

ソフトウェアテスト

[13] バグ予測とテスト計画

Software Testing
[13] Bug Prediction and Test Plan

あまん ひろひさ
阿萬 裕久(AMAN Hirohisa)
aman@ehime-u.ac.jp

Fault-Prone モジュール分析

- バグのことをソフトウェア工学では**フォールト** (fault)と呼ぶことも多い
 - プログラムの**誤り**(error)により, プログラム中に**フォールト**が作り込まれる. 結果, プログラムの実行中に**障害, 故障, 不具合** (failure)が発生する.

- メトリクスにより**フォールトがありそうな**(Fault-Prone)モジュールの特徴を見つける

(C) 2016-2022 Hirohisa AMAN

2

Fault-Prone モジュール分析 (Fault-Prone モジュールぶんせき): Fault-Prone module analysis

バグ: bug

ソフトウェア工学(こうがく): software engineering

障害(しょうがい)

故障(こしょう)

不具合(ふぐあい)

特徴(とくちょう): characteristics, features

Fault-Prone モジュール分析の意義

□ ソフトウェア品質保証に必要な活動:

- テスト
- レビュー

□ これらの活動の計画に役立てる:

フォールト(fault)が**どの部分にありそうか**特定する

※直感的には「事故が起こりやすい交差点はどこか?」という考え方

意義(いぎ): purpose

ソフトウェア品質保証(ソフトウェアひんしつほしょう): software quality assurance

テスト: testing

レビュー: reviewing

活動(かつどう): activity

計画(けいかく): planning

直感的(ちょっかんてき): intuitive

事故が起こりやすい(じこがおこりやすい): an accident is likely to occur

交差点(こうさてん): intersection

フォールト(バグ)の分布に着目

- フォールト(バグ)は人間によって誤って作られている
 - 1つ1つのバグだけに注目してもそれぞれ原因が違うので、それだけでは解析が難しい
 - 全体の傾向をとらえる目的で統計学的アプローチが重要になる

- フォールト(バグ)が, これまでにどういった分布になっていたのかを試してみる

(C) 2016-2022 Hirohisa AMAN

4

誤って(あやまって): at fault

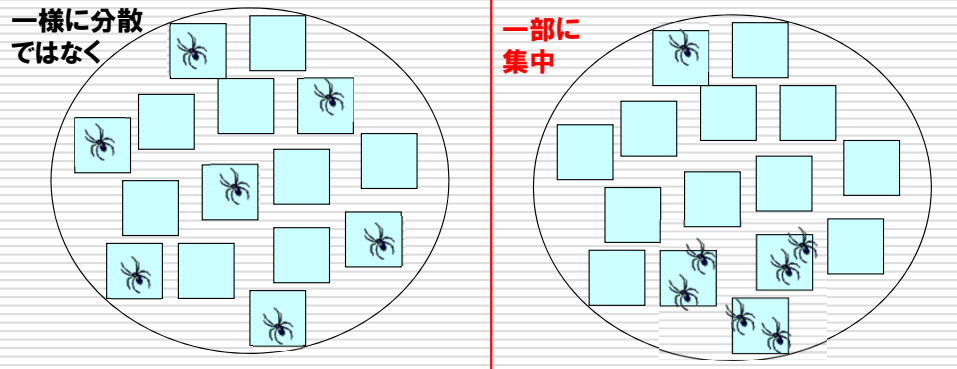
原因が違う(げんいんがちがう): different causes

統計学的アプローチ(とうけいがくてきアプローチ): statistical approach

分布(ぶんぷ): distribution

パレートの原理

□ バグの約80%は、約20%のモジュールに存在



(C) 2016-2022 Hirohisa AMAN

5

パレートの原理 (パレートのげんり): Pareto principle

一様に分散 (いちようにぶんさん): spread uniformly

一部に集中 (いちぶにしゅうちゅう): concentrated in fewer parts

Rstudio を起動しなさい

- これから Rstudio を使ったデータ分析を行います
- Teams から lecture-materials.zipをダウンロードしなさい(この中に `Rscript13.R` と `data13.csv` があります)
- `Rscript13.R` を Rstudio で開きなさい

Now we perform a data analysis exercise using Rstudio.

Download “lecture-materials.zip” from Microsoft Teams.

The zip file contains “Rscript13.R” and “data13.csv”

Open “Rscript13.R” by Rstudio

前回と同じメトリクスを使う

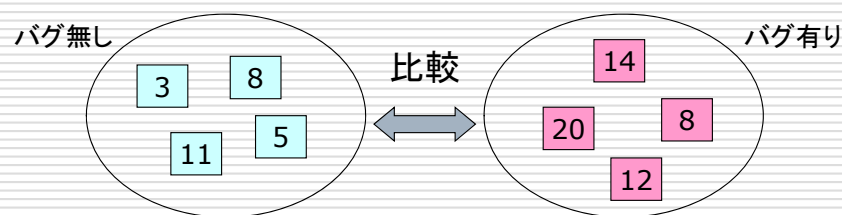
- まずは前回と同じ NASA 公開のデータを使う
 - データファイル名 `data13.csv`
 - この内容を `data` という名前のデータフレームとして読み込みます

```
data = read.csv ( file.choose ( ) )
```

