

ソフトウェアジャパン2013, ITフォーラムセッション  
2013年2月15日

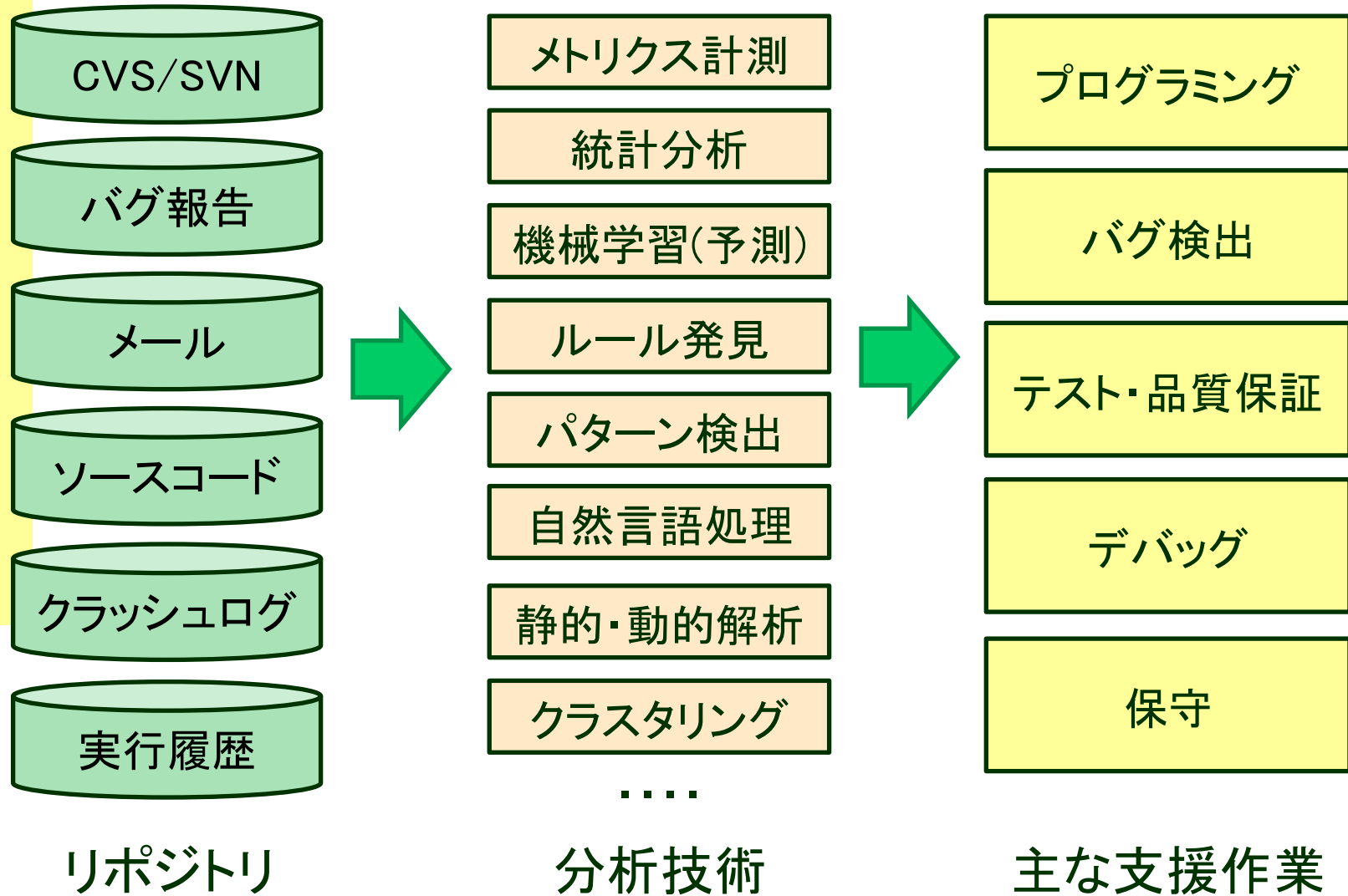
# ソフトウェアリポジトリマイニングの 技術動向とその応用

奈良先端科学技術大学院大学

門田暁人

akito-m@is.naist.jp

# リポジトリマイニングとは



# リポジトリマイニングとは

---

- 学術界では、リポジトリマイニングの研究が盛んである。
  - 若手研究者Ahmed E. Hassan, Tao Xie, Sunghun Kim, Tien Nguyen, Thomas Zimmermanなどが、ICSE, FSE, ASEなどトップカンファレンスに次々と論文を発表している。
  - OSSプロジェクトを題材としている。
- 産業界に役立つ技術・知見もたくさんあるはず。

リポジトリマイニングは  
何の役に立つのか？

# プログラミング支援:ライブラリ使用状況の分析

- 頻出オープンソースライブラリの分析[1]
- バージョンごとのライブラリ使用頻度の分析[2]

| ライブラリ               | 使用頻度 |
|---------------------|------|
| commons-logging     | 62%  |
| log4j               | 55%  |
| commons-collections | 55%  |
| commons-codec       | 52%  |
| commons-lang        | 41%  |
| xercesImpl          | 36%  |
| slf4j-api           | 33%  |
| junit               | 30%  |
| commons-discovery   | 29%  |
| axis                | 24%  |
| stax-api            | 21%  |

| Library name and version | Times used |
|--------------------------|------------|
| junit 3.8.1              | 60         |
| junit 3.8.2              | 9          |
| junit 4.4                | 7          |
| log4j 1.2.8              | 10         |
| log4j 1.2.14             | 9          |
| log4j 1.2.15             | 0          |
| servlet-api 2.3          | 4          |
| servlet-api 2.5          | 1          |
| derby 10.1               | 6          |
| derby 10.2               | 1          |

[1] 砂田, 門田, 松本, "Javaソフトウェア開発における使用ライブラリの分析," ソフトウェア信頼性研究会 第8回ワークショップ, Nov. 2012.

[2] Mileva, Y. M., Dallmeier, V., Burger, M., Zeller, A.: Mining Trends of Library Usage, IWPSE2009, pp.57–62, 2009.

# プログラミング支援:コーディングパターン検出

- ライブラリのAPIの呼び出しパターンの検出[1]
- メソッド呼び出しパターンの検出[2]

例:jEditでは,

◆ openNodeScope/jjtreeOpenNodeScope/closeNodeScope/jjtreeCloseNodeScope

というパターンが55回出現している.

- 要素技術:シーケンシャルパターンマイニング
  - 要素列から頻出する部分列をパターンとして抽出する.
  - 部分列の個々の出現は, 順序が同一でなければならないが, 不連続なものでよい.

[1] T. Xie and J. Pei: MAPO: Mining API Usages from Open Source Repositories, MSR2006, pp.54–57, 2006.

[2] 石尾,伊達,三宅,井上, ”シーケンシャルパターンマイニングを用いたコーディングパターン抽出,” 情報処理学会論文誌, Vol.50, No.2, pp.860-871, 2009.

# プログラミング支援:ソフトウェア部品検索

- キーワードベースのソフトウェア部品検索エンジン
  - SPARS[1], Koders[2], Jarhoo[3]
- 入出力ベースの検索エンジン Prospector[4]
- コードクローンベースの検索エンジン[5]
- ランキング手法[6]

[1] SPARS Project, <http://sel.ist.osaka-u.ac.jp/SPARS/>

[2] Koders.com, <http://www.koders.com/>

[3] Jarhoo, <http://www.jarhoo.com/>

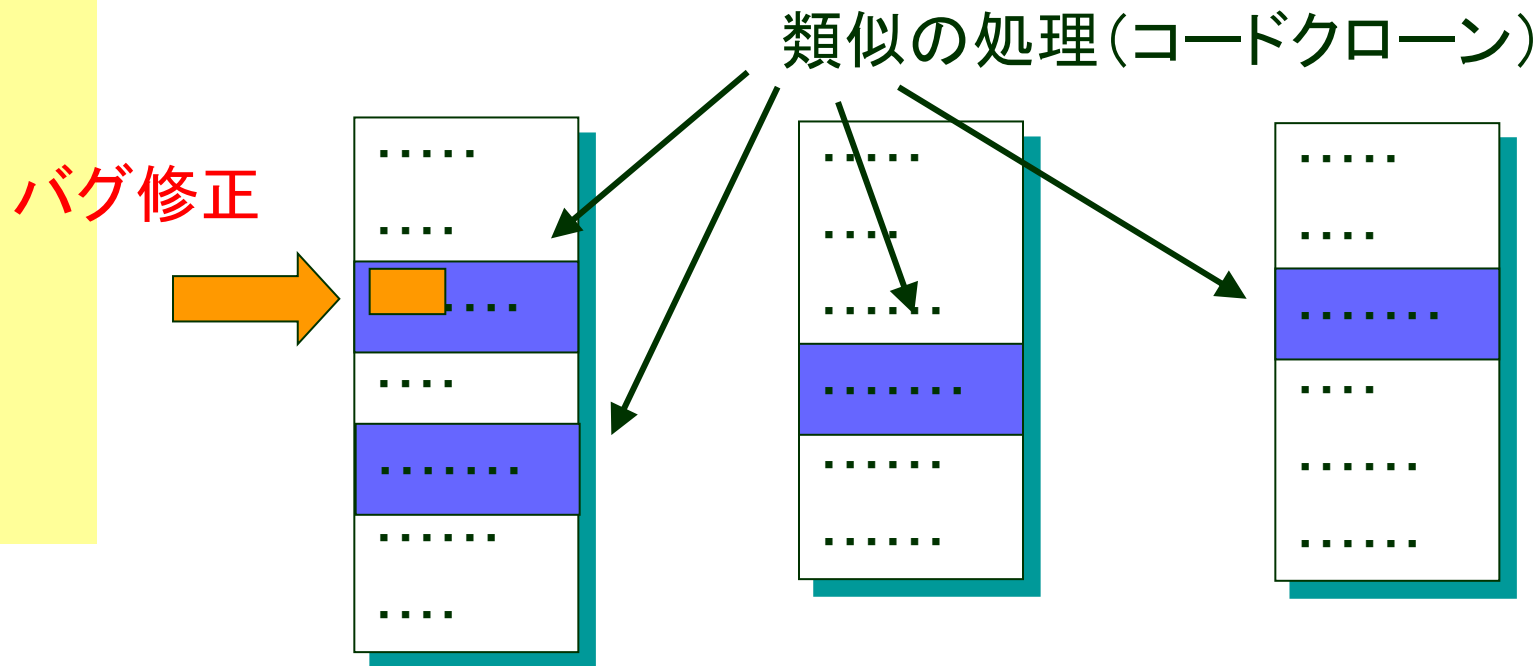
[4] D. Mandelin, L. Xu, R. Bodík, D. Kimelman, “Jungloid mining: helping to navigate the API jungle,” PLDI2005, pp.48-61, 2005.

