Defect Prediction in Software Engineering

Daniel Rodriguez and Javier Dolado

2017-07-01

Contents

Welcome **7**

I Introduction to Software Engineering Defect Prediction **9**

1 Introduction **11**

II Data Sources and Metrics and Standards in Software Engineering Defect Prediction **13**

2 Data Sources in Software Engineering **15**

3 Repositories **17**

4 Open Tools/Dashboards to extract data **19**
4.1 Issues . . . 19

5 Metrics in Software Enginering Prediction **21**
5.1 Complexity Metrics . . . 21
5.2 Object-Oriented Metrics . . . 23
5.3 Churn and Process Metrics . . . 23
5.4 Defect Standards . . . 24

III Machine Learning in Defect Prediction **27**

6 Supervised Classification **29**
6.1 Classification Trees . . . 29
6.2 Rules . . . 33
6.3 Distanced-based Methods . . . 35
6.4 Neural Networks . . . 37
6.5 Support Vector Machine . . . 37
6.6 Probabilistic Methods . . . 37
6.7 Linear Discriminant Analysis (LDA) . . . 40
6.8 Binary Logistic Regression (BLR) . . . 45
6.9 The caret package . . . 48
6.10 Ensembles . . . 48

7 Regression tecniques in Software defect prediction **51**
7.1 Zero Poison . . . 51
7.2 Poisson Regression . . . 57
7.3 Compare to Null Model (intercept) . . . 58

7.4 Negative Binomial . . . 58
7.5 Compare Models Fit. AIC and BIC . . . 64
7.6 Compare prediction of Defects . . . 65
7.7 Plot predictions . . . 66
7.8 Regression diagnostics . . . 68
7.9 References and R code used . . . 76

IV Evaluation 79

8 Evaluation of Models 81
8.1 Building and Validating a Model . . . 81
8.2 Evaluation of Classification Models . . . 82
8.3 Other Metrics used in Software Engineering with Classification . . . 84
8.4 Graphical Evaluation . . . 84
8.5 Numeric Prediction Evaluation . . . 85
8.6 Underfitting vs. Overfitting . . . 86

V Problems and Issues in Defect Prediction 89

9 Data Mining Issues in Defect Prediction 91

10 Outliers, missing values, inconsistencies data noise 93

11 Redundant and irrelevant attributes and instances 95

12 Overlapping or class separability 97

13 Data shifting 99

14 Imbalance datasets 101

15 Evaluation metrics and the evaluation of models 103

16 Cross project defect prediction 105

VI Examples 107

17 Feature Subsect Selection in Defect Prediction 109

18 Imbalanced data 111

19 Subgroup Discovery 113

20 Semi-supervised learning 117

21 Learning from Crowds 119

22 Multi-objective Rules for defect prediction 121

23 Settings Thresholds for Defect Prediction 123
23.1 Use of Mean and Standard Deviation . . . 123
23.2 Use of Weighted Benchmark Data . . . 123
23.3 Use of Quantiles . . . 123
23.4 Some further literature . . . 124

VII Bibliography **125**

Welcome

These notes will try to describe the state of the art in software defect prediction.

Acknowledgments

- TIN2016-76956-C3-1-R
- TIN2015-71841-REDT

Work in progress.

Part I

Introduction to Software Engineering Defect Prediction

Chapter 1

Introduction

Defect prediction in software engineering addresses the problem of predicting which software modules will be error prone. In this way, testing effort can be focused on those modules that are more error prone.

Techniques are typically classified into:

- Defect Classification - using data mining tecniques to find error prone modules
- Defect Ranking - sort the modules according to their probability of being error-prone. For example, finding the 20 per cent of the modules with most bugs.

There are other works that make use of textual information to classify defects. For example in order to find what to fix first or what requirement should be implemented (functinal or non-functional requierements).

- *Categorise reviews* - for example from Bug Tracking tools (e.g. Bugzilla) defect descriptions or Mobile aplications (Appstore or Google play) reviews
- *Defect categories* - Standards to deal with defects, for example *Orthogonal Defect Classification* (ODC) or with classifying requirements so that we can answer "What to fix first or implement (non-functional requierements)". For example, (Huang et al., 2011)

There are other tools to deal with defects such as Findbugs or PMD that work with a set of rules describing possible problems but these tools do not use predictive models but here we refer mainly to *machine learning* approaches to defect prediction.

Some systematic reviews of software fault prediction studies include:

- (Hall et al., 2012) - A Systematic Literature Review on Fault Prediction Performance in Software Engineering
- (Catal and Diri, 2009) - A systematic review of software fault prediction studies
- (Catal, 2011) - Software fault prediction: A literature review and current trends

Part II

Data Sources and Metrics and Standards in Software Engineering Defect Prediction

Chapter 2

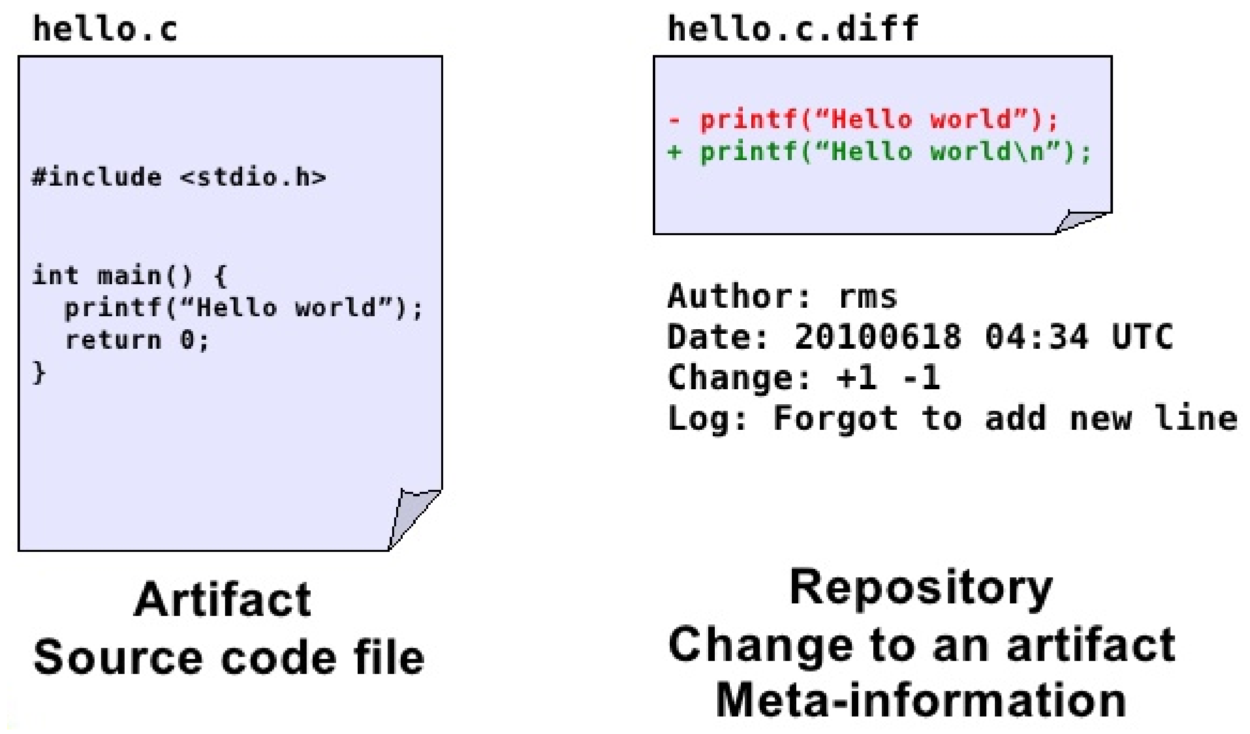
Data Sources in Software Engineering

We classify this trail in the following categories:

- *Source code* can be studied to measure its properties, such as size or complexity.
- *Source Code Management Systems* (SCM) make it possible to store all the changes that the different source code files undergo during the project. Also, SCM systems allow for work to be done in parallel by different developers over the same source code tree. Every change recorded in the system is accompanied with meta-information (author, date, reason for the change, etc) that can be used for research purposes.
- *Issue or Bug tracking systems* (ITS). Bugs, defects and user requests are managed in ISTs, where users and developers can fill tickets with a description of a defect found, or a desired new functionality. All the changes to the ticket are recorded in the system, and most of the systems also record the comments and communications among all the users and developers implied in the task.
- *Messages* between developers and users. In the case of free/open source software, the projects are open to the world, and the messages are archived in the form of mailing lists and social networks which can also be mined for research purposes. There are also some other open message systems, such as IRC or forums.
- *Meta-data about the projects.* As well as the low level information of the software processes, we can also find meta-data about the software projects which can be useful for research. This meta-data may include intended-audience, programming language, domain of application, license (in the case of open source), etc.
- *Usage data.* There are statistics about software downloads, logs from servers, software reviews, etc.

Types of information stored in the repositories:

- Meta-information about the project itself and the people that participated.
  - Low-level information
    - Mailing Lists (ML)
    - Bugs Tracking Systems (BTS) or Project Tracker System (PTS)
    - Software Configuration Management Systems (SCM)
  - Processed information. For example project management information about the effort estimation and cost of the project.
- Whether the repository is public or not
- Single project vs. multiprojects. Whether the repository contains information of a single project with multiples versions or multiples projects and/or versions.



```
hello.c

#include <stdio.h>

int main() {
  printf("Hello world");
  return 0;
}
```

```
hello.c.diff

- printf("Hello world");
+ printf("Hello world\n");
```

```
Author: rms
Date: 20100618 04:34 UTC
Change: +1 -1
Log: Forgot to add new line
```