Load the metric data file (data13.csv) as a data frame “data” in R

まずは簡単な分析を

- モジュール集合を「バグ無し」と「バグ有り」の2種類に分割し,
 - サイクロマティック数の違いを見る
 - バグ予測のために(サイクロマティック数の)閾値を見つける



(C) 2016-2022 Hirohisa AMAN

8

簡単(かんたん)な分析(ぶんせき): a simple analysis

モジュール集合(モジュールしゅうごう): set of modules

バグ無し(バグなし): non buggy

バグ有り(バグあり): buggy

分割(ぶんかつ): partition

サイクロマティック数: cyclomatic number

違い(ちがい): difference

バグ予測(バグよそく): bug prediction

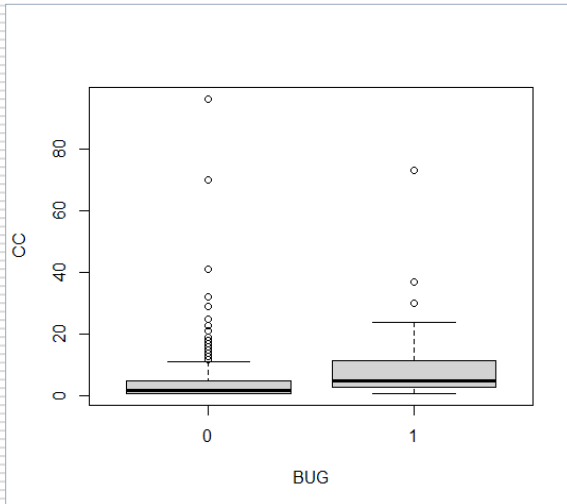
閾値(しきいち): threshold

比較(ひかく): compare

箱ひげ図

列名 列名 データフレーム

```
boxplot(CC~BUG, data=data)
```



大まかであるが
分布の違いが
見てとれる

縦軸: サイクロマティック数

横軸: バグの有無
(0 = なし, 1 = あり)

(C) 2016-2022 Hirohisa AMAN

9

箱ひげ図 (はこひげず): **boxplot**

大まか (おおまか): **rough**

分布 (ぶんぷ) の違い (ちがい): **difference in distribution**

縦軸 (たてじく): **vertical axis, Y axis**

横軸 (よこじく): **horizontal axis, X axis**

有無 (うむ): **presence or absence**

要約統計量でも比較

- バグ無しとバグ有りでサイクロマティック数を分けて summary 関数でチェック

```
cc0 = data$CC[data$BUG==0]
```

```
cc1 = data$CC[data$BUG==1]
```

```
summary(cc0)
```

```
summary(cc1)
```



Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.000	1.000	2.000	4.705	5.000	96.000
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.000	3.000	5.000	9.729	11.250	73.000

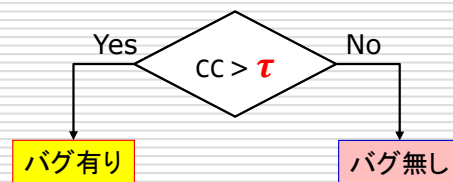
(C) 2016-2022 Hirohisa AMAN

10

要約統計量(ようやくとうけいりょう): summary statistics

メトリクスの閾値を考える

- データを見ると, サイクロマティック数はバグの有無と関係がありそうだといえる
 - サイクロマティック数が高い方が疑わしい
- そこで, ある値より大きいようであればバグ有りと予測することにする



(C) 2016-2022 Hirohisa AMAN

11

閾値(しきいち): threshold

有無(うむ): presence or absence

サイクロマティック数が高いほうが疑わしい: the higher the cyclomatic number, the more suspicious

データセットを分割

- 簡単な予測モデルを考える
 - この**予測モデルを構築する**には**訓練データ**が必要
 - さらに, **予測モデルの能力を評価する**には,
テストデータも必要

- そこで data を2つに分割してこれらを用意することにする

予測モデル(よそくモデル): prediction model

構築する(こうちくする): build

訓練する(くんれんする): train

訓練データ(くんれんデータ): training data

能力を評価(のうりよくをひょうか): evaluate the performance

テストデータ: test data

分割する(ぶんかつする): divide

ランダムに振り分ける

- まずは行番号 1 ~ `nrow(data)` をシャッフルする: 次の関数 `sample` でシャッフルされた行番号のリストを得る

```
set.seed(1234)
idx = sample( nrow(data) )
```

行番号(ぎょうばんごう): line (row) number

シャッフル後のデータを分割

- 訓練データ(**d.train**): 最初の 300 個

```
d.train = data[idx[1:300], ]
```

- テストデータ(**d.test**): 残り(205個)

```
d.test = data[idx[301:nrow(data)], ]
```

訓練データ: training data

テストデータ: test data

(例)サイクロマティック数の閾値を10にした場合(1／6)

- 簡単な例として、**サイクロマティック数(CC)**が**10を超える**ようであればバグ有りと予測する
としよう
- 訓練データでこれに該当するモジュールでの
バグの有無を確認してみる

```
d.train$BUG[d.train$CC>10]
```



```
[1] 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0
```