[5] P. Xia, Y. Manabe, N. Yoshida, K. Inoue: Development of a code clone search tool for open source repositories, コンピュータソフトウェア, Vol.29, No.3, pp.181-187, 2012.

[6] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, S. Kusumoto: Ranking Significance of Software Components Based on Use Relations, IEEE Transactions on Software Engineering, Vol.31, No.3, pp.213–225, 2005.

# プログラミング支援:コードクローン分析

- Patchを当てるべきコードの発見 [1]
- 流用コードの発見 [2]

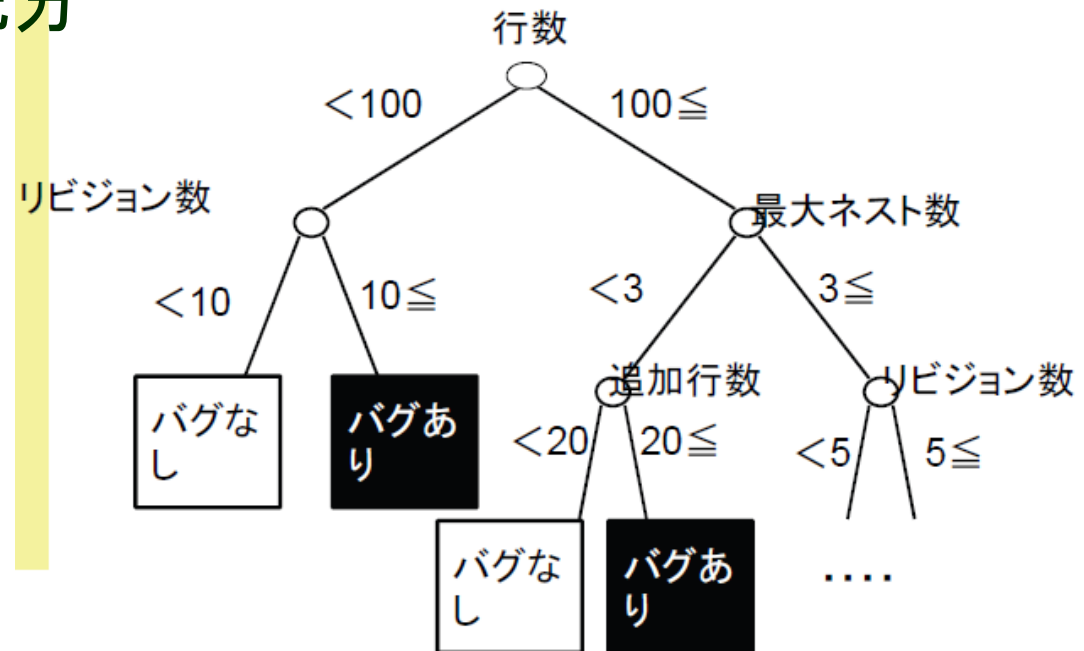


[1] Jang, J., Agrawal, A., and Brumley, D.: ReDe-Bug: Finding Unpatched Code Clones in Entire OS Distributions, Proc. 33rd Symposium on Security and Privacy (2012), pp.48-62.

[2] Monden, A., Okahara, S., Manabe, Y., and Matsumoto, K.: Guilty or Not Guilty: Using Clone Metrics to Determine Open Source Licensing Violations, IEEE Software, Vol.28, No.2 (2011), pp.42-47.

# テスト支援: バグモジュール予測

- バグを含むモジュール(機能, ソースファイル)の予測[1]
- 80%のバグが20%のコードに含まれている[2]
- テスト工数の最適な配分



[1] 畑秀明, 水野修, 菊野亨: 不具合予測に関するメトリクスについての研究論文の系統的レビュー, コンピュータソフトウェア, Vol.29, No.1 (2012), pp.106–117.

[2] Robert L. Glass, "Facts and Fallacies of Software Engineering," Addison-Wesley, 2003.

# 保守・デバッグ支援:バグの要因分析

## ■使用ライブラリに着目した分析[1]

- Eclipse.jdt.internal.compilerパッケージをimportしている71%のファイルは後にバグ修正されている.
- uiパッケージの場合は14%

## ■開発者とバグ混入の関係の分析[2]

- Eclipse3.0, 3.1, 3.2を分析
- どのバージョンでもバグ混入率が高い／低い開発者が存在する.
- 複数の開発者が変更したファイルはバグ混入率が高い.

[1] A. Schroter , Predicting Defects and Changes with Import Relations, MSR2007.

[2] まつ本, 亀井, 門田, 松本, ``開発者メトリックスに基づくソフトウェア信頼性の分析,<sup>10</sup>  
電子情報通信学会論文誌, Vol.J93-D, No.8, pp.1576-1589, August 2010.

# 保守・デバッグ支援: エキスパート推薦

- 特定のソースコードや, バグ修正依頼などに対し, 最も適切と思われる開発者を推薦する[1][2][3].
  - 過去の修正履歴をマイニングする.

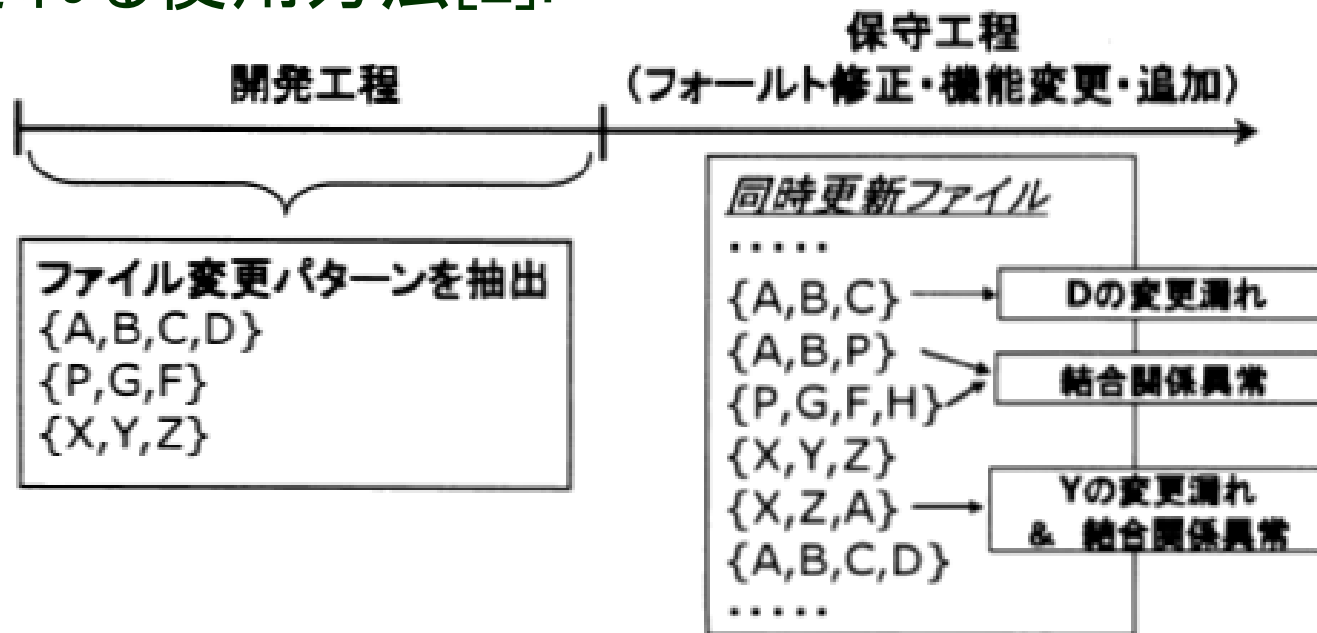
[1] J. Anvik, L. Hiew, G. C. Murphy, Who should fix this bug? ICSE'06, pp.361-370, 2006.

[2] H. Kagdi, D. Poshyvanyk: Who can help me with this change request?, ICPC 2009, pp.273–277, 2009.

[3] F. Servant, J.A. Jones: WhoseFault: Automatic Developer-to-Fault Assignment Through Fault Localization, ICSE2012, pp.36-46, 2012.

# 保守・デバッグ支援：同時変更コードの分析

- 同時に変更されることの多いソースファイル群を特定し、開発時の同時変更のし忘れを警告する[1].
- 想定される使用方法[2]:



[1] Fischer, M, Pinzger, M, Gall, H.: Populating a release history database from version control and bug tracking systems. ICSM'03 (2003), pp.23–32.