Figure 2.1: Metadata

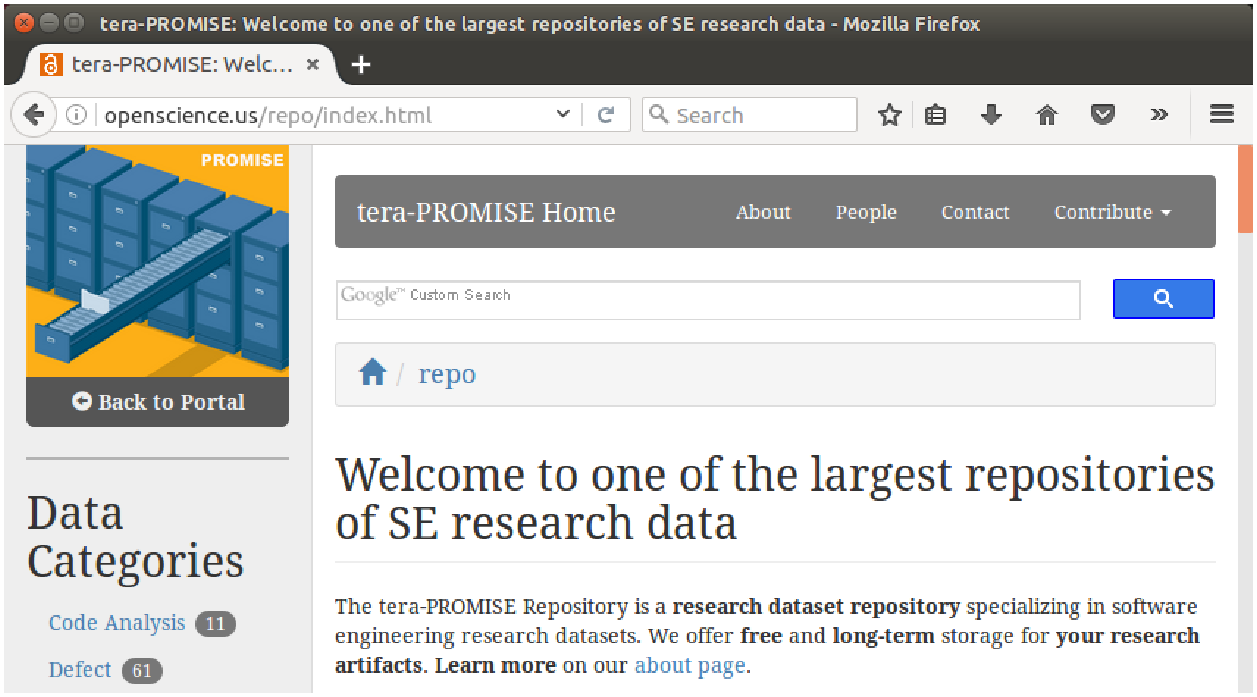
- Type of content, open source or industrial projects
- Format in which the information is stored and formats or technologies for accessing the information:
  - Text. It can be just plain text, CSV (Comma Separated Values) files, Attribute-Relation File Format (ARFF) or its variants
  - Through databases. Downloading dumps of the database.
  - Remote access such as APIs of Web services or REST

Chapter 3

Repositories

There is a number of open research repositories in Software Engineering. Among them:

- PROMISE (PRedictOr Models In Software Engineering): http://openscience.us/repo/



Welcome to one of the largest repositories of SE research data

The tera-PROMISE Repository is a **research dataset repository** specializing in software engineering research datasets. We offer **free** and **long-term** storage for **your research artifacts**. **Learn more** on our about page.

Figure 3.1: Promise Repository

- Finding Faults using Ensemble Learners (ELFF) (Shippey et al., 2016) http://www.elff.org.uk/
- FLOSSMole (Howison et al., 2006) http://flossmole.org/
- FLOSSMetrics (Herraiz et al., 2009): http://flossmetrics.org/
- Qualitas Corpus (QC) (Tempero et al., 2010): http://qualitascorpus.com/
- Sourcerer Project (Linstead et al., 2009): http://sourcerer.ics.uci.edu/
- Ultimate Debian Database (UDD) (Nussbaum and Zacchiroli, 2010) http://udd.debian.org/
- SourceForge Research Data Archive (SRDA) (Van Antwerp and Madey, 2008) http://zerlot.cse.nd.edu/

- SECOLD (Source code ECOsystem Linked Data): http://www.secold.org/
- Software-artifact Infrastructure Repository (SIR) [http://sir.unl.edu]
- OpenHub: https://www.openhub.net/

Not openly available (and mainly for effort estimation):

- The International Software Benchmarking Standards Group (ISBSG) http://www.isbsg.org/
- TukuTuku http://www.metriq.biz/tukutuku/

Some papers and publications/theses that have been used in the literature:

- Helix Data Set (Vasa, 2010): http://www.ict.swin.edu.au/research/projects/helix/
- Bug Prediction Dataset (BPD) (D'Ambros et al., 2010, D'Ambros et al. (2011)): http://bug.inf.usi.ch/
- Eclipse Bug Data (EBD) (Zimmermann et al., 2007, Nagappan et al. (2012)): http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

Chapter 4

Open Tools/Dashboards to extract data

Process to extract data:

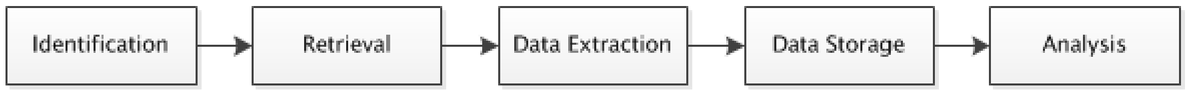


Figure 4.1: Process

Within the open source community, several toolkits allow us to extract data that can be used to explore projects:

Metrics Grimoire http://metricsgrimoire.github.io/

SonarQube http://www.sonarqube.org/

CKJMS http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

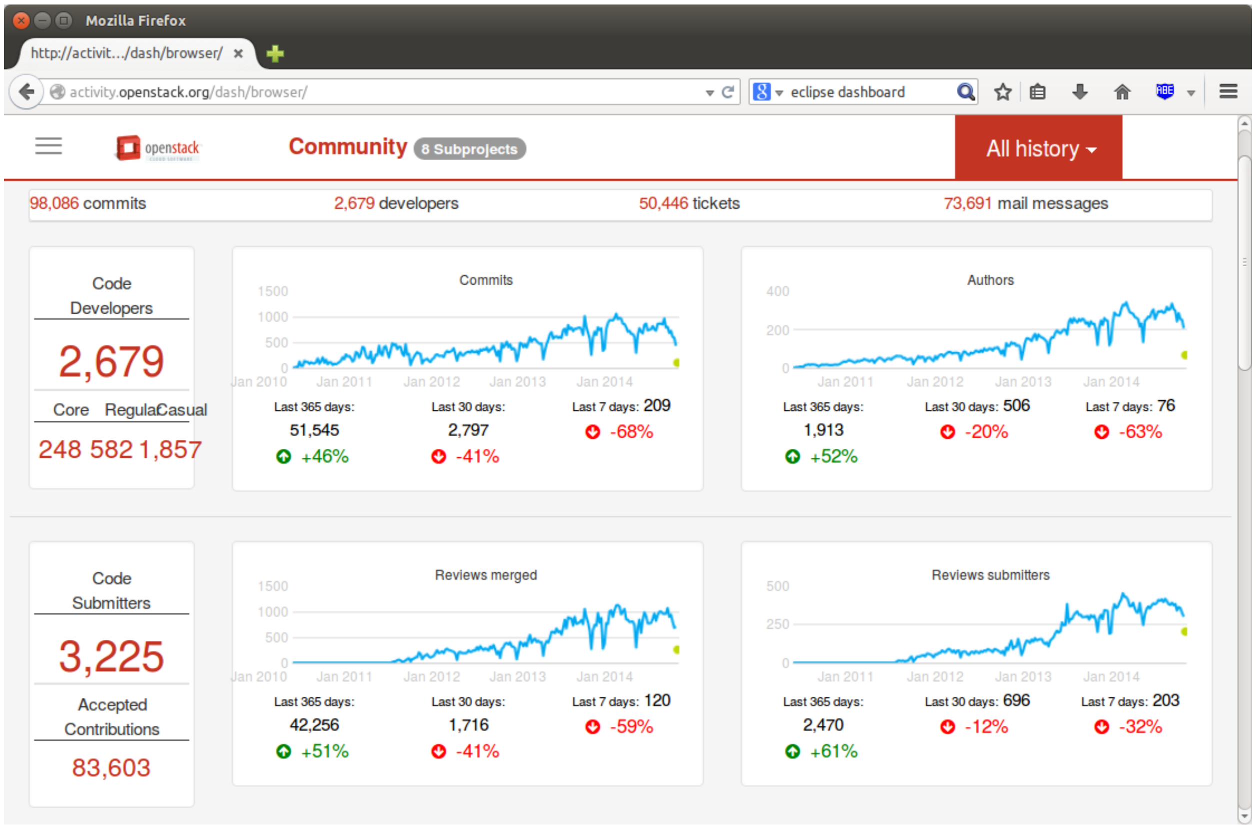
Collects a large number of object-oriented metrics from code.

4.1 Issues

There are problems such as different tools report different values for the same metric (Lincke et al., 2008)

It is well-know that the NASA datasets have some problems:

- (Gray et al., 2011a) The misuse of the NASA metrics data program data sets for automated software defect prediction
- (Shepperd et al., 2013) Data Quality: Some Comments on the NASA Software Defect Datasets