(C) 2016-2022 Hirohisa AMAN

15

Suppose we predict that the module is buggy if its cyclomatic number > 10

(例)サイクロマティック数の閾値を10にした場合(2/6)

□ 「**バグ有りと予測**」した結果を `result` として

```
result = d.train$BUG[d.train$CC>10]
```

```
[1] 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0
```

```
length(result)
```



予測した個数

```
[1] 41
```

```
sum(result)
```



予測の中で正解した個数

```
[1] 10
```

(C) 2016-2022 Hirohisa AMAN

16

予測(よそく): prediction

正解(せいかい): correct

(例)サイクロマティック数の閾値を10にした場合(3／6)

□ 結果を整理すると

length(result)



予測した個数

[1] 41

sum(result)



予測の中で正解した個数

[1] 10

訓練データを使った 予測の結果		実際		合計
		Buggy	Non Buggy	
予測	Buggy	10	31	41
	Non Buggy			
合計				

(C) 2016-2022 Hirohisa AMAN

17

(例)サイクロマティック数の閾値を10にした場合(4/6)

□ 実際にバグ有りだった個数は

```
sum(d.train$BUG)
```



実際にバグ有りの個数

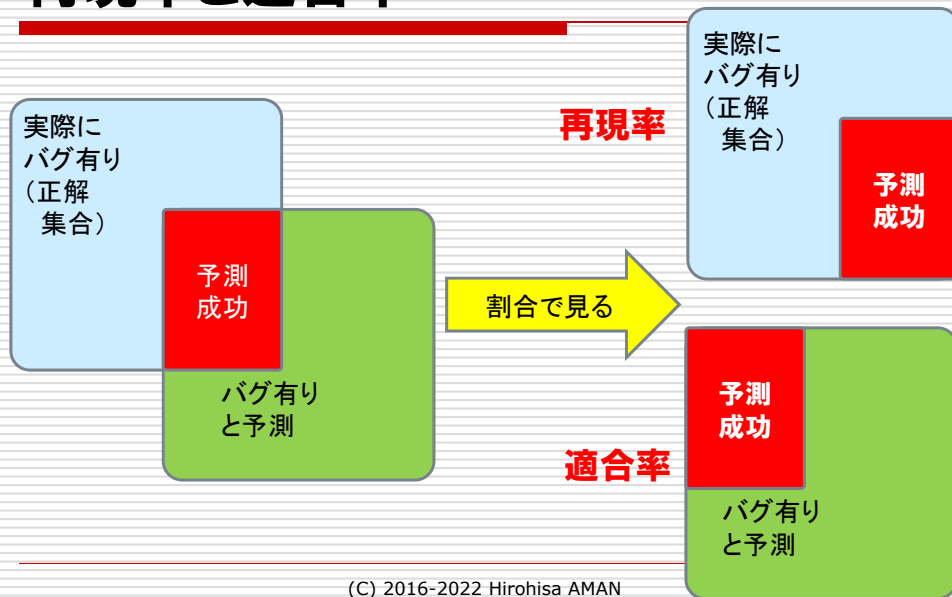
[1] 30

訓練データを使った 予測の結果		実際		合計
		Buggy	Non Buggy	
予測	Buggy	10	31	41
	Non Buggy	20		
合計		30		

(C) 2016-2022 Hirohisa AMAN

18

再現率と適合率



(C) 2016-2022 Hirohisa AMAN

再現率(さいげんりつ): recall

適合率(てきごうりつ): precision

成功(せいこう): success

(例)サイクロマティック数の閾値を10にした場合(5/6)

訓練データを使った 予測の結果		実際		合計
		Buggy	Non Buggy	
予測	Buggy	10		41
	Non Buggy			
合計		30		

$$\text{再現率} = \frac{10}{30} \cong 0.333 \quad \text{適合率} = \frac{10}{41} \cong 0.244$$

F 値

□ 再現率と適合率の調和平均

$$\frac{1}{F\text{値}} = \frac{\frac{1}{\text{再現率}} + \frac{1}{\text{適合率}}}{2}$$



$$F\text{値} = \frac{1}{\frac{1}{2} \left(\frac{1}{\text{再現率}} + \frac{1}{\text{適合率}} \right)} = \frac{2 \cdot \text{再現率} \cdot \text{適合率}}{\text{再現率} + \text{適合率}}$$

(C) 2016-2022 Hirohisa AMAN

21

F 値(Fち): F value, F score

調和平均(ちょうわへいきん): harmonic mean

(参考)調和平均の例

小学生の算数の問題

□ 家と駅の間を車で往復しました。その際の移動速度は、行きが30km/h、帰りは50km/hでした。平均の移動速度は？

$$\frac{\text{移動距離}}{\text{行きの時間}} = 30 \quad \rightarrow \quad \text{行きの時間} = \frac{\text{移動距離}}{30}$$

$$\frac{\text{移動距離}}{\text{帰りの時間}} = 50 \quad \rightarrow \quad \text{帰りの時間} = \frac{\text{移動距離}}{50}$$

$$\text{平均速度} = \frac{2 \cdot \text{移動距離}}{\text{行きの時間} + \text{帰りの時間}} = \frac{2 \cdot \text{移動距離}}{\frac{\text{移動距離}}{30} + \frac{\text{移動距離}}{50}} = \frac{1}{\frac{1}{2} \left(\frac{1}{30} + \frac{1}{50} \right)} = 37.5$$