[2] 松村, 横森, 大杉, 川口, 松下, ファイルの同時変更パターンと変更差分の分析による論理的結合関係の自動抽出, ソフトウェアシンポジウム2005, 104-112, June 2005.

# 保守・デバッグ支援: Bug Localization

---

## ■故障の詳細な記述

□故障を再現させる動作, 故障が起こった機能の名称などを入力として, テキストマイニングにより, その原因となるソースコードを推定する[1].

[1] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed?. Proc. ICSE'12, pp.14-24, 2012.

# 保守・デバッグ支援：再修正バグの分析・予測

## ■再修正が必要となるバグ（デグレード，エンバグ）の要因分析・予測[1]

□Eclipseでは約16%のバグがreopenされた。

### ◆バグレポートの内容の影響が最も大きい。

- 再修正：“control”，“background”，“debugging”，“breakpoint”，“blocked” and “platforms”
- 問題なし：“verified”，“duplicate”，“screenshot”，“important”，“testing” and “warning”

### ◆修正者・報告者による差が見られる。

### ◆特定の時期に修正されたバグは再修正されやすい。

### ◆1日以内に修正されたバグの多くは問題ない。

[1] Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W.M., Ohira, M., Adams, B., Hassan, A. E. and Matsumoto, K.: Studying Re-Opened Bugs in Open Source Software, Empirical Software Engineering (available online Sep. 2012).

# 管理支援:ライセンスの分析

- ライセンスマッチングシステム [1]
  - ソースコードを入力として, OSSのライセンスを自動的に同定する.
- ライセンス自動分析サービス
  - Black Duck Software社のProtex[2]
    - 200,000以上のOSSプロダクトをベースにチェック
  - Palamida社のPalamida[3]
    - コードのマッチングによってライセンス違反を検出

[1] D. M. German, Y. Manabe, K. Inoue. A Sentence-Matching Method for Automatic License Identification of Source Code Files. ASE'10, pp. 437-446, 2010.

[2] [www.blackducksoftware.com/protex](http://www.blackducksoftware.com/protex)

[3] [www.palamida.com](http://www.palamida.com)

# 管理支援:コミュニケーション分析

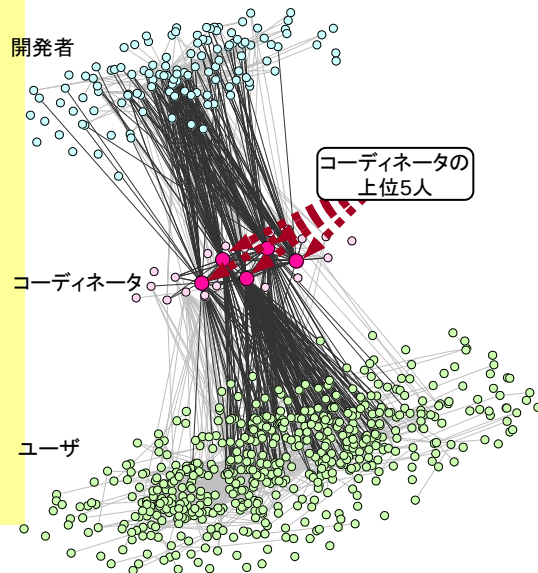
## ■E-mailに基づくソーシャルネットワーク分析 [1]



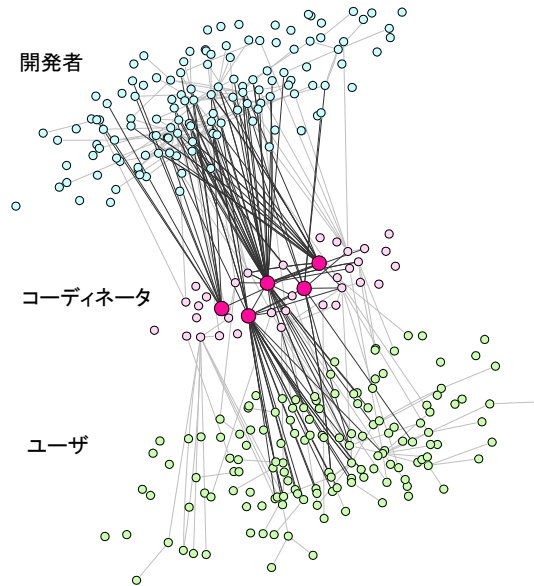
[1] C. Bird, A. Gourley, P. Devanbu, M. Gertz, A. Swaminathan, Mining e-mail social network, MSR2006.

# 管理支援:コミュニケーション分析

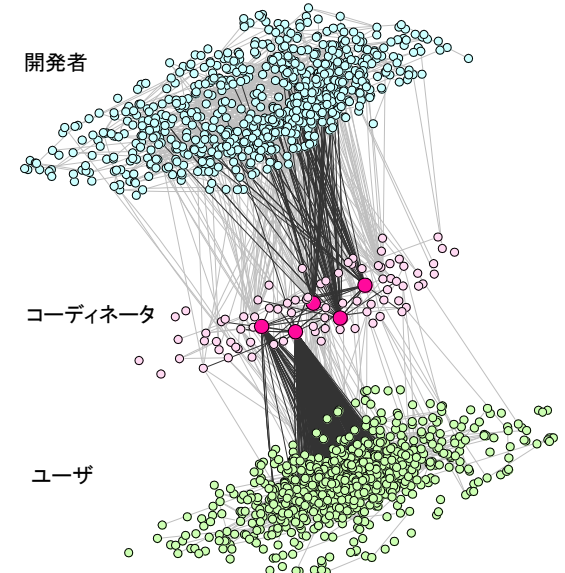
- 開発者／ユーザ・コミュニティの活動メトリクス [1]
- コーディネーターの活動の分析 [2]



Apache Ver.2.2.0リリース時(2005年11月)



GIMP Ver.1.2.0リリース時(2000年10月)



Netscape Ver.6リリース時(2000年11月)

コーディネータ 企業のプロジェクトにおけるプロジェクトマネージャ、チームリーダーに相当

- [1] H. Zhong, Y. Yang, and J. Keung. Assessing the representativeness of open source projects in empirical software engineering studies. Proc. APSEC'12, pp. 808-817, 2012.
- [2] S. Matsumoto, Y. Kamei, M. Ohira, and K. Matsumoto. A comparison study on the coordination between developers and users in FOSS communities. STC'08, pp.1-9, 2008.

# OSSユーザ支援：OSSの品質，成熟度の評価

## ■ QualiPSo (Quality Platform for OSS) project [1]

■ ユーロッパを中心としたOSS評価プロジェクト

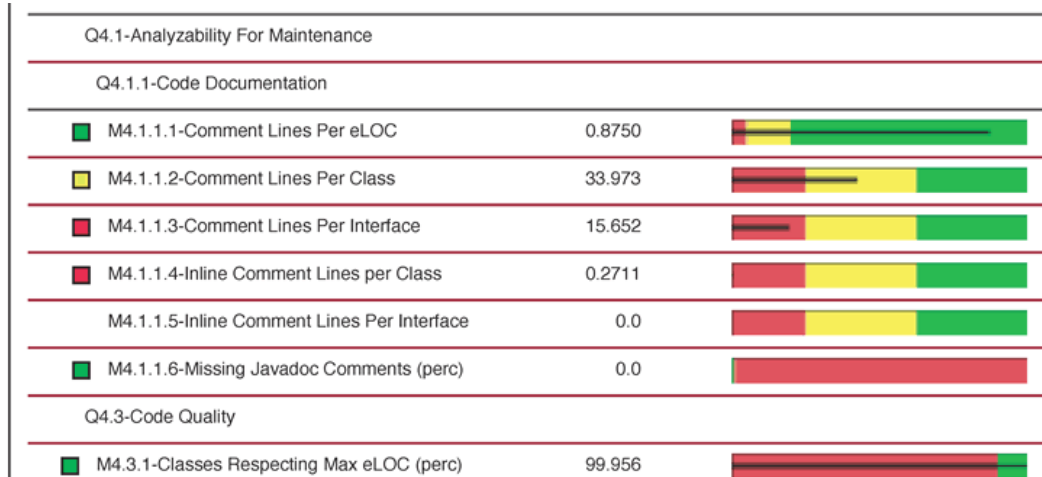
■ 2006～2010年

■ 法的問題，ビジネスモデル，情報管理，相互接続性，信頼性評価，プロセス評価などを扱う。

## ■ 評価の例

□ リポジトリ統計計測

□ コードメトリクス計測[2]



[1] <http://www.qualipso.org/>

[2] <http://www.ipa.go.jp/about/toolcatalog/osc/osc-4-3.htm>

# OSSユーザ支援：修正されそうなバグの予測

- OSSの不具合は全て修正されるわけではない.
  - Firefoxの場合, 全報告バグの15%しか修正されない.
- 次期リリースまでに修正されそうなバグを予測する[1]
  - 不具合情報メトリクス
    - ◆ 不具合が発見された機能, 環境, コメント数
  - 状態メトリクス
    - ◆ 不具合の状態(審査, 修正, 検証, 保留), 修正担当者など
  - 時期メトリクス: 経過日数など
  - 参加者メトリクス
    - ◆ 報告者, 修正割り当て担当者, 修正者

[1] A. Ihara, Y. Kamei, A. Monden, M. Ohira, J. Keung, N. Ubayashi, and K. Matsumoto, ``An Investigation on Software Bug Fix Prediction for Open Source Software Projects -A Case Study on the Eclipse Project -, " Proc. SATA 2012.

