases, change histories</th>
<th>Dynamic Data: prog states, structural profiles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Examples</strong></td>
<td>MAPO, DynaMine, …</td>
<td>Spec discovery, …</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>No need to set up exec environment; More scalable</td>
<td>More-precise info</td>
</tr>
<tr>
<td><strong>Issues</strong></td>
<td>How to reduce false positives?</td>
<td>How to reduce false negatives?</td>
</tr>
</tbody>
</table>

### Snapshot vs. Changes

<table>
<thead>
<tr>
<th></th>
<th>Code snapshot</th>
<th>Code change history</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Examples</strong></td>
<td>MAPO, …</td>
<td>DynaMine, …</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>Larger amount of available data</td>
<td>Revision transactions encode more-focused entity relationships</td>
</tr>
<tr>
<td><strong>Issues</strong></td>
<td>How to group CVS changes into transactions?</td>
<td></td>
</tr>
</tbody>
</table>

### Characteristics in Mining SE Data

- Improve quality of source data: data preprocessing
  - MAPO: inlining, reduction
  - DynaMine: call association
  - BugTriage: labeling heuristics, inactive-developer removal
- Reduce uninteresting patterns: pattern postprocessing
  - MAPO: compression, reduction
  - DynaMine: dynamic validation
- Source data may not be sufficient
  - DynaMine: revision histories
  - BugTriage: historical bug reports

**SE-Domain-Specific Heuristics are important**