(C) 2016-2022 Hirohisa AMAN

22

小学生(しょうがくせい): elementary school student

算数(さんすう): math

家(いえ): home

駅(えき): station

車(くるま): car

往復(おうふく): round trip

速度(そくど): speed

移動距離(いどうきょり): distance

(例)サイクロマティック数の閾値を10にした場合(6／6)

$$\text{再現率} = \frac{10}{30} \cong 0.333 \quad \text{適合率} = \frac{10}{41} \cong 0.244$$

□ この場合の F 値は

$$\text{F値} = \frac{2 \cdot \text{再現率} \cdot \text{適合率}}{\text{再現率} + \text{適合率}} = \frac{2 \cdot \frac{10}{30} \cdot \frac{10}{41}}{\frac{10}{30} + \frac{10}{41}} \cong 0.282$$

【演習1】(宿題:homework)

- メトリクスとしてサイクロマティック数(CC)ではなく **LOC** を使うことにする
- 訓練データに対して, **LOC** の閾値を 30 とした場合の再現率, 適合率, F値を答えなさい

p15 -- p23
を参考にしなさい

Let us use the metric LOC for our bug prediction, rather than the cyclomatic number (CC).

Compute the recall, the precision and the F value when we use 30 as our threshold.

(see pages 13—23)

F 値の計算を行う関数(1 / 5)

□ F 値を計算する関数を作っていく

□ まずは**再現率**(recall)の計算について考える

$$\text{再現率} = \frac{\text{バグ有りと予測して正解した個数}}{\text{実際にバグ有りだった個数}}$$

```
result = d.train$BUG[d.train$CC>10]  
recall = sum(result)/sum(d.train$BUG)
```

(C) 2016-2022 Hirohisa AMAN

25

Let us consider a function for computing the F value.

バグ有りと予測(バグありとよそく): predicted as buggy modules

正解した(せいかいした): the prediction was correct

実際にバグ有りだった(じっさいにバグありだった): they were buggy

F 値の計算を行う関数(2/5)

□ 次に, **適合率**(precision)は

$$\text{適合率} = \frac{\text{バグ有りと予測して正解した個数}}{\text{バグ有りと予測した個数}}$$

```
result = d.train$BUG[d.train$CC>10]
recall = sum(result)/sum(d.train$BUG)
precision = sum(result)/sum(d.train$CC>10)
```

バグ有りと予測(バグありとよそく): predicted as buggy modules
正解した(せいかいした): the prediction was correct

F 値の計算を行う関数(3/5)

- 表記を簡単にするため、次のように置き換えることにする

```
d.train$BUG  
d.train$CC>10
```



```
bug  
target
```



```
result = bug[target]  
recall = sum(result)/sum(bug)  
precision = sum(result)/sum(target)
```

表記(ひょうき): notation

簡単(かんたん)にする: simplify

置き換える(おきかえる): replace

F 値の計算を行う関数(4/5)

$$F\text{値} = \frac{2 \cdot \text{再現率} \cdot \text{適合率}}{\text{再現率} + \text{適合率}}$$

□ 次のようにして F 値が求まる

```
result = bug[target]
recall = sum(result)/sum(bug)
precision = sum(result)/sum(target)
2*recall*precision/(recall + precision)
```

F 値の計算を行う関数(5／5)

- 先ほど置き換えた記号を引数として, Rで利用できる関数を定義すると次のようになる

```
f.value = function(bug, target){  
  result = bug[target]  
  recall = sum(result)/sum(bug)  
  precision = sum(result)/sum(target)  
  2*recall*precision/(recall + precision)  
}
```

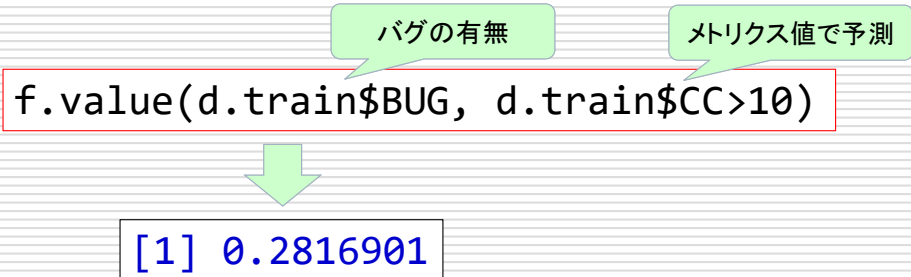
置き換えた記号(おきかえたきごう): replaced symbol

引数(ひきすう): argument

関数(かんすう)を定義(定義)する: define a function

作成した関数を使う

- 次のようにして呼び出せば F 値の計算を行ってくれる


`f.value(d.train$BUG, d.train$CC>10)`
↓
`[1] 0.2816901`

Utilize the function we made.