# OSSユーザ支援：問題の解決策の検索

- OSSのオンラインフォーラムから解決策を検索する.
- Gottipatiら[1]は, 自然言語で記述された問題文を入力とするセマンティック検索エンジンフレームワークを提案している.
  - 従来のキーワードベースの検索エンジンと比べて大幅に検索精度が改善したことを示している.

# リポジトリマイニングの 要素技術

# リポジトリマイニングの主要素技術

- **機械学習**：決定木，ランダムフォレスト，サポートベクターマシン...
- **ルール発見**：アソシエーション分析
- **テキストマイニング**：潜在意味解析(LSA)，潜在的ディリクレ配分(LDA)
- **可視化**：平行座標プロット，多次元尺度構成法，自己組織化マップ
- **静的解析**：ソースコードメトリクス
- **動的解析**：シーケンシャルパターンマイニング

# 紹介する要素技術

---

- 機械学習によるバグモジュール予測
- アソシエーションルールマイニング

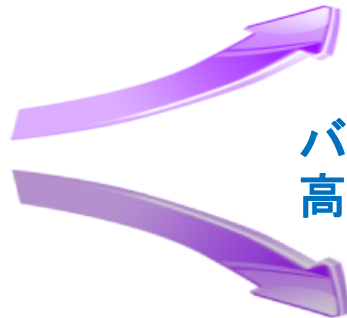
# バグモジュール予測

- バグを含んでいる可能性の高いモジュールを予測する.

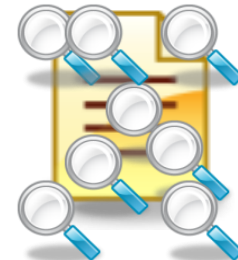
ソフトウェア  
モジュール群



各モジュールの特徴(メトリクス)  
・変更回数, 規模, 開発者  
・最大ネストの深さ など



バグを含む可能性  
高い



重点的にテストを行う



バグを含む可能性  
低い

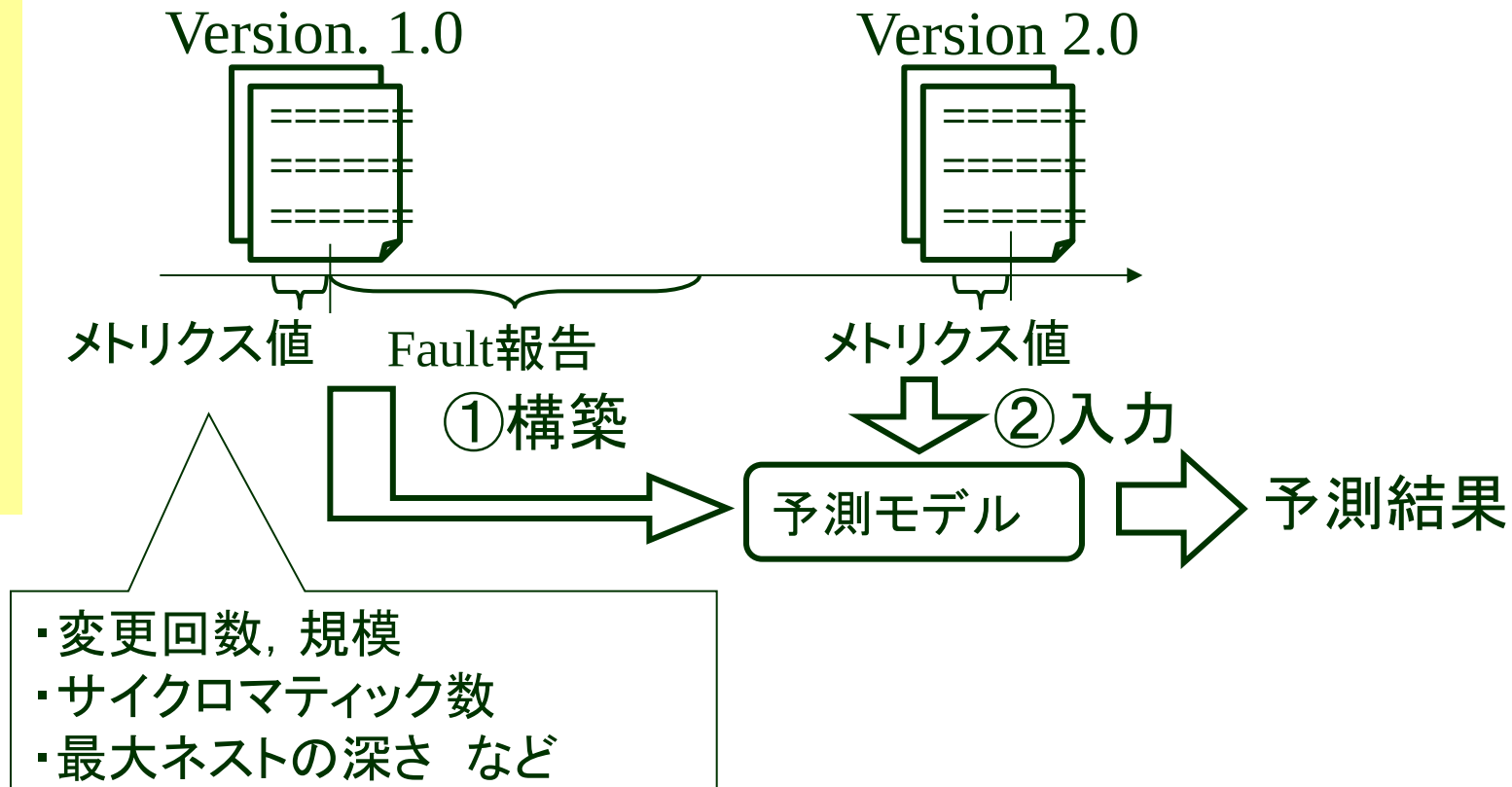


テストの優先度を下げる.

少ないテスト工数で高い品質の確保が期待できる.

# モデル構築と予測

- 旧バージョンの開発実績データからモデルを構築し、次期バージョンにおけるバグモジュールを予測する。



# 企業における事例

- Ericsson Telecom AB (1996)[1] 保守工程
- NTT DATA (1999)[2] 保守工程
- ABB Inc. (2006)[3] 結合・システムテスト工程
- Microsoft (2006)[4] 出荷後のバグ
- NEC (2010)[5] 結合テスト工程
- NTT西日本(2012)[6] 受け入れテスト工程

[1] Niclas Ohlsson, Hans Alberg, Predicting Fault-Prone Software Modules in Telephone Switches, IEEE Trans. on Software Engineering, pp.886-894, Dec. 1996.

[2] 佐藤 慎一, 端山 毅, 門田 暁人, 松本 健一, ``ソフトウェア複雑度メトリクスに基づく大規模システムの欠陥モジュール分類とその評価," 第19回ソフトウェア生産における品質管理シンポジウム報告集, September 1999.

[3] P.L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and Results from Initiating Field Defect Prediction and Product Test Prioritization Efforts at ABB Inc," In Proceedings of 28th International Conference on Software Engineering, pp.413-422, Shanghai, China, 2006.

[4] N. Nagappan, T. Ball, A. Zeller, "Mining Metrics to Predict Component Failures", In Proceedings of 28th International Conference on Software Engineering, pp.452-?, Shanghai, China, 2006.

[5] 瀬瀬伸子, 川村真弥, 野村准一, 野中誠, "プロセスおよびプロダクトメトリクスを用いたFault-Proneクラス予測の適用事例", 情報処理学会研究報告, ソフトウェア工学研究会, Vol.2010-SE-168, No.6 (2010年6月)

[6] 林卓磨, 信田祥司, 白井 久美子, 吉田順一, 門田暁人, 松本健一, ``Fault-Proneモジュール判別モデルを用いた検証項目数削減に関する一検討," 2011年電子情報通信学会ソサイエティ大会通信講演論文集, Vol.2, No.B-18-2, pp.406, 2011.

# 予測モデル

---

- 線形重回帰モデル ←このあと説明する.
- 線形判別モデル
- ロジスティック回帰モデル
- 分類木(決定木, 回帰木)
- ニューラルネットワーク
- ベイジアンネットワーク
- SVM(サポートベクターマシン)
- ランダムフォレスト(2001年登場) ←最有力

# 線形重回帰モデルによるモジュールバグ数の予測

|        | 設計書<br>ページ数 | ソースコード<br>変更行数 | 最大ネストレ<br>ベル | バグ数 |
|--------|-------------|----------------|--------------|-----|
| モジュールA | 45          | 2012           | 6            | 2   |
| モジュールB | 55          | 330            | 5            | 0   |
| モジュールC | 10          | 1405           | 4            | 1   |
| ...    | ...         | ...            | ...          | ... |



重回帰分析

モデル式:

$$\text{バグ数} = \text{設計書ページ数} \times 0.02 + \text{変更規模} \times 0.001 + \text{最大ネストレベル} \times 0.05 - 1.0$$



バグ数の予測

|        | 設計書<br>ページ数 | ソースコード<br>変更行数 | 最大ネストレ<br>ベル | バグ数 |
|--------|-------------|----------------|--------------|-----|
| モジュールX | 62          | 2348           | 6            | ?   |

# 重回帰分析(線形回帰モデル, 重回帰モデル)

## ■モデル式

$$\hat{Y} = a_1 N_1 + a_2 N_2 + \cdots + a_k N_k + C$$

$\hat{Y}$  : 目的変数の予測値

$N_j$  : 説明変数

$a_j$  : 係数(偏回帰係数)

$C$  : 定数項

実測値  $Y$  と予測値  $\hat{Y}$  の差を残差と呼ぶ.  
残差の2乗和が最小となるように  $a_j$  と  $C$  を定める.