98,086 commits 2,679 developers 50,446 tickets 73,691 mail messages

Code Developers
2,679
Core Regular Casual
248 582 1,857

Commits

Last 365 days:	Last 30 days:	Last 7 days: 209
51,545	2,797	-68%
+46%	-41%	

Authors

Last 365 days:	Last 30 days: 506	Last 7 days: 76
1,913	-20%	-63%
+52%		

Code Submitters
3,225
Accepted Contributions
83,603

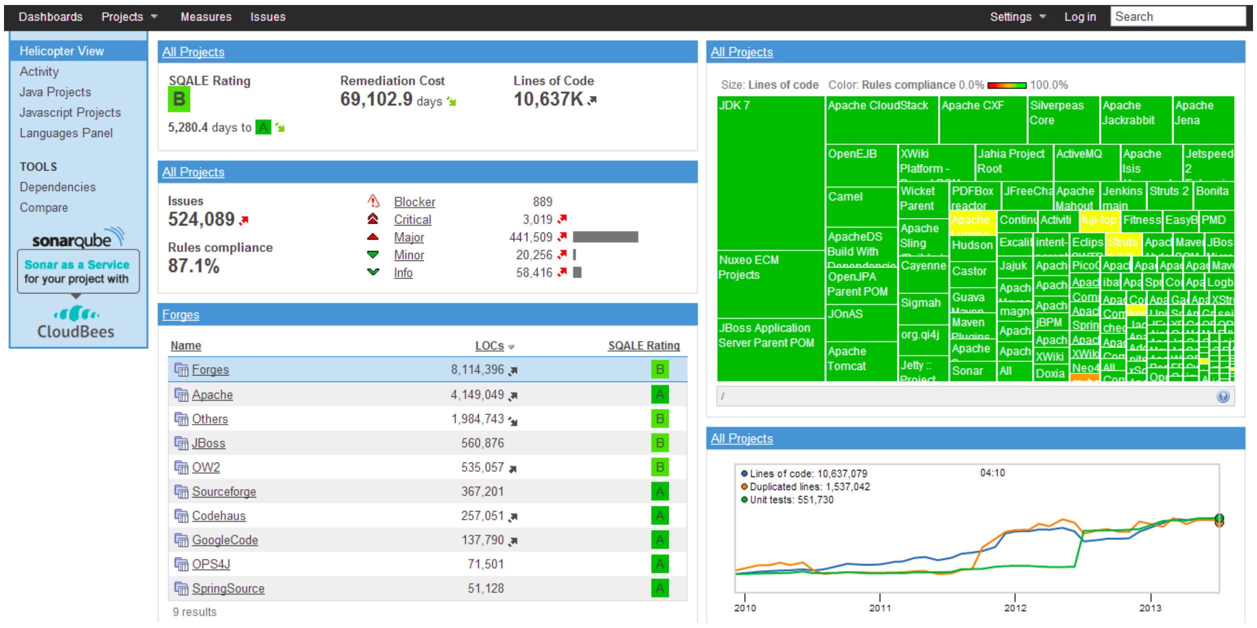
Reviews merged

Last 365 days:	Last 30 days:	Last 7 days: 120
42,256	1,716	-59%
+51%	-41%	

Reviews submitters

Last 365 days:	Last 30 days: 696	Last 7 days: 203
2,470	-12%	-32%
+61%		

Figure 4.2: Grimoire



All Projects

SQALE Rating B
5,280.4 days to A

Remediation Cost 69,102.9 days

Lines of Code 10,637K

All Projects

Issues 524,089

Rules compliance 87.1%

Blocker	889
Critical	3,019
Major	441,509
Minor	20,256
Info	58,416

Forges

Name	LOCs	SQALE Rating
Forges	8,114,396	B
Apache	4,149,049	A
Others	1,984,743	B
JBoss	560,876	B
OW2	535,057	B
Sourceforge	367,201	A
Codehaus	257,051	A
GoogleCode	137,790	A
OPS4J	71,501	A
SpringSource	51,128	A

9 results

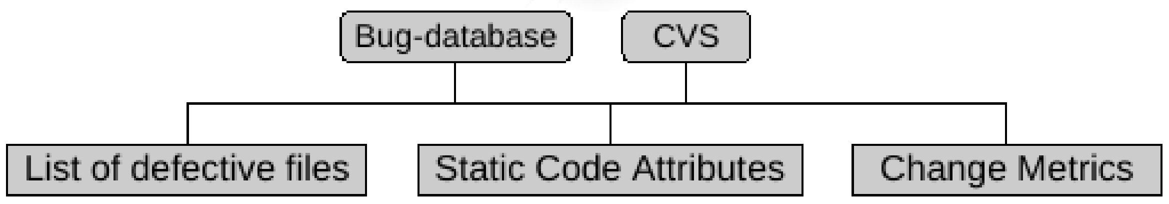
Figure 4.3: SonarQube

Chapter 5

Metrics in Software Enginering Prediction

There are many types of metrics that can be easily collected from source code or software configuration management systems.

Most work have focused on static metrics from code but recently other metrics



(Source: Moser et al)

Change Metrics describe how a file changed in the past, e.g., REVISIONS, REFACTORINGS, BUGFIXES, LOC_ADDED, etc.

5.1 Complexity Metrics

These metrics have been used for quality assurance during:

- Development to obtain quality measures, code reviews etc.
- Testing to focus and prioritize testing effort, improve efficiency etc.
- Maintenance as indicators of comprehensibility of the modules etc.

Generally, the developers or maintainers use rules of thumb or threshold values to keep modules, methods etc. within certain range. For example, if the cyclomatic complexity ($v(g)$) of a module is between 1 and 10, it is considered to have a very low risk of being defective; however, any value greater than 50 is considered to have an unmanageable complexity and risk. For the essential complexity ($ev(g)$), the threshold suggested is 4 etc. Although these metrics have been used for long time, there are no clear thresholds, for example, although McCabe suggests a threshold of 10 for $v(g)$, NASA's in–house studies for this metric concluded that a threshold of 20 can be a better predictor of a module being defective.

Several authors have studied the relation between lines of code and defects, for example, (Zhang, 2009)

5.1.1 McCabe's Complexity Measures

McCabe defined a set of complexity measures based on the graph of a program (McCabe, 1976):

- Cyclomatic complexity $v(g)$ is number of the control graph of the function $v(g) = e - n + 2$, where e represents the number of edges and n the number of nodes.
- Module Design Complexity $iv(G)$
- Essential Complexity $ev(G)$ is a measure of the degree to which a module contains unstructured constructs

5.1.2 Halstead-s Software Science

Late 70s, Halstead developed a set of metrics based on (Halstead, 1977):

- Operators, keywords such as `if`, `while`, `for`, operators `+`, `=`, `AND`, etc.
- Operands, program variables and constants.

All metrics are calculated using the following:

- n_1, number of distinct operators
- N_1, total number of operators
- n_2, number of distinct operands
- N_2, the total number of operands

For example, given the following excerpt:

```
if (MAX < 2) {
    a = b * MAX;
    System.out.print(a);
}
```

We obtain the following values:

- $n1 = 6$ (`if, { }, System.out.print(), =, *, $<$`)
- $N1 = 6$ (`if, { }, System.out.print(), =, *, $<$`)
- $n2 = 4$ (`MAX, a, b, 2`)
- $N2 = 6$ (`MAX, 2, a, b, MAX, a`)

Using this four variable, several metrics are calculated.

- Program vocabulary ($n = n_1 + n_2$), as a measure of complexity, the less number of elements, the simpler it should be the program.
- Program length, $N = N_1 + N_2$, as a proxy for size, the larger it is, the harder it is to understand it. Program length can be estimated with $\hat{N} = N_1 log_2 n_1 + N_2 log_2 n_2$ (calculated program length)
- Volumen, $V = N \cdot log_2(n)$, number of bits needed to codify a program. While *length* is just the count of operators and operands, the *volumen* considers the number of distint operators. Given two same size programs, the one of with more distint operators will be harder to understand, i.e, greater volumen.
- Level, $L = V'/V$, where V' is the potential volumen (minimun number of operands and operators needed and V. As V' is difficult to calculate, it can be approximated with $\hat{L} = \frac{n_1'}{n_1} \cdot \frac{n_2}{N_2}$, where n_1' is the minimun number of distint operators.

- Mental effort, $E = V/L$, where L depends on the programming language. It can also be calculated as $E = \frac{n_1 N_2 N log_2 n}{2n_2}$
- Time, $T = E/S$, time to develop a program in seconds, being S the number of metal discrimitaions with a value between $5 \leq S \leq 20$.
- Inteligence, $i = (2n_2/n_1N_2) \cdot (N_1 + N_2)log_2(n_1 + n_2)$

Although these metrics are still highly used, they are also criticised, e.g., the mental discriminations is very hard to define and calculate.

5.2 Object-Oriented Metrics

chidamber and Kemerer introduced a set of OO complexity Metrics that is being highly used (Chidamber and Kemerer, 1994).

- Coupling Between Objects (CBO) for a class is a count of the number of other classes to which it is coupled. Coupling between two classes is said to occur when one class uses methods or variables of another class. COB is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends.
- Depth of Inheritance Tree (DIT) measures the maximum level of the inheritance hierarchy of a class; the root of the inheritance tree inherits from no class and is at level zero of the inheritance tree. Direct count of the levels of the levels in an inheritance hierarchy.
- Number of Children (NOC) is the number of immediate subclasses subordinate to a class in the hierarchy. NOC counts the number of subclasses belonging to a class. According to C&K, NOC indicate the level of reuse, the likelihood of improper abstraction and as possible indication of the level of testing required.
- Response For a Class (RFC) is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. According to C&K, RFC is a measure of the complexity of a class through the number of methods and the amount of communication with other classes.
- Weighted Methods per Class (WMC) measures the complexity of an individual class. If all methods are considered equally complex, then WMC is simply the number of methods defined in each class. C&K suggests that WMC is intended to measure the complexity of a class. Therefore, it is an indicator of the effort needed to develop and maintain a class.
- Lack of Cohesion in Methods (LCOM) measures the extent to which methods reference the classes instance data.
- Etc.

(Jureczko and Spinellis, 2010) Jurescko and Spinellis

(Bansiya and Davis, 2002) A hierarchical model for object-oriented design quality assessment

5.3 Churn and Process Metrics

Churned LOC is usually the number of source code lines added, deleted or changes to a component between a baseline version and a new version.

Authors can also count the number of files created/deleted and the time span between those changes.

Madeyski and Jurescko present a taxonomy of process metrics (Madeyski and Jureczko, 2015):

Metric	Source code	Developers	Defects
Number of Revisions (NR)	X		
Number of Modified Lines (NML)	X		
Is New (IN)	X		
Number of Distinct Commiters (NDC)	X	X	
Number of Defects in Previous Version (NDPV)	X		X

A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction (Moser et al., 2008)

Metrics	Definition
Commits	No. of commits a file has gone through
Refactorings	No. of times a file has been refactored
Bugfixes	No. of times a file has been involved in bug fixing
Authors	No. of distinct authors that made commits to the file
LoC-Added	No. of lines of code added to the file
Max-LoC-Added	Max no. of lines of code added for all commits
Avg-LoC-Added	Avg lines of code added per commit
LoC-DELETED	No. of lines of code deleted from the file
Max-LoC-Added	Max no. of lines of code deleted for all commits
Avg-LoC-Added	Avg lines of code deleted per commit
CodeChurn	Sum of all commits (added lines of code deleted lines of code)
Max-CodeChurn	Maximum CODECHURN for all commits
Avg-CodeChurn	Average CODECHURN per commit
Max-Changeset	Maximum number of files committed together with the repository
Avg-Changeset	Average number of files committed together with the repository
Weighted-Age	Age of a file normalized by added lines of code

A further description of typical metrics used (Krishnan et al., 2011):

5.4 Defect Standards

5.4.1 IBM's Orthogonal Defect Classificatoin (ODC)

ODC Activity	ODC Trigger	ODC Impact
Design review	Design conformance	Installability
Code inspection	Logic, flow	Standards
Unit test	Backward compatibility	Serviceability
Build	Lateral compatibility	Integrity
BVT	Concurrency	Security
CVT	Internal document	Migration
SVT	Language dependency	Reliability
IVT	Side effect	Performance
GVT	Rare situations	Documentation
TVT	Simple path	Requirements
DBCS IVT	Complex path	Maintenance
Performance	Coverage	Usability
Scalability	Variation	Accessibility