呼び出す(よびだす): call

バグの有無(バグの有無): presence or absence of bugs

メトリクス値で予測(メトリクスちでよそく): predict it using the metric value

テストデータで検証

□ 訓練データ(d.train)ではなく
テストデータ(d.test)について予測してみる

□ F 値で評価する

```
f.value(d.test$BUG, d.test$CC>10)
```



```
[1] 0.2051282
```

サイクロマティック数
だけの判定なので
性能は低い...

Predict bugs in test data set (d.test)

サイクロマティック数だけの判定(サイクロマティックすうだけのはんてい): judgement
using only the cyclomatic number

性能は低い(せいのうはひくい): low performance



10-minutes rest break

10分休憩

テストの優先順位付け

- バグがあるかもしれないモジュールを優先的にテストすることを考える
- 完璧ではないが、メトリクス値を使うと優先順位付けができる

メトリクス値の大きい順(降順)にテストするという方法

優先順位付け(ゆうせんじゅんいづけ): prioritization

バグがあるかもしれない(バグがあるかもしれない): it might have a bug (fault)

メトリクス値の大きい順(メトリクスちのおおきいじゅん), 降順(こうじゅん): in descending order of the metric value

サイクロマティック数の大きい順にテスト

- テストデータ(d.test)のモジュールをサイクロマティック数(d.test\$CC)の**大きい順(降順)に並べる**

```
order(d.test$CC, decreasing=T)
```

- その順番で**バグの有無を記録**する

```
found.bugs =  
d.test$BUG[order(d.test$CC, decreasing=T)]
```

大きい順に並べる(おおきいじゅんにならべる): sort them in descending (decreasing) order

バグの有無(バグの有無): presence or absence of bug

記録する(きろくする): record, store

バグの検出数を累積する

- 変数 `found.bugs` には順番にバグの有無(1 or 0)が記録されている

```
0 0 0 0 1 0 1 0 0 0 0 0 ..... 
```

- これを累積する(足していく)ことで「**それまでに見つかったバグの数**」を調べる

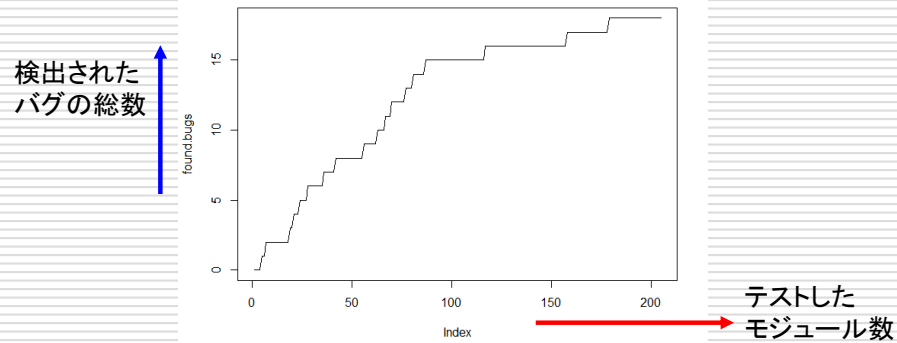
```
for ( i in 2:length(found.bugs) ){  
    found.bugs[i] = found.bugs[i-1] + found.bugs[i]  
}
```

バグの検出数(バグのけんしゅつすう): number of detected (found) bugs
累積する(るいせきする): accumulate

累積バグ検出数をグラフで表す

□ 累積した found.bugs をグラフで表す

```
plot(found.bugs, type='l')
```



(C) 2016-2022 Hirohisa AMAN

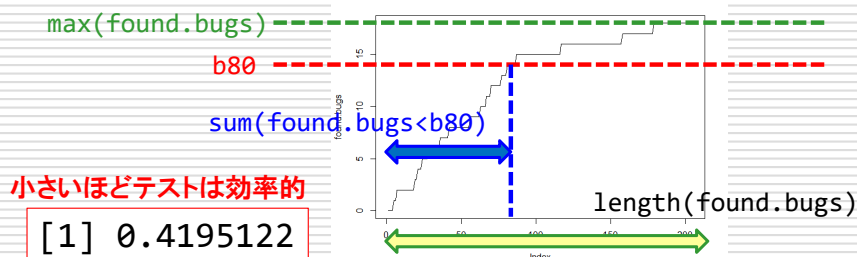
36

累積バグ検出数(るいせきバグけんしゅつすう): cumulative number of detected bugs

80% のバグをどれだけ早く見つけたか

□ 全体のバグのうち, 80% のバグをどれだけ少ないテストで見つけることができたか？

```
b80 = max(found.bugs)*0.8  
sum(found.bugs<b80)/length(found.bugs)
```



(C) 2016-2022 Hirohisa AMAN

37

バグを早く見つける(バグをはやくみつける): detect bugs earlier

テストの効率(テストのこうりつ): testing efficiency

小さいほどテストは効率的(こうりつてき): the lower the value, the higher the efficiency

【演習2】(宿題:homework)

- メトリクスとして **LOC** を使い, その大きい順にテストを行ったとする
テストデータについて, 80%のバグを見つけるのに必要なテストの割合を答えなさい
(サイクロマティック数の場合は 0.4195122だった)
- メトリクスとして **CALL** を使った場合についても答えなさい

p34 -- p37
を参考にしなさい

Let us use the metric LOC for our bug prediction, rather than the cyclomatic number (CC).