モデル式の例:

開発工数 = 設計工数 $\times$ 0.02 + 基本設計バグ数 $\times$ 0.001 + 画面数 $\times$ 0.05 - 1.0

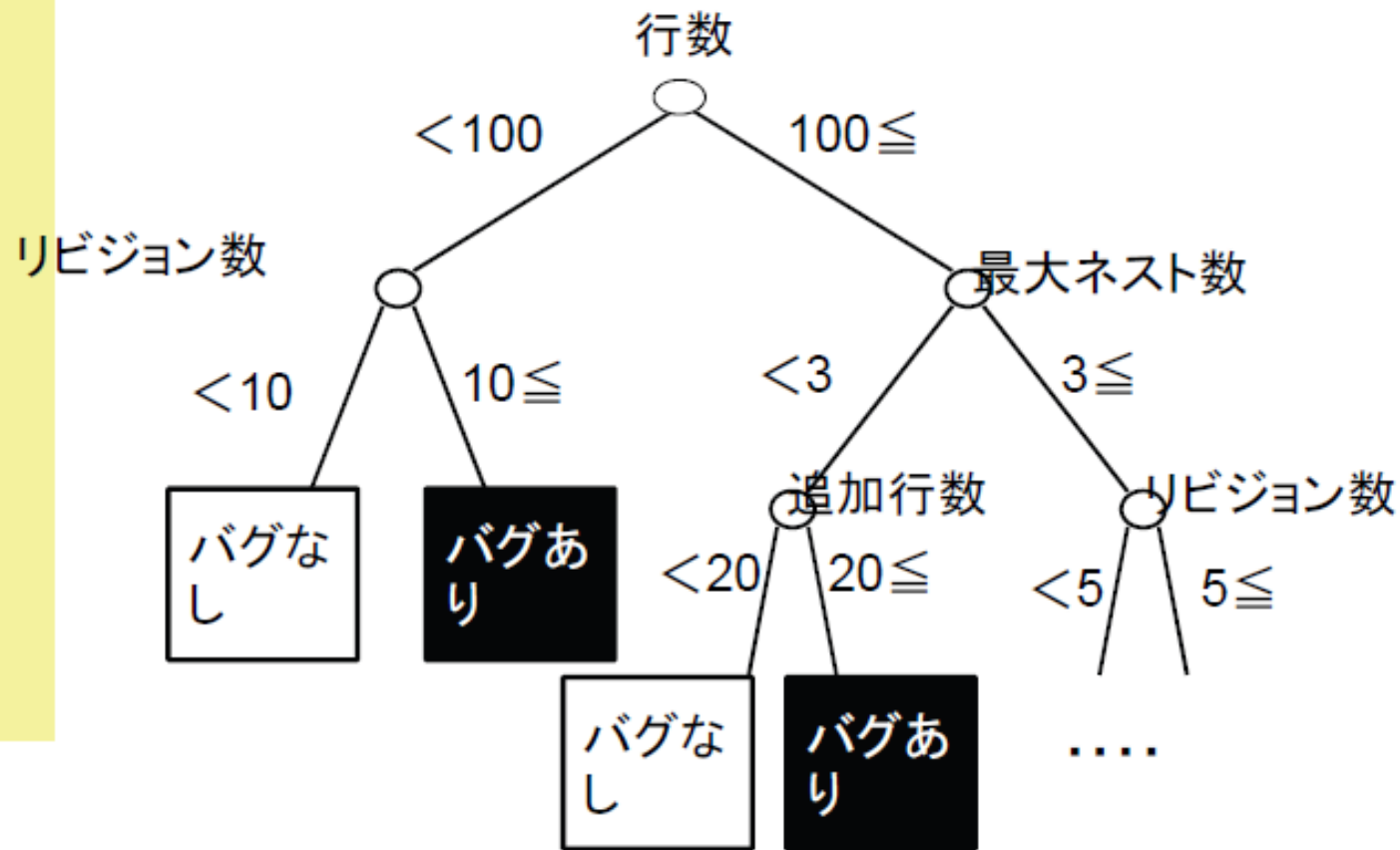
# ランダムフォレスト(Random Forest)

- 2001年にLeo Breiman[1]によって提案された.
  - “Random” Samplingを使い, 大量の”Tree”(決定木, 回帰木)を構築するので「Random Forest」.
- 集団学習を用いた手法の1つ
  - 低性能なモデルを多数組み合わせることで, 高性能な予測モデルを構築することを目指す.
- 統計パッケージ「R」で実施可能[2]

[1] Breiman, L.: Random Forests, Machine Learning, Vol.45, No.1, pp.5–32 (2001).

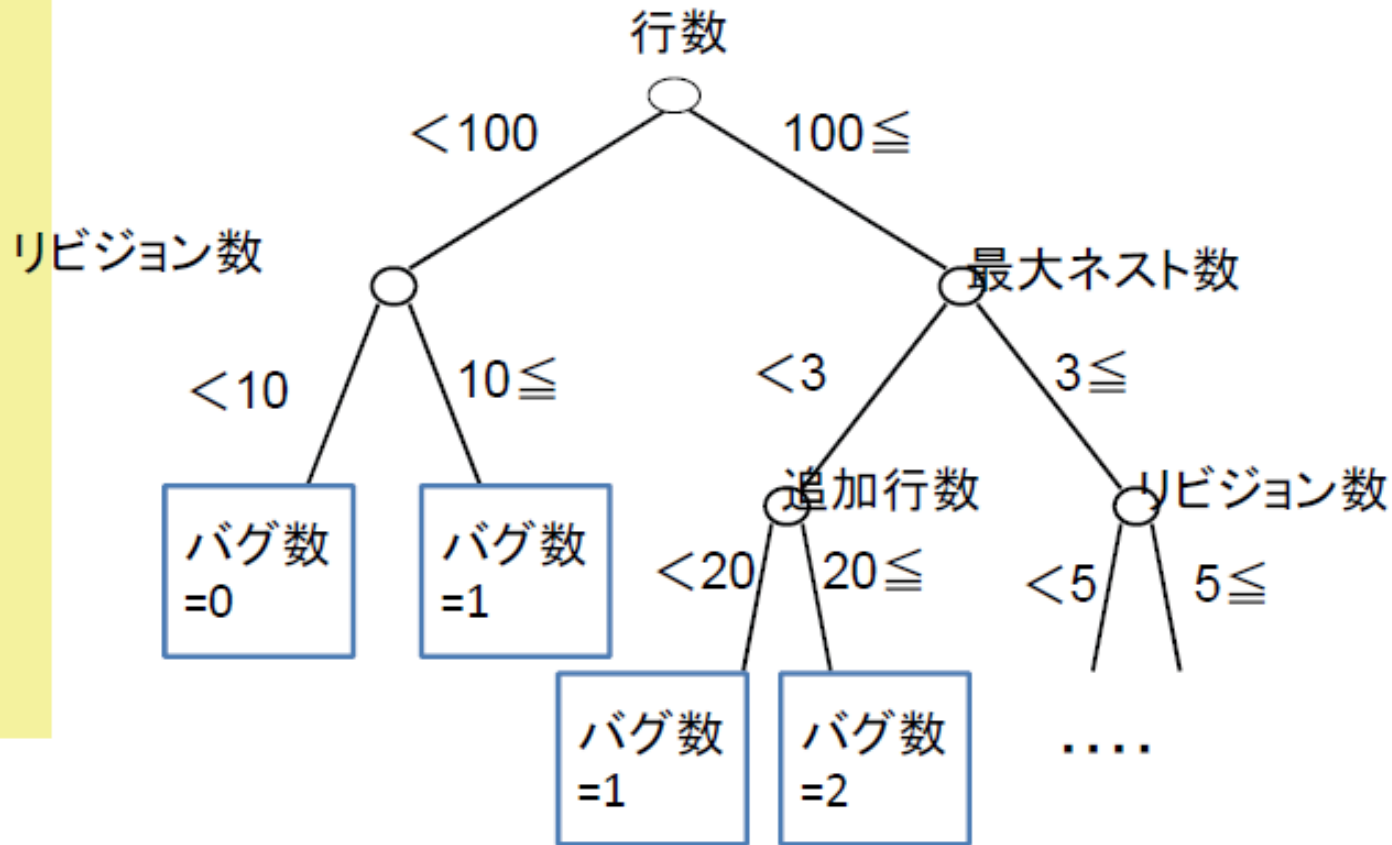
[2] [http://cran.r-project.org/doc/Rnews/Rnews\\_2002-3.pdf](http://cran.r-project.org/doc/Rnews/Rnews_2002-3.pdf)