ODC Activity	ODC Trigger	ODC Impact
ID review	Sequencing	Capability
GUI review	Interaction	
Acceptance	Workload, stress	
Beta	Recovery, exception	
Start, restart		
	Hardware configuration	
	Software configuration	
	Screen text, characters	
	Navigation	
	Widget	GUI behavior

The impact is classified in the following categories:

ODC Impact	Description
Installability	The ability of the customer to prepare and place the software in position for use (not include Usability).
Integrity/Security	The protection of systems, programs, and data from inadvertent or malicious destruction, alteration, or disclosure.
Performance	The speed of the software as perceived by the customer and the customer's end users, in terms of their ability to perform their tasks.
Maintenance	The ease of applying preventive or corrective fixes to the software. An example would be that the fixes can not be applied due to a bad medium. Another example might be that the application of maintenance requires a great deal of manual effort, or is calling many pre- or co-requisite requisite maintenance.
Serviceability	The ability to diagnose failures easily and quickly, with minimal impact to the customer.
Migration	The ease of upgrading to a current release, particularly in terms of the impact on existing customer data and operations. This would include planning for migration, where a lack of adequate documentation makes this task difficult. It would also apply in those situations where a new release of an existing product introduces changes effecting the external interfaces between the product and the customer's applications.
Documentation	The degree to which the publication aids provided for understanding the structure and intended uses of the software are correct and complete.
Usability	The degree to which the software and publication aids enable the product to be easily understood and conveniently employed by its end user.
Standards	The degree to which the software complies with established pertinent standards.
Reliability	The ability of the software to consistently perform its intended function without unplanned interruption. Severe interruptions, such as ABEND and WAIT would always be considered reliability.
Requirements	A customer expectation, with regard to capability, which was not known, understood, or prioritized as a requirement for the current product or release. This value should be chosen during development for additions to the plan of record. It should also be selected, after a product is made generally available, when customers report that the capability of the function or product does not meet their expectation.

ODC Impact	Description
Accessibility	Ensuring that successful access to information and use of information technology is provided to people who have disabilities.
Capability	The ability of the software to perform its intended functions, and satisfy known requirements, where the customer is not impacted in any of the previous categories.

These impact categories have been used to classify defects (Huang et al., 2011)

Part III

Machine Learning in Defect Prediction

Chapter 6

Supervised Classification

A classification problem can be defined as the induction, from a dataset $\mathcal{D}$, of a classification function ψ that, given the attribute vector of an instance/example, returns a class c. A regression problem, on the other hand, returns an numeric value.

Dataset, $\mathcal{D}$, is typically composed of n attributes and a class attribute C.

Att_1	...	Att_n	$Class$
a_{11}	...	a_{1n}	c_1
a_{21}	...	a_{2n}	c_2
...	...	...	...
a_{m1}	...	a_{mn}	c_m

Columns are usually called *attributes* or *features*. Typically, there is a *class* attribute, which can be numeric or discrete. When the class is numeric, it is a regression problem. With discrete values, we can talk about binary classification or multiclass (multinomial classification) when we have more than three values.

We have multiple types of models such as *classification trees*, *rules*, *neural networks*, and *probabilistic classifiers* that can be used to classify instances.

Fernandez et al provide an extensive comparison of 176 classifiers using the UCI dataset (Fernández-Delgado et al., 2014).

We will show the use of different classification techniques in the problem of defect prediction as running example. In this example,the different datasets are composed of classical metrics (*Halstead* or *McCabe* metrics) based on counts of operators/operands and like or object-oriented metrics (e.g. Chidamber and Kemerer) and the class attribute indicating whether the module or class was defective.

Most works in defect predicition have compared and analysed different classifiers with different datasets. Some relevant works include:

6.1 Classification Trees

There are several packages for inducing classification trees, for example with the party package (recursive partitioning):

```r
library(foreign) # To load arff file
library(party) # Build a decision tree
library(caret)
```

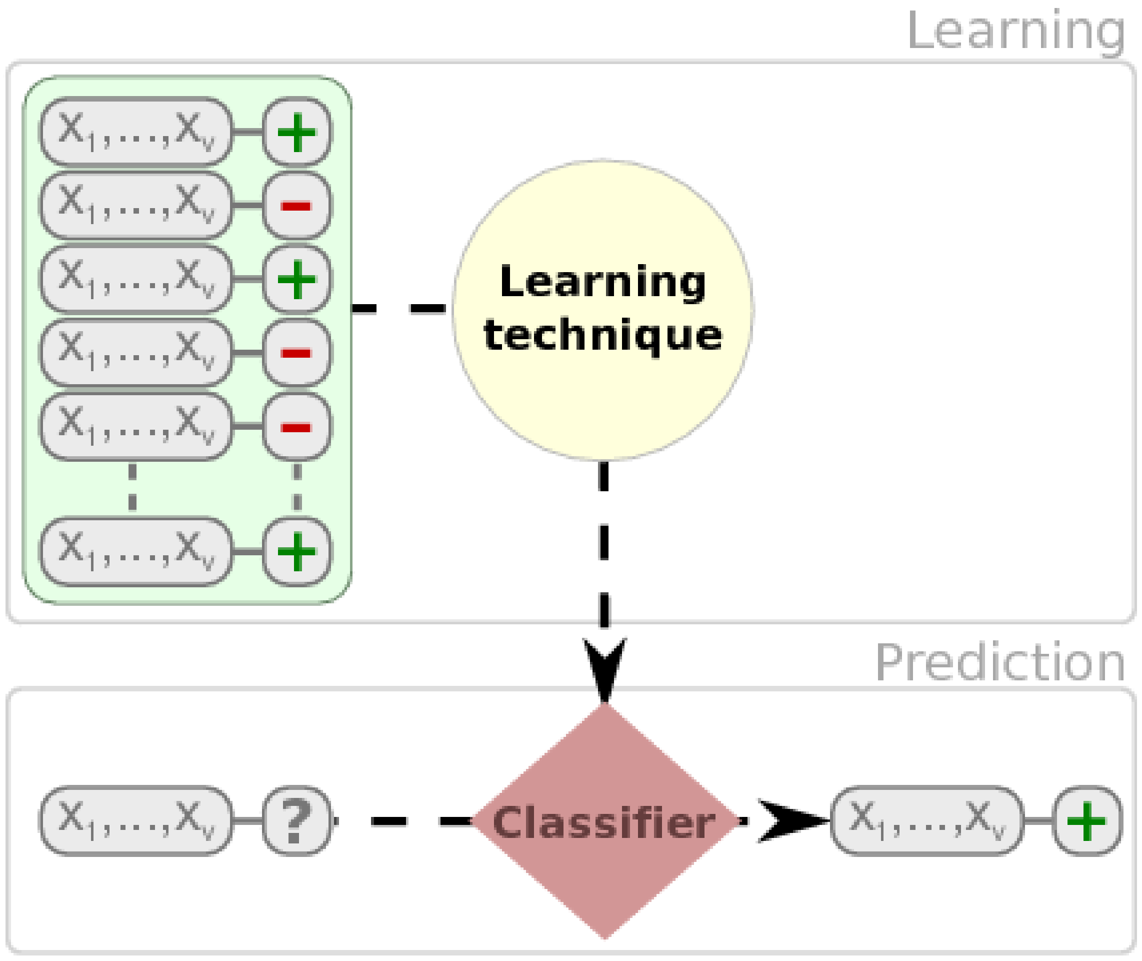


Figure 6.1: Supervised Classification

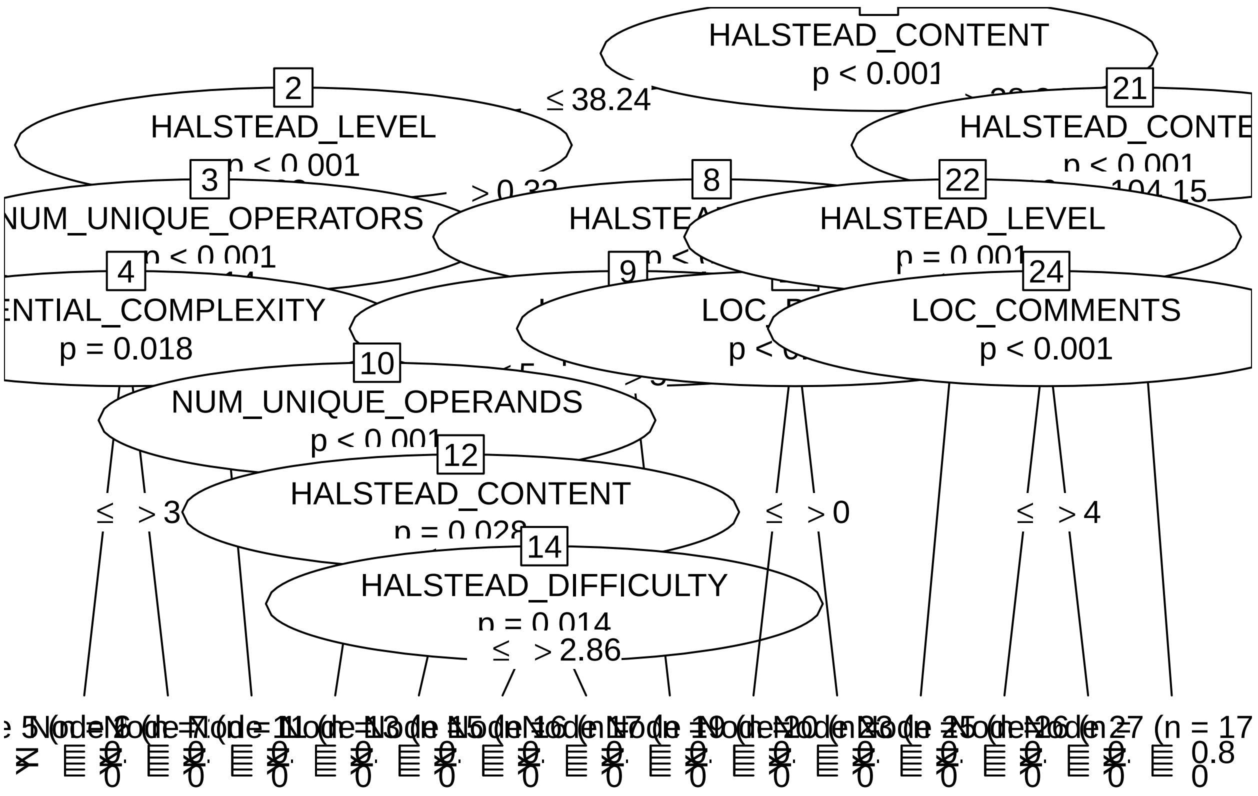
```r
jm1 <- read.arff("./datasets/defectPred/D1/JM1.arff")
str(jm1)
```

```
## 'data.frame':    9593 obs. of  22 variables:
##  $ LOC_BLANK            : num  447 37 11 106 101 67 105 18 39 143 ...
##  $ BRANCH_COUNT         : num  826 29 405 240 464 187 344 47 163 67 ...
##  $ LOC_CODE_AND_COMMENT : num  12 8 0 7 11 4 9 0 1 7 ...
##  $ LOC_COMMENTS         : num  157 42 17 344 75 1 40 10 6 49 ...
##  $ CYCLOMATIC_COMPLEXITY: num  470 19 404 127 263 94 207 24 94 34 ...
##  $ DESIGN_COMPLEXITY    : num  385 19 2 105 256 63 171 13 67 25 ...
##  $ ESSENTIAL_COMPLEXITY : num  113 6 1 33 140 27 58 1 3 1 ...
##  $ LOC_EXECUTABLE       : num  2824 133 814 952 1339 ...
##  $ HALSTEAD_CONTENT     : num  210 108 101 218 106 ...
##  $ HALSTEAD_DIFFICULTY  : num  384.4 46.3 206 215.2 337.4 ...
##  $ HALSTEAD_EFFORT      : num  31079782 232044 4294926 10100867 12120796 ...
##  $ HALSTEAD_ERROR_EST   : num  26.95 1.67 6.95 15.65 11.98 ...
##  $ HALSTEAD_LENGTH      : num  8441 685 2033 5669 4308 ...
##  $ HALSTEAD_LEVEL       : num  0 0.02 0 0 0 0.02 0 0.03 0.01 0.02 ...
##  $ HALSTEAD_PROG_TIME   : num  1726655 12891 238607 561159 673378 ...
##  $ HALSTEAD_VOLUME      : num  80843 5009 20848 46944 35928 ...
##  $ NUM_OPERANDS         : num  3021 295 813 2301 1556 ...
##  $ NUM_OPERATORS        : num  5420 390 1220 3368 2752 ...
##  $ NUM_UNIQUE_OPERANDS  : num  609 121 811 262 226 167 279 47 117 355 ...
##  $ NUM_UNIQUE_OPERATORS : num  155 38 411 49 98 27 105 18 52 23 ...
##  $ LOC_TOTAL            : num  3442 222 844 1411 1532 ...
##  $ Defective            : Factor w/ 2 levels "N","Y": 2 2 2 2 2 2 2 2 1 2 ...
```

```r
# Stratified partition (training and test sets)
set.seed(1234)
inTrain <- createDataPartition(y=jm1$Defective,p=.60,list=FALSE)
jm1.train <- jm1[inTrain,]
jm1.test <- jm1[-inTrain,]

jm1.formula <- jm1$Defective ~ . # formula approach: defect as dependent variable and the rest as indep
jm1.ctree <- ctree(jm1.formula, data = jm1.train)

# predict on test data
pred <- predict(jm1.ctree, newdata = jm1.test)
# check prediction result
table(pred, jm1.test$Defective)
```

```
##
## pred    N    Y
##    N   82    3
##    Y 3051  700
```

```r
plot(jm1.ctree)
```