Compute the ratio of run tests to detect 80% of bugs.

Also compute such a ratio using the metric CALL.

(see pages 34—37)

複数のメトリクスを使った予測

- 単一のメトリクスだけでバグ予測を行うよりも**複数のメトリクス**を使った方がより精度の高い予測ができるのでは？
- **各メトリクスを説明変数**(予測の材料)とした次式の**線形重回帰モデル**は使えないか？

$$y = a_1x_1 + a_2x_2 + a_3x_3 + b$$

CC

CALL

LOC

(ただし, a_1, a_2, a_3, b はいずれも定数)

(C) 2016-2022 Hirohisa AMAN

39

単一の(たんいつの): single

複数の(ふくすうの): two or more

精度の高い(せいどのたかい): high accuracy

説明変数(せつめいへんすう): explanatory variables

材料(ざいりょう): material, input

線形重回帰モデル(せんけいじゅうかいきモデル): linear multiple regression model

1つ困ったことが...

- このままだと目的変数 y の値域が $(-\infty, +\infty)$

$$y = a_1x_1 + a_2x_2 + a_3x_3 + b$$

- 予測したいのはバグの有無の可能性で, 仮にこれを p とおくと(確率と同様に) p の値域は $[0, 1]$ でなければ困る

y と p の間で何か適切な変換を行う必要がある

値域(ちいき): range

可能性(かのうせい): possibility

確率(かくりつ): probability

適切な変換(てきせつなへんかん): proper transformation

ロジット変換

□ p を $\log \frac{p}{1-p}$ に置き換えることを**ロジット変換**という(ただし, $0 < p < 1$)

■ $p = 0.5$: $\log \frac{p}{1-p} = \log \frac{0.5}{1-0.5} = \log 1 = 0$

■ $p \cong 1$: $\log \frac{p}{1-p} \cong \log \frac{1}{0} = \log \infty = +\infty$

■ $p \cong 0$: $\log \frac{p}{1-p} \cong \log \frac{0}{1} = \log 0 = -\infty$

可能性 p が50%の時に 0 となり, それよりも上ならばプラスに, 下ならばマイナスになる. しかも値域は $(-\infty, +\infty)$ になっている.

ロジット変換(ロジットへんかん): logit transformation

目的変数 y を p のロジット変換に

$$y = a_1x_1 + a_2x_2 + a_3x_3 + b$$



$$\log \frac{p}{1-p} = a_1x_1 + a_2x_2 + a_3x_3 + b$$

□ これを**ロジスティック回帰モデル**という

ロジスティック回帰モデル(ロジスティックかいきモデル): logistic regression model

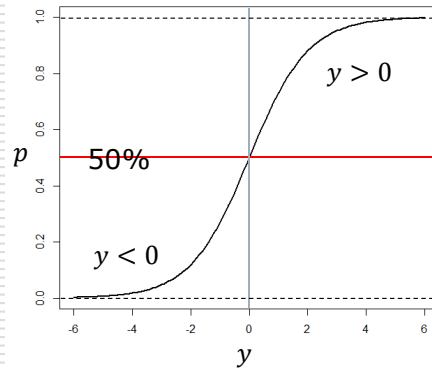
ロジスティック回帰モデルの別表現

□ 先ほどの数式で左辺が p となるよう変形

$$p = \frac{1}{1 + e^{-y}}$$

$$y = a_1x_1 + a_2x_2 + a_3x_3 + b$$

ニューラルネットワークの活性化関数
でよく使われるシグモイド関数と同じ



(C) 2016-2022 Hirohisa AMAN

43

別表現(べつひょうげん): another expression

ニューラルネットワーク: neural network

活性化関数(かつせいいかかんすう): activation function

シグモイド関数: sigmoid function

R でロジスティック回帰モデルを作る

□ 関数 glm を使う

```
model = glm(BUG ~ CC+CALL+LOC, d.train,  
            family="binomial")  
summary(model)
```



```
Call:
glm(formula = BUG ~ CC + CALL + LOC, family = "binomial", data = d.train)

Deviance Residuals:
    Min       1Q   Median       3Q      Max 
-1.6289  -0.4119  -0.3388  -0.3103   2.4587 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -3.08035     0.20626  -19.149   <2e-16 ***
CC           -0.17097     0.06649   -2.571   0.01013 *
CALL         0.14645     0.00581    2.225   0.02607 *
LOC          0.03375     0.01202    2.808   0.00499 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 195.05 on 299 degrees of freedom
Residual deviance: 168.69 on 296 degrees of freedom
AIC: 176.69

Number of Fisher Scoring iterations: 5
```

$$\log \frac{p}{1-p} = -0.17x_1 + 0.15x_2 + 0.03x_3 - 3.01$$

予測結果の取得

□ テストデータに対する予測結果を取得する

```
predict(model, type="response", d.test)
```



	99	505	124	97
	0.053450137	0.067460143	0.148106912	0.050136245

この中から出力された確率のみを取り出す

```
result = predict(model, type="response", d.test)  
result = as.vector(result)
```