# 決定木 (Decision Tree) ー 分類木



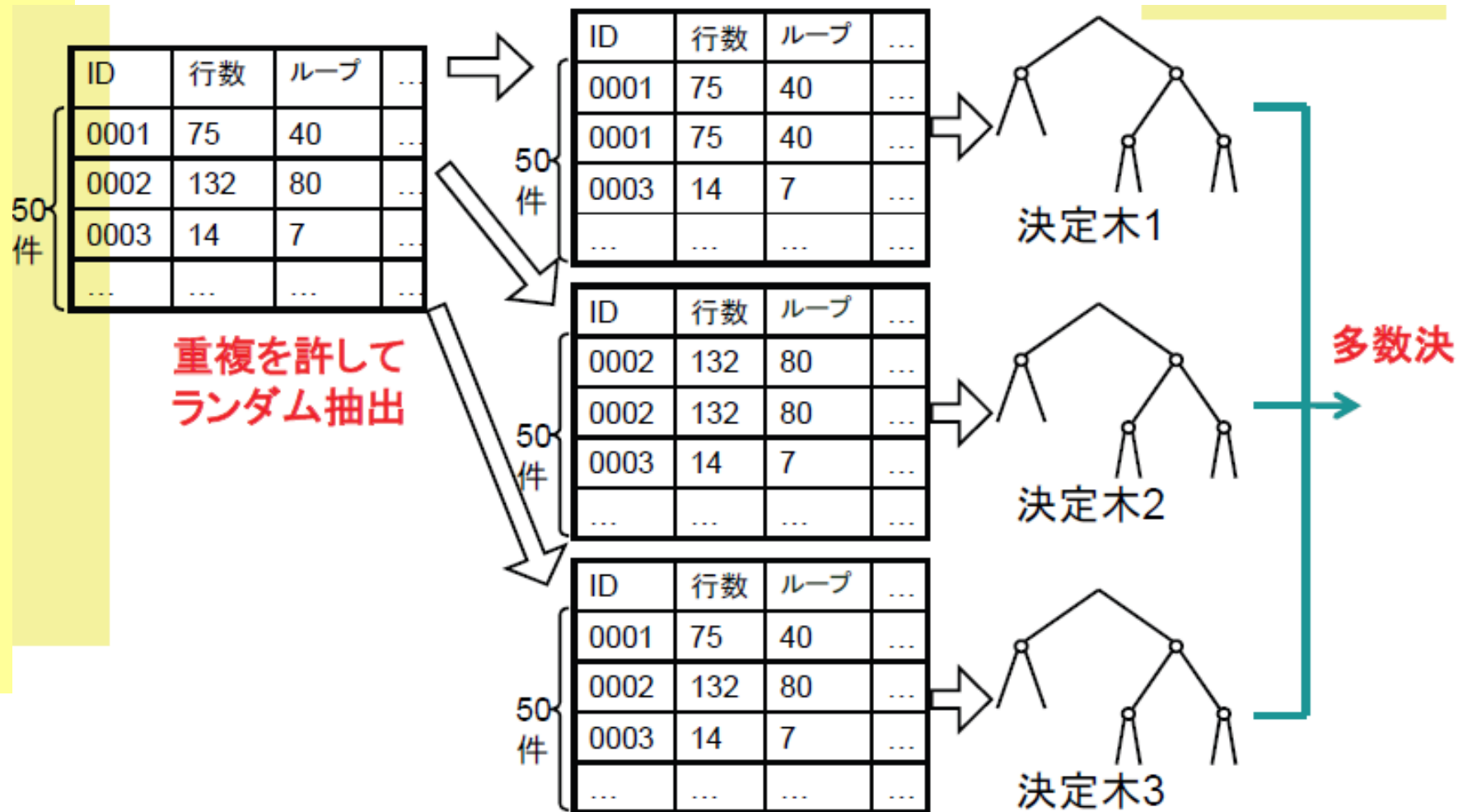
- 分かりやすい.
- ただし, 精度はかならずしも高くない.

# 決定木 (Decision Tree) 一回帰木



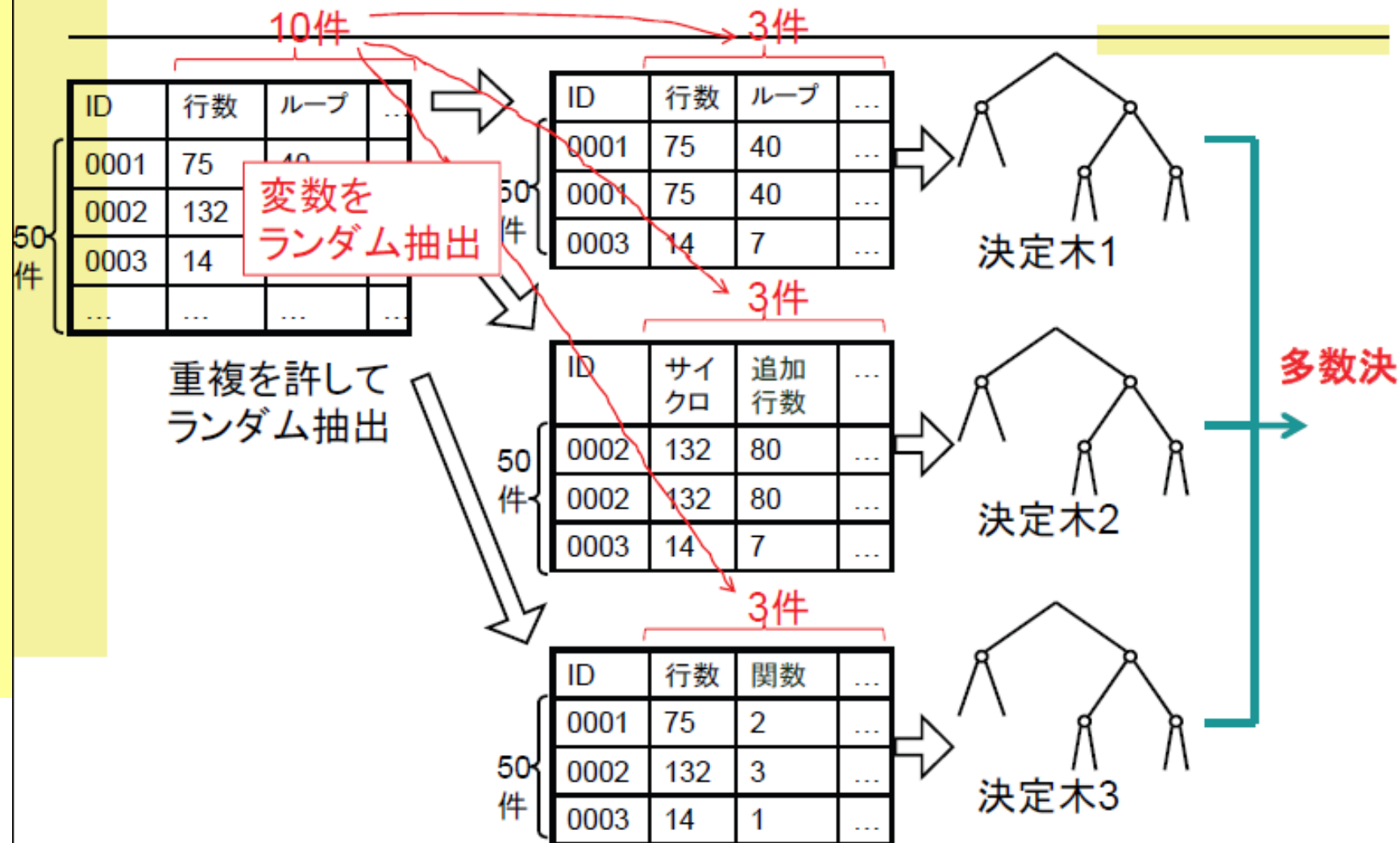
- 分かりやすい.
- ただし, 精度はかならずしも高くない.

# 集団学習 (Bagging)



■モデルの汎化能力が高まる。(オーバーフィッティングににくい)

# ランダムフォレスト



■モデルの汎化能力がさらに高まる。

# Rによるランダムフォレスト

```
library(randomForest)
fitdata = read.table("学習データ.csv", header = TRUE, sep = ",", quote="¥",
dec=".", fill = TRUE)
testdata = read.table("評価データ.csv", header = TRUE, sep = ",", quote="¥",
dec=".", fill = TRUE)

model <- randomForest(BUG~., data=fitdata)
result <- predict(model, testdata, type="response")
```

- ライブラリが用意されている.
- モデル構築, 予測がそれぞれ1行で書ける.

# バグモジュール予測を行うにあたっての検討事項

---

## ■ 準備

- 予測の粒度

- 説明変数の候補, 計測ツール開発

## ■ 評価尺度

## ■ データの準備

## ■ モデル構築

## ■ テスト工数割り当てのシミュレーション

# 予測の粒度(モジュール)

## ■ 予測するモジュールの粒度の候補

- ソースコードの1ファイル
- C言語の関数, JavaやC++の1クラス
- Javaなどの1パッケージ
- 機能
- サブシステム

## ■ 一つの考え方: ファイルや関数では細かすぎる. サブシステムでは大きすぎる.

### □ モジュール=機能とする.

- ◆ 受入れテスト時にバグ位置が特定できる.
- ◆ 受入テスト時にテストの優先順位付けができる.
- ◆ 予測の効果が見込める.