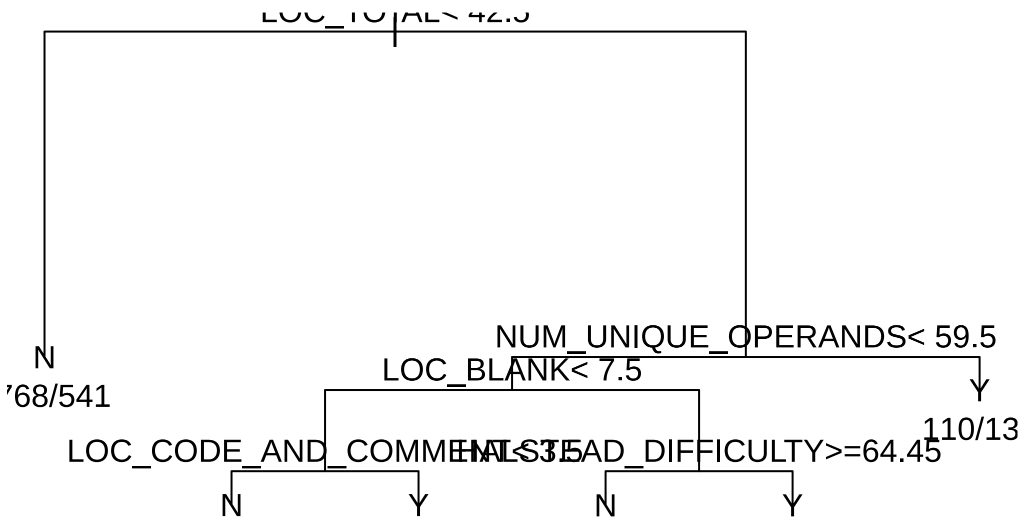


Using the C50 package, there are two ways, specifying train and testing

```
library(C50)
require(utils)
# c50t <- C5.0(jm1.train[,-ncol(jm1.train)], jm1.train[,ncol(jm1.train)])
c50t2 <- C5.0(Defective ~ ., jm1.train)
summary(c50t)
plot(c50t)
c50tPred <- predict(c50t, jm1.train)
# table(c50tPred, jm1.train$Defective)
```

Using the ‘rpart’ package

```
# Using the 'rpart' package
library(rpart)
jm1.rpart <- rpart(Defective ~ ., data=jm1.train, parms = list(prior = c(.65,.35), split = "information
# par(mfrow = c(1,2), xpd = NA)
plot(jm1.rpart)
text(jm1.rpart, use.n = TRUE)
```


LOC_TOTAL< 42.5
N
'68/541
NUM_UNIQUE_OPERANDS< 59.5
LOC_BLANK< 7.5
110/13
LOC_CODE_AND_COMMENT< 3.5
HALSTEAD_DIFFICULTY>=64.45
N
Y
N
Y

```
jm1.rpart
```

```
## n= 5757
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 5757 2014.95000 N (0.6500000 0.3500000)
##    2) LOC_TOTAL< 42.5 4309 1032.28000 N (0.7439560 0.2560440) *
##    3) LOC_TOTAL>=42.5 1448  742.67870 Y (0.4304514 0.5695486)
##      6) NUM_UNIQUE_OPERANDS< 59.5 1206  655.11750 Y (0.4726955 0.5273045)
##       12) LOC_BLANK< 7.5 449  209.89060 N (0.5624895 0.4375105)
##         24) LOC_CODE_AND_COMMENT< 3.5 412  171.72870 N (0.5988063 0.4011937) *
##         25) LOC_CODE_AND_COMMENT>=3.5 37   13.53220 Y (0.2617743 0.7382257) *
##       13) LOC_BLANK>=7.5 757  385.26960 Y (0.4251579 0.5748421)
##         26) HALSTEAD_DIFFICULTY>=64.45 55   13.35668 N (0.7409751 0.2590249) *
##         27) HALSTEAD_DIFFICULTY< 64.45 702  347.06100 Y (0.4061023 0.5938977) *
##      7) NUM_UNIQUE_OPERANDS>=59.5 242   87.56126 Y (0.2579656 0.7420344) *
```

```
library(rpart.plot)
# asRules(jm1.rpart)
# fancyRpartPlot(jm1.rpart)
```

6.2 Rules

C5 Rules

```
library(C50)
c50r <- C5.0(jm1.train[,-ncol(jm1.train)], jm1.train[,ncol(jm1.train)], rules = TRUE)
summary(c50r)
```

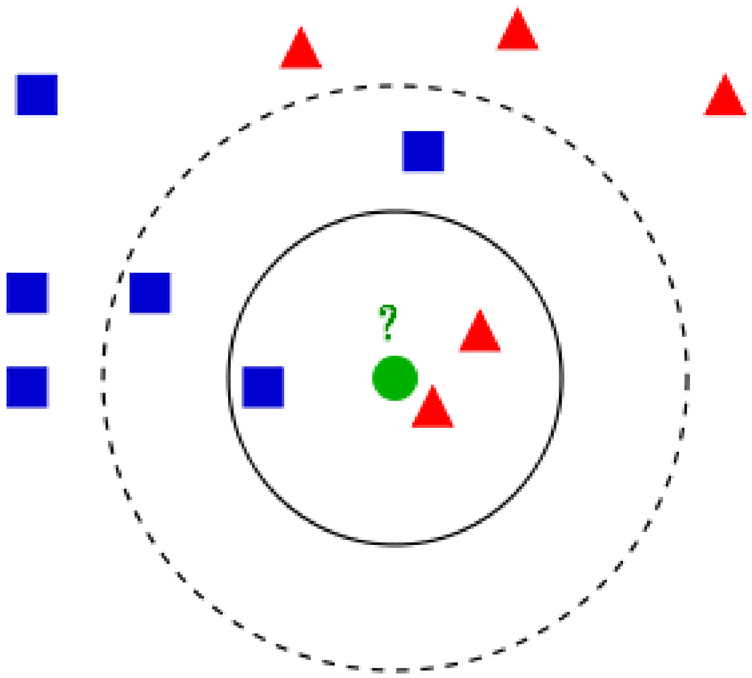
```
##
## Call:
## C5.0.default(x = jm1.train[, -ncol(jm1.train)], y =
##  jm1.train[, ncol(jm1.train)], rules = TRUE)
##
##
```

```
## C5.0 [Release 2.07 GPL Edition]      Sat Jul  1 11:30:28 2017
## -------------------------------
##
## Class specified by attribute `outcome'
##
## Read 5757 cases (22 attributes) from undefined.data
##
## Rules:
##
## Rule 1: (5512/923, lift 1.0)
##  CYCLOMATIC_COMPLEXITY <= 67
##  NUM_UNIQUE_OPERANDS <= 59
##  ->  class N  [0.832]
##
## Rule 2: (11/1, lift 4.6)
##  LOC_BLANK > 3
##  DESIGN_COMPLEXITY > 3
##  LOC_EXECUTABLE > 31
##  LOC_EXECUTABLE <= 33
##  LOC_TOTAL <= 42
##  ->  class Y  [0.846]
##
## Rule 3: (25/5, lift 4.2)
##  CYCLOMATIC_COMPLEXITY > 67
##  ->  class Y  [0.778]
##
## Rule 4: (242/110, lift 3.0)
##  NUM_UNIQUE_OPERANDS > 59
##  LOC_TOTAL > 42
##  ->  class Y  [0.545]
##
## Default class: N
##
##
## Evaluation on training data (5757 cases):
##
##          Rules
##    ----------------
##      No      Errors
##
##       4 1025(17.8%)   <<
##
##
##     (a)   (b)    <-classified as
##    ----  ----
##    4589   112    (a): class N
##     913   143    (b): class Y
##
##
##  Attribute usage:
##
##   99.95% NUM_UNIQUE_OPERANDS
##   96.18% CYCLOMATIC_COMPLEXITY
##    4.39% LOC_TOTAL
##    0.19% LOC_BLANK
##    0.19% DESIGN_COMPLEXITY
##    0.19% LOC_EXECUTABLE
##
##
## Time: 0.2 secs
```

```
# c50rPred <- predict(c50r, jm1.train)
# table(c50rPred, jm1.train$Defective)
```

6.3 Distanced-based Methods

In this case, there is no model as such. Given a new instance to classify, this approach finds the closest k-neighbours to the given instance.