予測結果(よそくけっか): prediction results

取得(しゅとく)する: get, obtain

確率(かくりつ): probability

確率の高い順にテスト

- テストデータ(d.test)のモジュールをロジスティック回帰モデルの出力した確率(result)の**高い順(降順)**に**並べる**
- その順番で**バグの有無**を記録する

```
found.bugs =  
d.test$BUG[order(result, decreasing=T)]
```

高い順に並べる(たかいじゅんにならべる): sort them in descending (decreasing) order

バグの有無(バグのうむ): presence or absence of bug

記録する(きろくする): record, store

バグの検出数を累積する

- 変数 `found.bugs` には順番にバグの有無(1 or 0)が記録されている

```
0 0 0 0 1 0 1 0 0 0 0 0 ..... 
```

- これを累積する(足していく)ことで「**それまでに見つかったバグの数**」を調べる

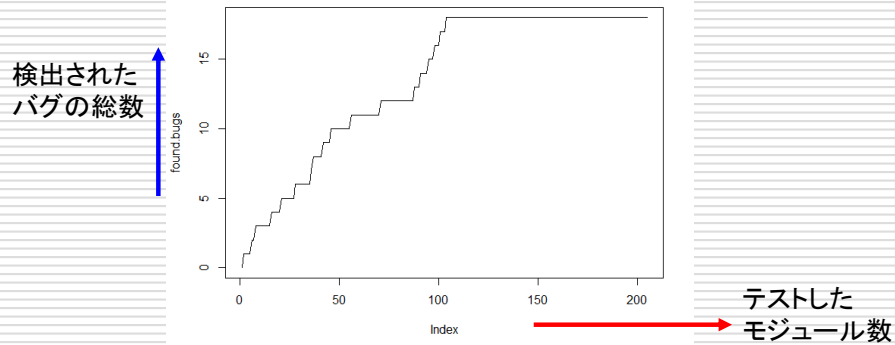
```
for ( i in 2:length(found.bugs) ){  
    found.bugs[i] = found.bugs[i-1] + found.bugs[i]  
}
```

バグの検出数(バグのけんしゅつすう): number of detected (found) bugs
累積する(るいせきする): accumulate

累積バグ検出数をグラフで表す

□ 累積した found.bugs をグラフで表す

```
plot(found.bugs, type='l')
```



(C) 2016-2022 Hirohisa AMAN

48

累積バグ検出数(るいせきバグけんしゅつすう): cumulative number of detected bugs

80% のバグをどれだけ早く見つけたか

- 全体のバグのうち, 80% のバグをどれだけ少ないテストで見つけることができたか?

```
b80 = max(found.bugs)*0.8  
sum(found.bugs<b80)/length(found.bugs)
```

小さいほどテストは効率的

[1] 0.4585366

サイクロマティック数だけの場合

[1] 0.4195122

(C) 2016-2022 Hirohisa AMAN

49

バグを早く見つける(バグをはやくみつける): detect bugs earlier

テストの効率(テストのこうりつ): testing efficiency

小さいほどテストは効率的(こうりつてき): the lower the value, the higher the efficiency

なぜ性能が良くならなかったのか？

□ バグ有りとバグ無しのサンプル数の**不均一性**が影響している

■ バグ無しのサンプルが圧倒的に多い

```
sum(d.train$BUG==1)  
sum(d.train$BUG==0)
```



```
[1] 30  
[1] 270
```

■ **全体の誤差が最小となるように**パラメータ調整が行われるため、結果的に**バグ無しと予測しやすいモデル**になってしまう

(C) 2016-2022 Hirohisa AMAN

50

Why did not the performance get better?

不均一性(ふきんいつせい): imbalance

圧倒的に多い(あつとうてきにおおい): great majority

全体の誤差(ぜんたいのござ): whole error

パラメータ調整(パラメータちょうせい): parameter adjustment

結果的に(けっかてきに): eventually

バグ無しと予測しやすいモデル(バグなしとよそくしやすいモデル): model that tends to predict modules as non buggy