# 説明変数の例

## ■機能単位で計測可能なメトリクス

### □ソースコードメトリクス

- ◆ 規模(SLOC)
- ◆ サイクロマティック数
- ◆ 最大ネストレベル など

### □設計メトリクス

- ◆ 設計書ページ数
- ◆ 設計書の図式表現のノード数
- ◆ 設計書の改訂回数 など

### □その他

- ◆ レビュー, テストの情報

## ■差分メトリクス(変更メトリクス)

### □前バージョン(フェーズ)からの差分の値:

- ◆ 例: 追加行数, 変更行数, 削除行数

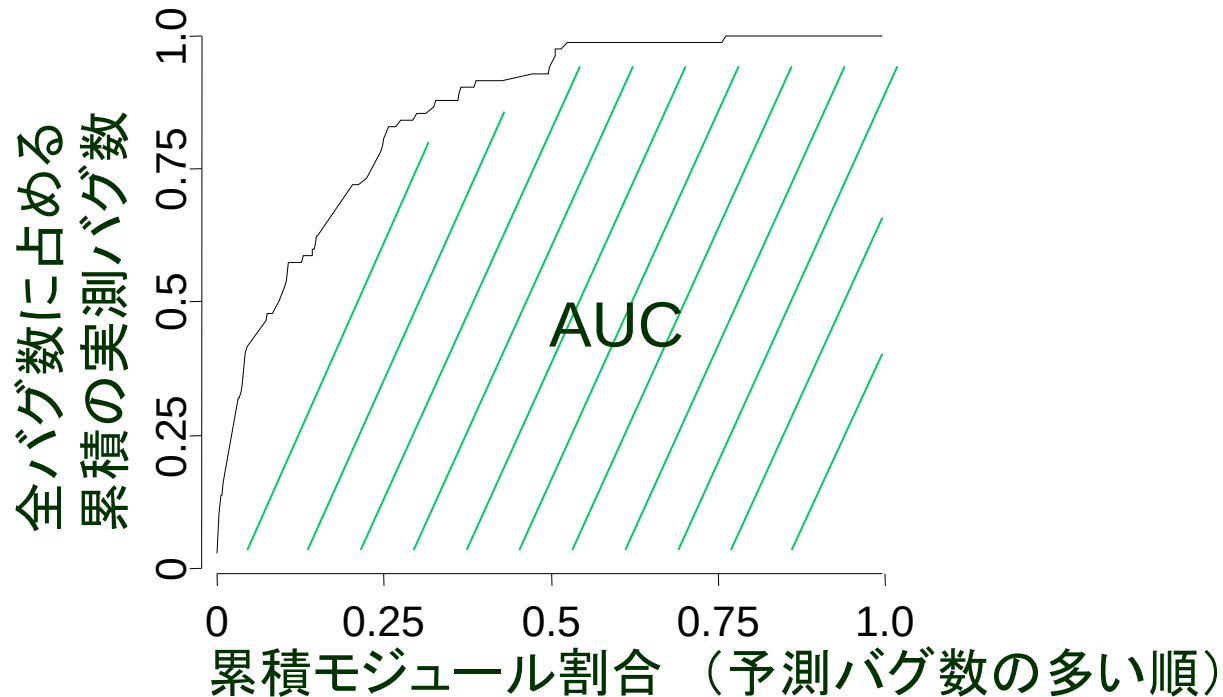
# 評価尺度: Alberg DiagramのAUC

## ■ Alberg Diagram

□ X軸: 予測バグ数の多い順にモジュールを並べたときの,  
累積モジュール数 ÷ 全モジュール数

□ Y軸: 累積の検出バグ数 ÷ 全バグ数

■ AUC: 曲線の下側の面積



AUC=1に近くなるほど予測精度が高い. 目標AUC=0.7~0.8

# 予測した後は・・・

## ■テスト工数の割り当て戦略[1]

1. (基本) 各モジュールの予測バグ数に比例させる.
2. (テスト工数の制約がきついついとき) 予測バグ密度に比例させる.
3. (確実にバグを検出したいとき) 予測バグ数  $\times$  規模に比例させる.
4. (3.の改良版) 予測バグ数  $\times \log(\text{規模})$  に比例させる.
5. (余裕のあるとき) 予測せずに規模に比例して割り当てる.

## ■予測の効果の見積もり

□シミュレーションを行う.

[1]中野 大輔, 門田 暁人, 松本 健一, ``バグモジュール予測を用いたテスト工数割り当て戦略,"  
ソフトウェア工学の基礎ワークショップFOSE2012.

# シミュレーションの仮定

## ■ バグ発見率モデル

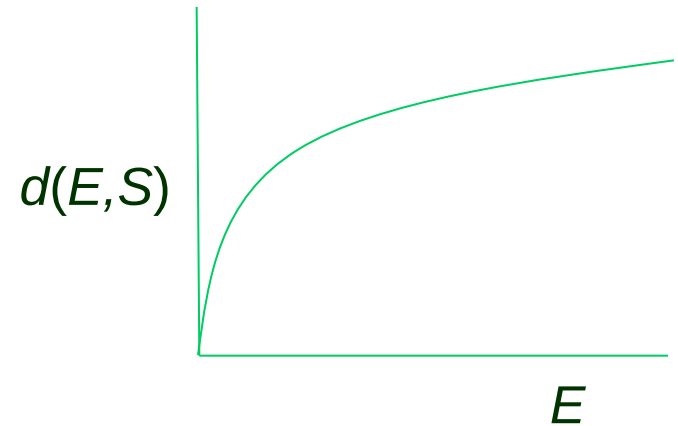
$$d(E, S) = 1 - e^{-\frac{ZE}{S}}$$

$d(E, S)$  : バグ発見率

$E$  : テスト工数

$S$  : モジュール規模(または複雑度)

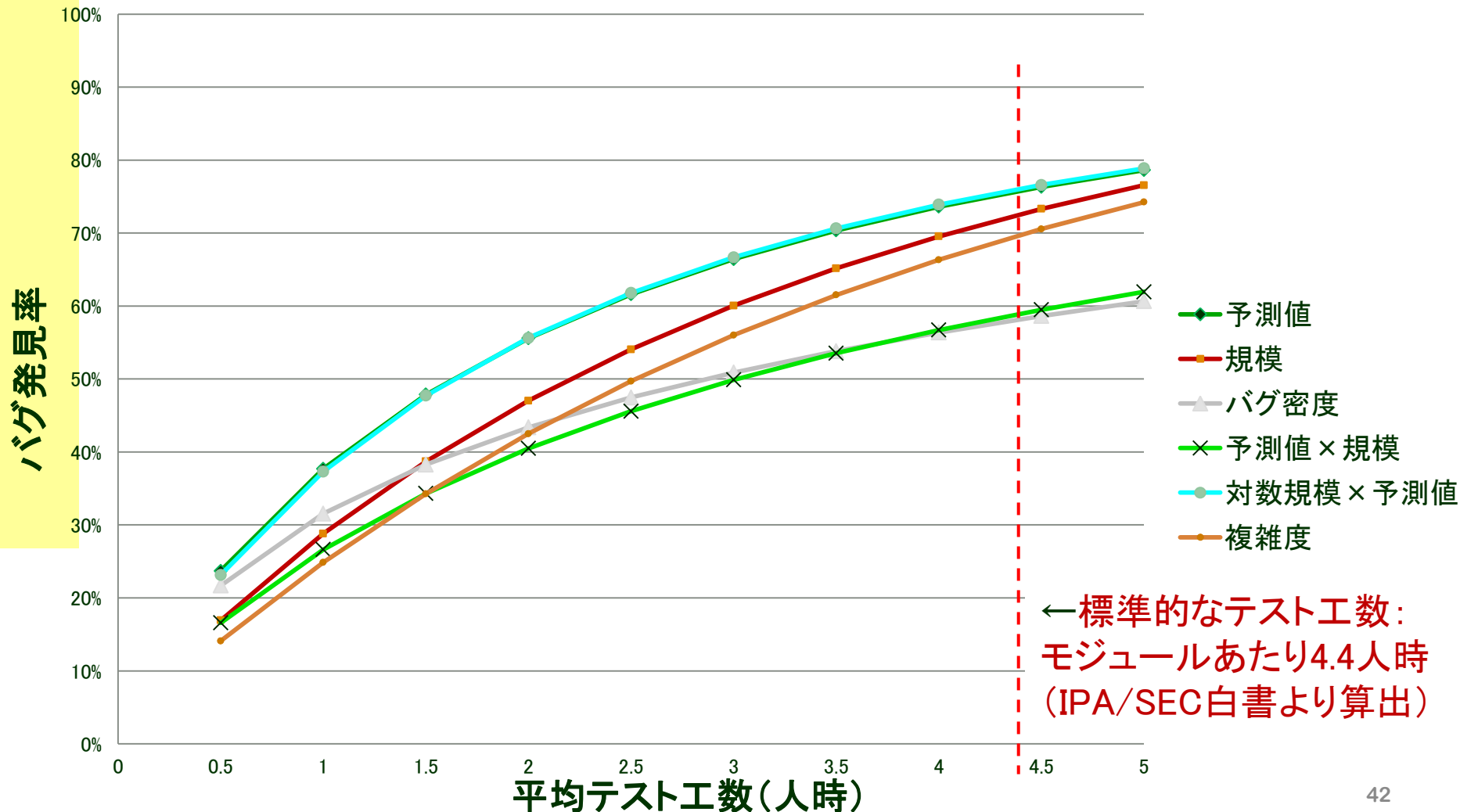
$Z$  : 単位工数当たりのfault発見率を決定する定数



- 費やしたテスト工数に応じて指数的にバグ発見率が増加するものと仮定する.
- ただし, 規模に反比例する.

# シミュレーションの事例(Eclipse)

各テスト戦略でのバグ発見率



# 紹介する要素技術

---

- 機械学習によるバグモジュール予測
- アソシエーションルールマイニング

# アソシエーションルールマイニング

- 大量のデータの中から、事象間の強い関係(アソシエーションルール)を発見する手法である.
  - アソシエーションルール:  $X \Rightarrow Y$ 
    - ある事象Xが発生するならばある事象Yも高い確率で発生する.
  - 例
    - 休日に「おにぎり」と「お茶」を同時に買う顧客は「レジャーシート」も買っている。
- 曜日=土日 & おにぎり & お茶  $\Rightarrow$  レジャーシート
- 休日には、レジャーシートの配置をおにぎりかお茶に近づけ、発見率、併せ買い率を上げる。

# ソフトウェア開発データリポジトリの場合

## ソフトウェア開発データリポジトリ

|             | 開発種別 | アーキテクチャ | 規模<br>(FP) | 試験工数比率 |
|-------------|------|---------|------------|--------|
| 過去プロジェクト001 | 新規   | 3階層CS   | 100未満      | 20%未満  |
| 過去プロジェクト002 | 拡張   | スタンドアロン | 100～500    | 40～50% |
| ....        | ...  | ...     | ...        | ...    |
| 過去プロジェクト500 | 拡張   | 2階層CS   | 100～500    | 20～30% |

↓ アソシエーション分析

### アソシエーションルール

開発種別＝拡張 **and** アーキテクチャ＝3階層CS  
⇒ 試験工数比率＝40～50%

# アソシエーションルールに関する指標

- アソシエーションルール  $X \Rightarrow Y$  に対して,

$$\text{支持度} = \frac{X \text{と} Y \text{が同時に出現したケース数}}{\text{全ケース数}}$$

支持度が大きいほど、よく起こる事象を表したルールである.

$$\text{信頼度} = \frac{X \text{の発生時に} Y \text{が出現したケース数}}{X \text{が発生したケース数}}$$

$$\text{リフト値} = \frac{X \Rightarrow Y \text{の信頼度}}{\text{全ケースにおける} Y \text{の発生割合}}$$

信頼度とリフト値は、前提 $X$ と結論 $Y$ の関連の強さを表す指標である.