(Source: Wikipedia - https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm)

```
library(class)
m1 <- knn(train=jm1.train[,-22], test=jm1.test[,-22], cl=jm1.train[,22], k=3)

table(jm1.test[,22],m1)
```

```
##    m1
##        N    Y
##   N 2827  306
##   Y  553  150
```

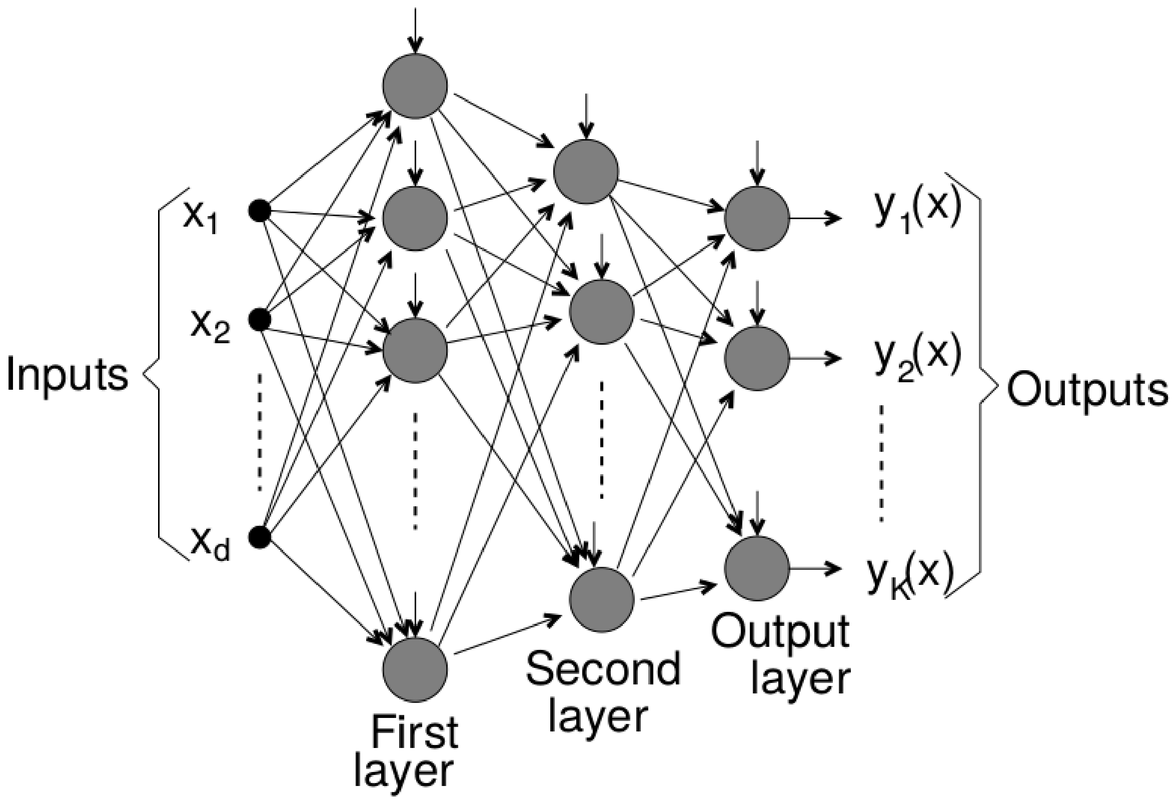


Figure 6.2: Neural Networks

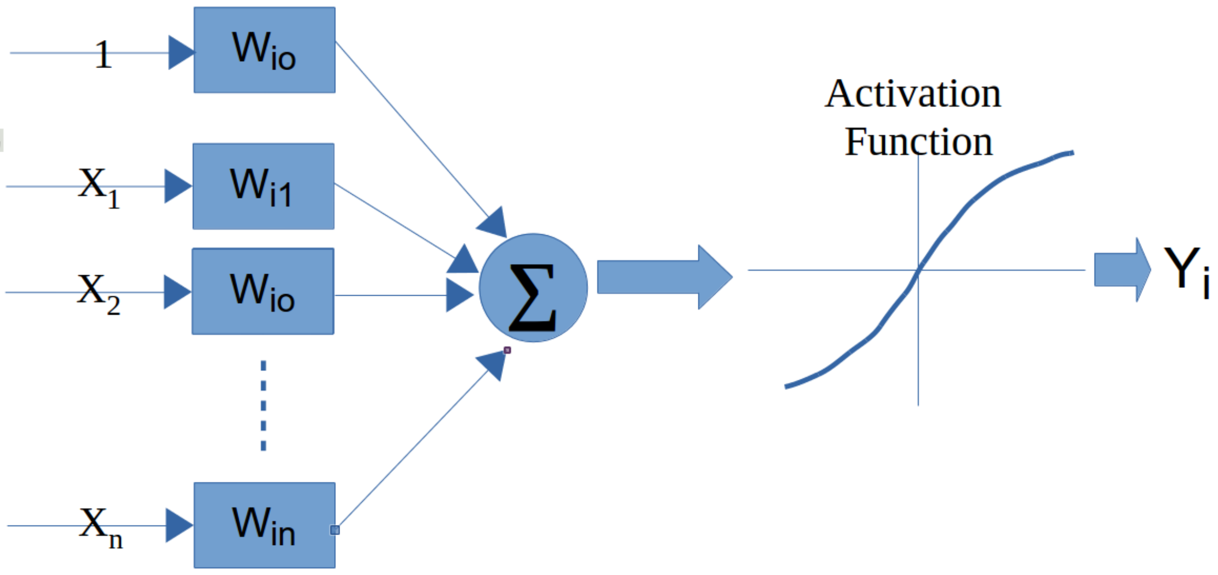
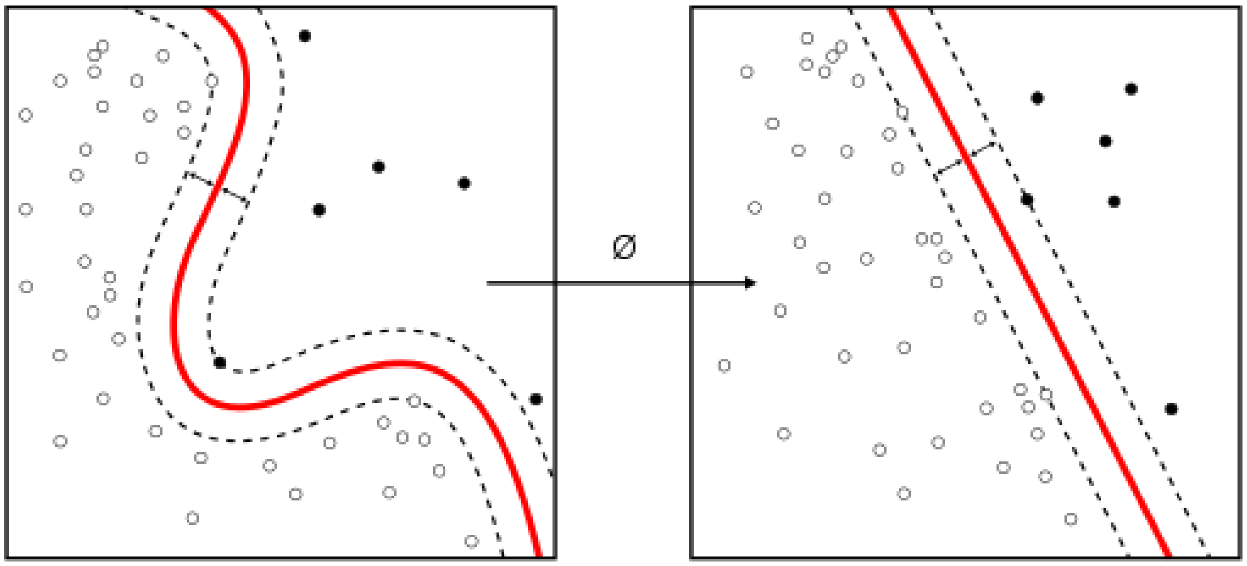


Figure 6.3: Neural Networks

6.4 Neural Networks

6.5 Support Vector Machine



(Source: wikipedia https://en.wikipedia.org/wiki/Support_vector_machine)

6.6 Probabilistic Methods

6.6.1 Naive Bayes

Probabilistic graphical model assigning a probability to each possible outcome $p(C_k, x_1, \ldots, x_n)$

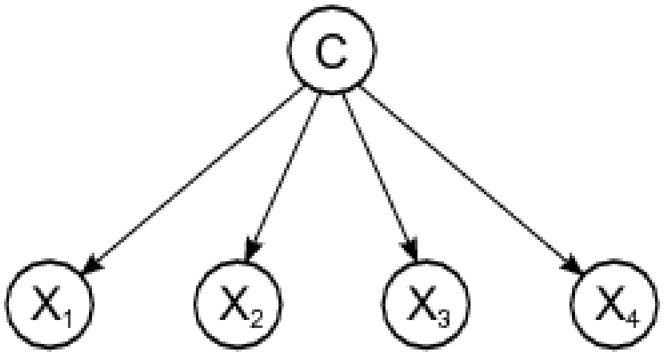


Figure 6.4: Naive Bayes

Using the `klaR` package with `caret`:

```
library(caret)
library(klaR)
```

```
## Loading required package: MASS
```

```
model <- NaiveBayes(Defective ~ ., data = jm1.train)
predictions <- predict(model, jm1.test[,-22])
confusionMatrix(predictions$class, jm1.test$Defective)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    N    Y
##          N 2990  548
##          Y  143  155
##
##                Accuracy : 0.8199
##                  95% CI : (0.8073, 0.8319)
##     No Information Rate : 0.8167
##     P-Value [Acc > NIR] : 0.3168
##
##                   Kappa : 0.2251
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.9544
##             Specificity : 0.2205
##          Pos Pred Value : 0.8451
##          Neg Pred Value : 0.5201
##              Prevalence : 0.8167
##          Detection Rate : 0.7795
##    Detection Prevalence : 0.9223
##       Balanced Accuracy : 0.5874
##
##        'Positive' Class : N
##
```

Using the e1071 package:

```
library (e1071)
n1 <-naiveBayes(jm1.train$Defective ~ ., data=jm1.train)

# Show first 3 results using 'class'
head(predict(n1,jm1.test, type = c("class")),3) # class by default
```

```
## [1] N Y Y
## Levels: N Y
```

```
# Show first 3 results using 'raw'
head(predict(n1,jm1.test, type = c("raw")),3)
```

```
##                 N         Y
## [1,] 1.0000000000 0.0000000
## [2,] 0.0000000000 1.0000000
## [3,] 0.0007540951 0.9992459
```

There are other variants such as TAN and KDB that do not assume the independece condition allowin us more complex structures.