どうやって対処するか？

□ 基本的な方策は大きく分けて2種類

①**学習での重みを変える**: 少ない方(バグ有り)のサンプルに対する誤差をより重くする;あるいは, 多い方のサンプルに対する誤差を軽くする

②**均一性のある訓練データを与える**: 訓練データにおけるバグ有りとバグ無しのサンプル数を概ね同じ(50:50)にする



ここでは②のアプローチをとることにする

対処する(たいしょする): deal

方策(ほうさく): measure

学習(がくしゅう): learning, training

重み(おもみ): weight

均一性のある(きんいつせいのある): balanced

アンダーサンプリングと オーバーサンプリング

□ アンダーサンプリング

多数派(バグ無し)をすべて使うのではなく、無作為抽出して得られる**一部のサンプルのみ**を使う

単純にアンダーサンプリングを行うと、
訓練データが極端に少なくなってしまう

□ オーバーサンプリング

少数派(バグ有り)を**水増し**して使う

水増しの方法に注意：
単純に同じサンプルを繰り返し選ぶだけで学習できるか疑問

(C) 2016-2022 Hirohisa AMAN

52

アンダーサンプリング: undersampling

オーバーサンプリング: oversampling

多数派(たすうは): majority

少数派(しょうすは): minority

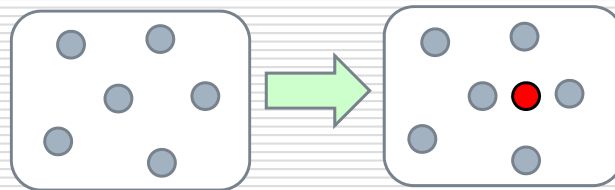
無作為抽出(むさくいちゅうしゅつ): random sampling

水増し(みずまし): pad

SMOTE アルゴリズム

Synthetic Minority Over-sampling Technique

- **少数派のサンプルを人工的に作成**する(近傍データを使ってランダムに作る)ことでオーバーサンプリングを行う



- 同時に**多数派に対してはアンダーサンプリング**も行う

(C) 2016-2022 Hirohisa AMAN

53

近傍データ(きんぼうデータ): neighborhood data

SMOTE アルゴリズムで訓練データを用意する(1 / 3)

- まずは専用のパッケージをインストール

```
install.packages("smotefamily")
```

- そして, それを読み込む

```
library(smotefamily)
```

SMOTE アルゴリズムで訓練データを用意する(2/3)

- SMOTE アルゴリズムでデータセットを作る
(ここでは乱数の種を設定してある)

```
set.seed(1234)
d.train.smote
  = SMOTE(d.train[,c(2,3,4)],
          d.train$BUG, K = 10)$data
```

メトリクス値の書いてある列のみを指定

目的変数(バグかどうか)

合成で使う近傍の個数

(C) 2016-2022 Hirohisa AMAN

55

d.train[, c(2,3,4)] ... the part of the training dataset corresponding to the explanatory variables (three metrics columns)

d.train\$BUG ... the prediction targets

K = 10 ... the number of neighborhood data that we use to generate artificial data

Set.seed(1234) ... setting the random seed

Why we set the random seed?

→ Because you can obtain the same results as the lecture note

SMOTE アルゴリズムで訓練データを用意する(3／3)

- SMOTE アルゴリズムで作られたデータのフォーマットがロジスティック回帰モデルで使うものと少しだけ違うので調整する

```
names(d.train.smote)[4] = "BUG"  
d.train.smote$BUG = as.numeric(d.train.smote$BUG)
```

To align the format of dataframe with the one expected in the logistic regression model, we do the above adjustment

SMOTE アルゴリズムによる 訓練データでモデルを再構築

- ロジスティック回帰モデルを作り直し,
テストデータに対する予測を行う

```
model = glm(BUG ~ CC+CALL+LOC,  
            d.train.smote, family="binomial")
```

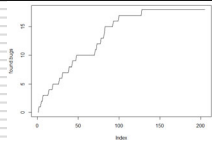
```
result = predict(model, type="response", d.test)  
result = as.vector(result)
```

テストの効率を評価

□ 改めてテストの効率を評価

```
found.bugs = d.test$BUG[order(result, decreasing = T)]  
for ( i in 2:length(found.bugs) ){  
  found.bugs[i] = found.bugs[i-1] + found.bugs[i]  
}  
plot(found.bugs, type='l')
```

```
b80 = max(found.bugs)*0.8  
sum(found.bugs<b80)/length(found.bugs)
```



[1] 0.4

少し改善
された結果
になった

改善された結果(かいぜんされたけっか): improved outcome

他にもいろいろな予測モデルがある

- ロジスティック回帰モデルは基本的なモデルであるが、最良なモデルとは限らない
- **他にも機械学習モデルは多くあるので、**
他のモデルを使うことで**テストをより効率化**
できる可能性はある

基本的なモデル(きほんてきなモデル): fundamental model

機械学習モデル(きかいがくしゅうモデル): machine learning model

【演習3】(宿題:homework)

- 乱数の種を 9876 に再設定し,

```
set.seed(9876)
```

訓練データとテストデータへの分割をやり直
なさい

- そして, 訓練データに対して, サイクロマティッ
ク数の閾値を 10 とした場合の再現率, 適合
率, 並びに F値を答えなさい

p13 -- p23
を参考にしなさい

Set the random number seed as “set.seed(9876)”

Then, redo dividing dataset into the training data and the test data,
and re-compute the recall, the precision and the F value when we use 10 as our
threshold.

(see pages 13—23)

【演習4】(宿題:homework)

- 演習3の続きとして, SMOTEアルゴリズムを使ってロジスティック回帰モデルを作りなさい (p55 のRスクリプトでも

```
set.seed(9876)
```

と指定すること)

- そして, テストデータに対するテストの効率を答えなさい

p54 -- p58
を参考にしなさい

After you did the exercise-3, build the SMOTE algorithm-based logistic regression model.

Notice that you have to set the random seed as 9876 in the R script on page 55.

Then, compute the ratio of run tests achieving 80% bug detection.

(see pages 54—58)