# アソシエーションルール適用の前提

- 全ての変数は、質的データ(カテゴリ変数)である.
  - 数値変数はあらかじめカテゴリ変数に変換しておく.
  - 欠損値があってもよい.

## 分析データのイメージ

| 業種  | アーキテクチャ | 主開発言語        | 規模<br>(FP) | 工期<br>(月数) | 平均要員数 | 外部委託率 | 生産性<br>(FP÷人時) |
|-----|---------|--------------|------------|------------|-------|-------|----------------|
| 銀行  | C/S     | PL/I         | 上位         | 上位         | 上位    | 上位    | 下位             |
|     | スタンドアロン | C            | 下位         | 中位         | 中位    | 下位    | 下位             |
| 銀行  | C/S     | COBOL        | 下位         | 中位         | 下位    | 下位    | 下位             |
| 銀行  |         | Visual Basic | 中位         |            | 中位    | 中位    | 中位             |
| 製造業 | スタンドアロン |              | 中位         | 下位         | 中位    | 中位    | 中位             |
| 銀行  | 混合      | C++          |            | 下位         | 上位    | 上位    | 上位             |
| 銀行  | C/S     | COBOL        | 上位         | 下位         | 下位    | 下位    | 上位             |
| 公共  | 混合      | Java         | 中位         | 上位         |       | 下位    | 上位             |

# 量的アソシエーションルール分析[1]

## ■結論部に量的変数を指定できる.

### バグ管理表データ分析の例

- (機能=A) & (検出部門=専任テスト部門) ⇒ (修正日数)  
平均 20.2    標準偏差 15.1
- (機能=B) & (検出部門=開発部門) ⇒ (修正日数)  
平均 8.5    標準偏差 5.3

## ■修正日数などの平均値やばらつきが大きくなるルールが抽出できる. 例外ルールも抽出可能

## ■数社での適用実績あり

# 例外ルール抽出

---

## ■常識ルール

□開発言語＝COBOL ⇒ 生産性＝低い

## ■例外ルール

常識ルールにある条件が加わることで、結論が逆となる.

□開発言語＝COBOL & アーキテクチャ＝スタンドアロン ⇒ 生産性＝高い

# ルール削減

## ■ 一般的なルール抽出条件

- 条件の最大組み合わせ数=3
- 最低支持率0.01, 最低信頼度0.8

## ■ データセットにもよるが, 1000以上のルールが出る.

## ■ ルール削減

1.  $A \Rightarrow X$       信頼度0.80
2.  $B \& C \Rightarrow X$     信頼度0.81      ...削減
3.  $D \& E \Rightarrow X$     信頼度0.90      ...残す
4.  $A \& B \Rightarrow X$     信頼度0.90      ...削減

## ■ 事例(次ページ)

- 1347のルール → 13ルールにまで減らした(予測能力を損なわずに)

# バグモジュールのルール化[1]

| No. | Support | Confidence | Rule |   |   |   |     |   |   |   |        |   |                      |
|-----|---------|------------|------|---|---|---|-----|---|---|---|--------|---|----------------------|
| 1   | 0.180   | 0.760      | NOF  | = | H | & | TPC | = | H | → | Faulty |   |                      |
| 2   | 0.152   | 0.800      | NBD  | = | H | & | TPC | = | H | → | Faulty |   |                      |
| 3   | 0.150   | 0.789      | FOUT | = | H | & | TPC | = | H | → | Faulty |   |                      |
| 4   | 0.148   | 0.763      | NSF  | = | H | & | TPC | = | H | → | Faulty |   |                      |
| 5   | 0.139   | 0.821      | VG   | = | H | & | TPC | = | H | → | Faulty |   |                      |
| 6   | 0.134   | 0.797      | PAR  | = | H | & | TPC | = | H | → | Faulty |   |                      |
| 7   | 0.128   | 0.799      | NOM  | = | H | & | TPC | = | H | → | Faulty |   |                      |
| 8   | 0.084   | 0.851      | NOT  | = | H | & | TPC | = | H | → | Faulty |   |                      |
| 9   | 0.016   | 0.889      | NOM  | = | M | & | NSM | = | H | & | TPC    | = | H → Faulty           |
| 10  | 0.014   | 0.824      | FOUT | = | H | & | NSM | = | H | & | TPC    | = | M → Faulty           |
| 11  | 0.014   | 0.824      | FOUT | = | H | & | NOF | = | L | & | NOM    | = | H → Faulty           |
| 12  | 0.018   | 0.900      | NBD  | = | M | & | NOM | = | L | & | NSF    | = | H & NSM = L → Faulty |
| 13  | 0.011   | 0.917      | VG   | = | H | & | NOF | = | H | & | NOM    | = | M & NOT = H → Faulty |

[1] A. Monden, J. Keung, S. Morisaki, Y. Kamei, K. Matsumoto, "A Heuristic Rule Reduction Approach to<sup>51</sup> Software Fault-proneness Prediction," Proc. of Asia-Pacific Software Engineering Conference (APSEC 2012)

# バグモジュールのルール化

| No. | Support | Confidence | D    |   |   |   |     |   |   |   |        |                        |
|-----|---------|------------|------|---|---|---|-----|---|---|---|--------|------------------------|
| 1   | 0.180   | 0.760      | NOF  | = | H | & | TPC | = | H | → | Faulty |                        |
| 2   | 0.152   | 0.800      | NBD  | = | H | & | TPC | = | H | → | Faulty |                        |
| 3   | 0.150   | 0.789      | FOUT | = | H | & | TPC | = | H | → | Faulty |                        |
| 4   | 0.148   | 0.763      | NSF  | = | H | & | TPC | = | H | → | Faulty |                        |
| 5   | 0.139   | 0.821      | VG   | = | H | & | TPC | = | H | → | Faulty |                        |
| 6   | 0.134   | 0.797      | PAR  | = | H | & | TPC | = | H | → | Faulty |                        |
| 7   | 0.128   | 0.799      | NOM  | = | H | & | TPC | = | H | → | Faulty |                        |
| 8   | 0.084   | 0.851      | NOT  | = | H | & | TPC | = | H | → | Faulty |                        |
| 9   | 0.016   | 0.889      | NOM  | = | M | & | NSM | = | H | & | TPC    | = H → Faulty           |
| 10  | 0.014   | 0.824      | FOUT | = | H | & | NSM | = | H | & | TPC    | = M → Faulty           |
| 11  | 0.014   | 0.824      | FOUT | = | H | & | NOF | = | L | & | NOM    | = H → Faulty           |
| 12  | 0.018   | 0.900      | NBD  | = | M | & | NOM | = | L | & | NSF    | = H & NSM = L → Faulty |
| 13  | 0.011   | 0.917      | VG   | = | H | & | NOF | = | H | & | NOM    | = M & NOT = H → Faulty |

リビジョン数  
 $\geq 3$

# バグモジュールのルール化

|    | Success | Failure | Condition                             | Result |
|----|---------|---------|---------------------------------------|--------|
| 1  | 0.100   | 0.760   | NOF = H & TPC = H                     | Faulty |
| 2  | 0.152   | 0.800   | NBD = H & TPC = H                     | Faulty |
| 3  | 0.789   | 0.789   | FOUT = H & TPC = H                    | Faulty |
| 4  | 0.763   | 0.763   | NSF = H & TPC = H                     | Faulty |
| 5  | 0.139   | 0.821   | VG = H & TPC = H                      | Faulty |
| 6  | 0.797   | 0.797   | PAR = H & TPC = H                     | Faulty |
| 7  | 0.799   | 0.799   | NOM = H & TPC = H                     | Faulty |
| 8  | 0.084   | 0.851   | NOT = H & TPC = H                     | Faulty |
| 9  | 0.016   | 0.889   | NOM = M & NSM = H & TPC = H           | Faulty |
| 10 | 0.014   | 0.824   | FOUT = H & NSM = H & TPC = M          | Faulty |
| 11 | 0.014   | 0.824   | FOUT = H & NOF = L & NOM = H          | Faulty |
| 12 | 0.018   | 0.900   | NBD = M & NOM = L & NSF = H & NSM = L | Faulty |
| 13 | 0.011   | 0.917   | VG = H & NOF = H & NOM = M & NOT = H  | Faulty |

頻繁に変更されているモジュールのうち、データ複雑さ、制御フロー複雑さ、規模のいずれかが大きいとバグが残存している可能性が高い。

# まとめ

---

- リポジトリマイニングとは
- リポジトリマイニングは何の役に立つか？
- 要素技術の紹介
  - 機械学習によるバグモジュール予測
  - アソシエーションルールマイニング