A comprehensice comparison of

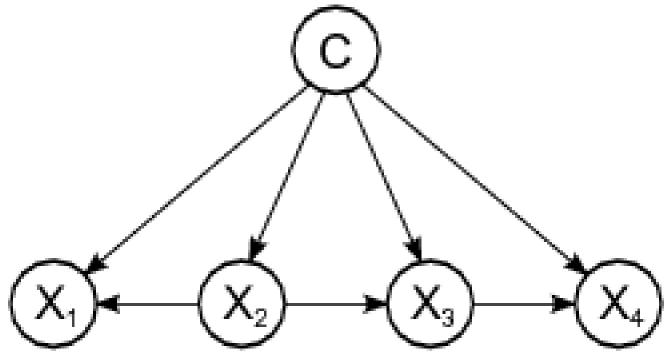


Figure 6.5: Naive Bayes

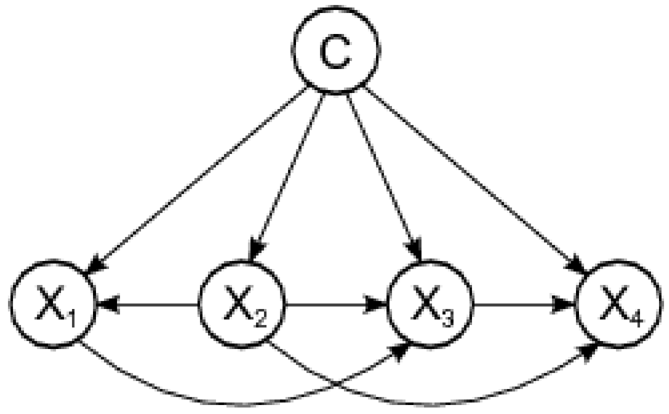


Figure 6.6: Naive Bayes

6.7 Linear Discriminant Analysis (LDA)

One classical approach to classification is Linear Discriminant Analysis (LDA), a generalization of Fisher's linear discriminant, as a method used to find a linear combination of features to separate two or more classes.

```
ldaModel <- train (Defective ~ ., data=jm1.train, method="lda", preProc=c("center","scale"))
ldaModel
```

```
## Linear Discriminant Analysis
##
## 5757 samples
##   21 predictor
##    2 classes: 'N', 'Y'
##
## Pre-processing: centered (21), scaled (21)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 5757, 5757, 5757, 5757, 5757, 5757, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.8184746  0.1371754
```

We can observe that we are training our model using `Defective ~ .` as a formula were `Defective` is the class variable separed by `~` and the ´.´ means the rest of the variables. Also, we are using a filter for the training data to (preProc) to center and scale.

Also, as stated in the documentation about the `train` method : > http://topepo.github.io/caret/training.html

```
ctrl <- trainControl(method = "repeatedcv",repeats=3)
ldaModel <- train (Defective ~ ., data=jm1.train, method="lda", trControl=ctrl, preProc=c("center","sca

ldaModel
```

```
## Linear Discriminant Analysis
##
## 5757 samples
##   21 predictors
##    2 classes: 'N', 'Y'
##
## Pre-processing: centered (21), scaled (21)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 5182, 5181, 5181, 5181, 5181, 5181, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.8167443  0.1308431
```

Instead of accuracy we can activate other metrics using `summaryFunction=twoClassSummary` such as `ROC`, `sensitivity` and `specificity`. To do so, we also need to speficy `classProbs=TRUE`.

```
ctrl <- trainControl(method = "repeatedcv",repeats=3, classProbs=TRUE, summaryFunction=twoClassSummary)
ldaModel3xcv10 <- train (Defective ~ ., data=jm1.train, method="lda", trControl=ctrl, preProc=c("center

ldaModel3xcv10
```

```
## Linear Discriminant Analysis
##
```

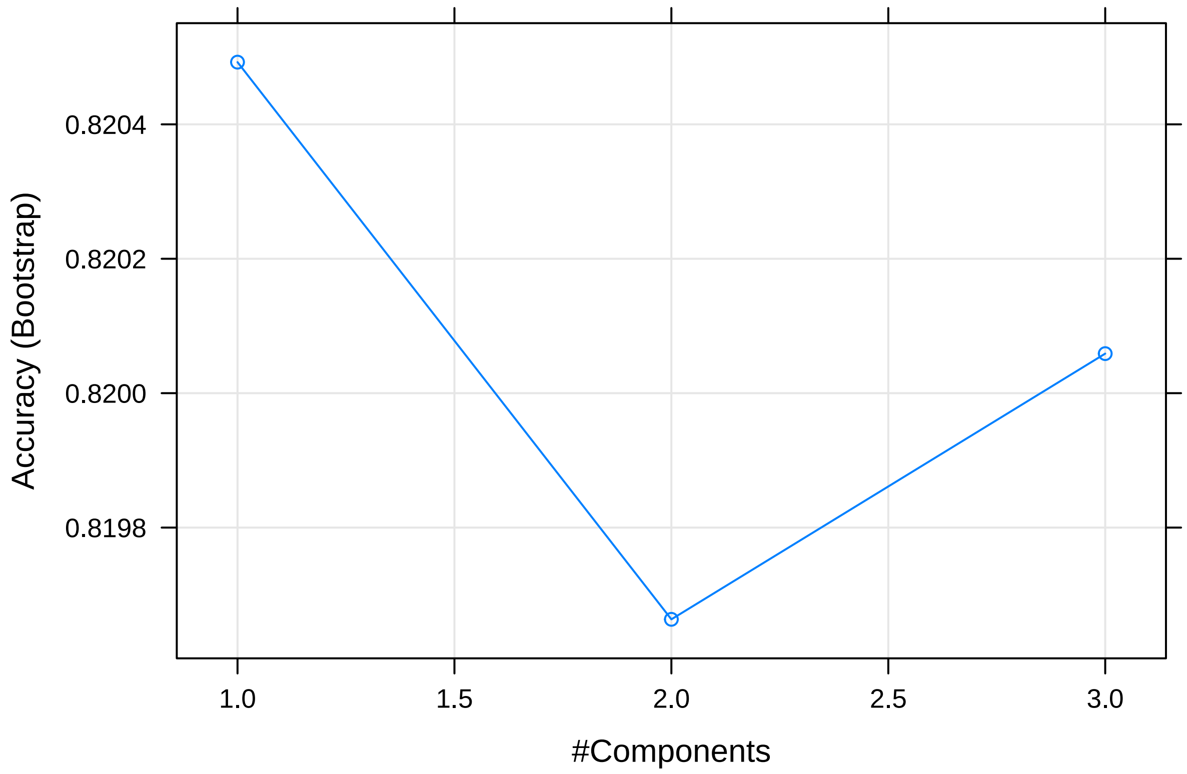
```
## 5757 samples
##   21 predictors
##    2 classes: 'N', 'Y'
##
## Pre-processing: centered (21), scaled (21)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 5181, 5182, 5180, 5181, 5182, 5181, ...
## Resampling results:
##
##   ROC        Sens       Spec
##   0.7073554  0.9731275  0.1189847
```

Most methods have parameters that need to be optimised and that is one of the

```
plsFit3x10cv <- train (Defective ~ ., data=jm1.train, method="pls", trControl=trainControl(classProbs=T

plsFit3x10cv
```

```
## Partial Least Squares
##
## 5757 samples
##   21 predictors
##    2 classes: 'N', 'Y'
##
## Pre-processing: centered (21), scaled (21)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 5757, 5757, 5757, 5757, 5757, 5757, ...
## Resampling results across tuning parameters:
##
##   ncomp  Accuracy   Kappa
##   1      0.8204925  0.06414865
##   2      0.8196635  0.08261024
##   3      0.8200588  0.08777031
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was ncomp = 1.
```

```
plot(plsFit3x10cv)
```

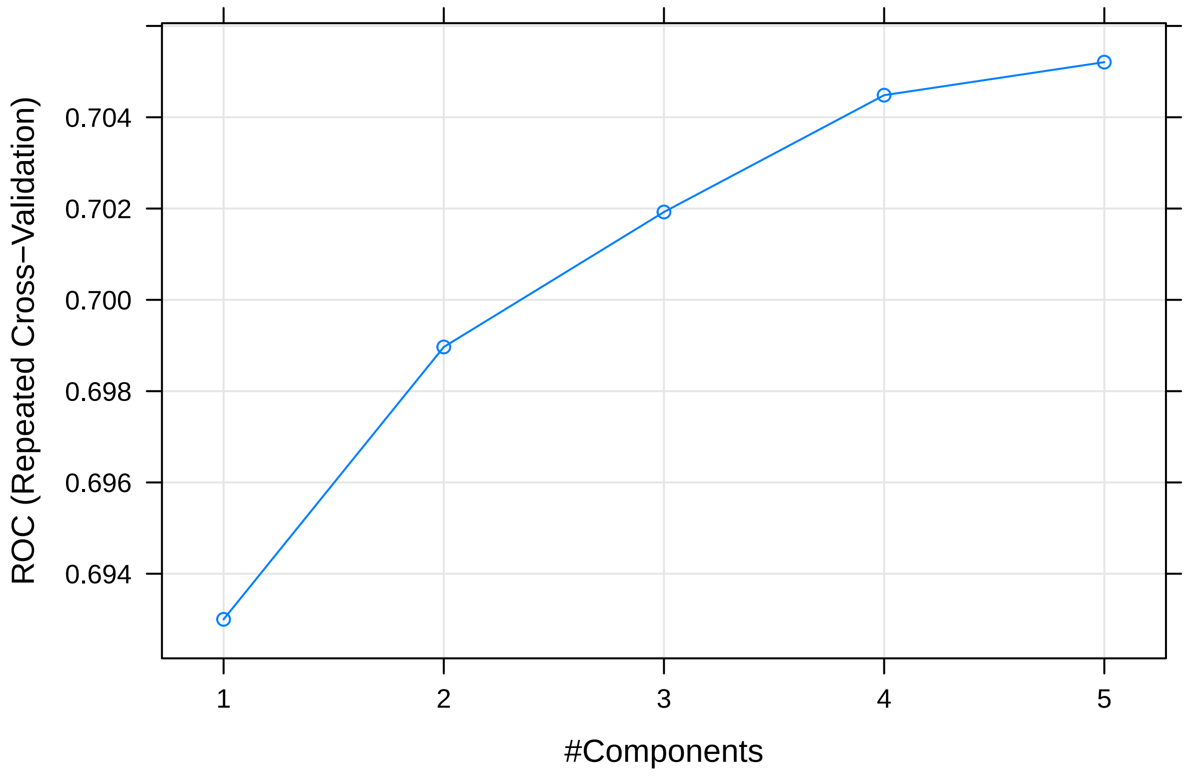


The parameter `tuneLength` allow us to specify the number values per parameter to consider.

```
plsFit3x10cv <- train (Defective ~ ., data=jm1.train, method="pls", trControl=ctrl, metric="ROC", tuneL

plsFit3x10cv
```

```
## Partial Least Squares
##
## 5757 samples
##   21 predictors
##    2 classes: 'N', 'Y'
##
## Pre-processing: centered (21), scaled (21)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 5182, 5182, 5181, 5181, 5181, 5181, ...
## Resampling results across tuning parameters:
##
##   ncomp  ROC        Sens       Spec
##   1      0.6930018  0.9951784  0.04482480
##   2      0.6989680  0.9904273  0.06441450
##   3      0.7019246  0.9892927  0.06441450
##   4      0.7044831  0.9887963  0.06693621
##   5      0.7052068  0.9884417  0.06756813
##
## ROC was used to select the optimal model using  the largest value.
## The final value used for the model was ncomp = 5.
```

```
plot(plsFit3x10cv)
```



Finally to predict new cases, `caret` will use the best classfier obtained for prediction.

```
plsProbs <- predict(plsFit3x10cv, newdata = jm1.test, type = "prob")
```

```
plsClasses <- predict(plsFit3x10cv, newdata = jm1.test, type = "raw")
confusionMatrix(data=plsClasses,jm1.test$Defective)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    N    Y
##          N 3108  643
##          Y   25   60
##
##                Accuracy : 0.8259
##                  95% CI : (0.8135, 0.8377)
##     No Information Rate : 0.8167
##     P-Value [Acc > NIR] : 0.07427
##
##                   Kappa : 0.1174
##  Mcnemar's Test P-Value : < 2e-16
##
##             Sensitivity : 0.99202
##             Specificity : 0.08535
##          Pos Pred Value : 0.82858
##          Neg Pred Value : 0.70588
##              Prevalence : 0.81674
##          Detection Rate : 0.81022
##    Detection Prevalence : 0.97784
##       Balanced Accuracy : 0.53868
```

```
##
##        'Positive' Class : N
##
```

6.7.1 Predicting the number of defects (numerical class)

From the Bug Prediction Repository (BPR) http://bug.inf.usi.ch/download.php

Some datasets contain CK and other 11 object-oriented metrics for the last version of the system plus categorized (with severity and priority) post-release defects. Using such dataset:

```
jdt <- read.csv("./datasets/defectPred/BPD/single-version-ck-oo-EclipseJDTCore.csv", sep=";")

# We just use the number of bugs, so we removed others
jdt$classname <- NULL
jdt$nonTrivialBugs <- NULL
jdt$majorBugs <- NULL
jdt$minorBugs <- NULL
jdt$criticalBugs <- NULL
jdt$highPriorityBugs <- NULL
jdt$X <- NULL

# Caret
library(caret)

# Split data into training and test datasets
set.seed(1)
inTrain <- createDataPartition(y=jdt$bugs,p=.8,list=FALSE)
jdt.train <- jdt[inTrain,]
jdt.test <- jdt[-inTrain,]
```

```
ctrl <- trainControl(method = "repeatedcv",repeats=3)
glmModel <- train (bugs ~ ., data=jdt.train, method="glm", trControl=ctrl, preProc=c("center","scale"))
glmModel
```

```
## Generalized Linear Model
##
## 798 samples
##  17 predictors
##
## Pre-processing: centered (17), scaled (17)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 718, 718, 718, 718, 719, 718, ...
## Resampling results:
##
##   RMSE       Rsquared
##   0.8411011  0.3855316
```

Others such as Elasticnet:

```
glmnetModel <- train (bugs ~ ., data=jdt.train, method="glmnet", trControl=ctrl, preProc=c("center","sca
```

```
## Loading required package: glmnet
```

```
## Loading required package: Matrix
```

```
## Loading required package: foreach
## Loaded glmnet 2.0-10
```

```r
glmnetModel
```

```
## glmnet
##
## 798 samples
##  17 predictors
##
## Pre-processing: centered (17), scaled (17)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 718, 718, 718, 718, 718, 718, ...
## Resampling results across tuning parameters:
##
##   alpha  lambda       RMSE       Rsquared
##   0.10   0.001202348  0.8127568  0.3411090
##   0.10   0.012023480  0.8183111  0.3344713
##   0.10   0.120234797  0.8077544  0.3396316
##   0.55   0.001202348  0.8119513  0.3412175
##   0.55   0.012023480  0.8227484  0.3268770
##   0.55   0.120234797  0.8117291  0.3473930
##   1.00   0.001202348  0.8116078  0.3407650
##   1.00   0.012023480  0.8189354  0.3309040
##   1.00   0.120234797  0.8167381  0.3445179
##
## RMSE was used to select the optimal model using  the smallest value.
## The final values used for the model were alpha = 0.1 and lambda
##  = 0.1202348.
```

6.8 Binary Logistic Regression (BLR)

Binary Logistic Regression (BLR) can models fault-proneness as follows

$$fp(X) = \frac{e^{logit()}}{1 + e^{logit(X)}}$$

where the simplest form for logit is:

$logit(X) = c_0 + c_1 X$

```r
jdt <- read.csv("./datasets/defectPred/BPD/single-version-ck-oo-EclipseJDTCore.csv", sep=";")

# Caret
library(caret)

# Convert the response variable into a boolean variable (0/1)
jdt$bugs[jdt$bugs>=1]<-1

cbo <- jdt$cbo
bugs <- jdt$bugs

# Split data into training and test datasets
jdt2 = data.frame(cbo, bugs)
inTrain <- createDataPartition(y=jdt2$bugs,p=.8,list=FALSE)
```

```
jdtTrain <- jdt2[inTrain,]
jdtTest <- jdt2[-inTrain,]
```

BLR models fault-proneness are as follows $fp(X) = \frac{e^{logit()}}{1+e^{logit(X)}}$

where the simplest form for logit is $logit(X) = c_0 + c_1X$

```
# logit regression
# glmLogit <- train (bugs ~ ., data=jdt.train, method="glm", family=binomial(link = logit))

glmLogit <- glm (bugs ~ ., data=jdtTrain, family=binomial(link = logit))
summary(glmLogit)
```

```
##
## Call:
## glm(formula = bugs ~ ., family = binomial(link = logit), data = jdtTrain)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.5734  -0.6125  -0.5378  -0.4968   2.0992
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.086378   0.134620 -15.498  < 2e-16 ***
## cbo          0.056462   0.007045   8.014 1.11e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 831.84  on 797  degrees of freedom
## Residual deviance: 725.93  on 796  degrees of freedom
## AIC: 729.93
##
## Number of Fisher Scoring iterations: 5
```

Predict a single point:

```
newData = data.frame(cbo = 3)
predict(glmLogit, newData, type = "response")
```

```
##         1
## 0.1281974
```

Draw the results, modified from: http://www.shizukalab.com/toolkits/plotting-logistic-regression-in-r

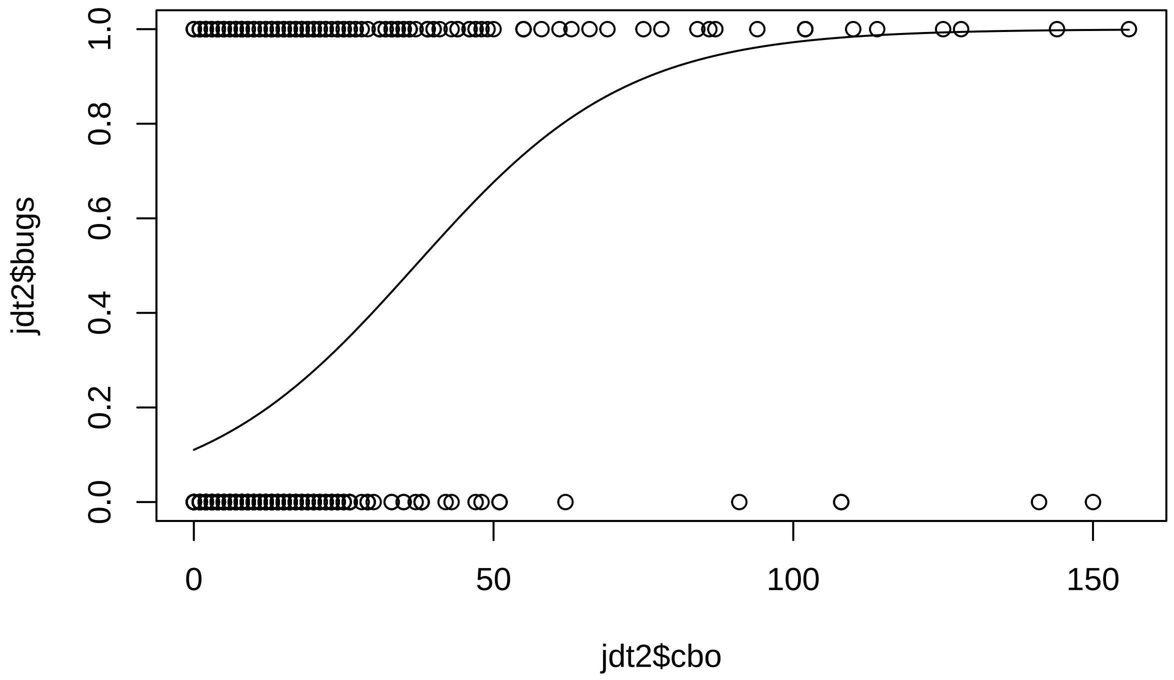
```
results <- predict(glmLogit, jdtTest, type = "response")

range(jdtTrain$cbo)
```

```
## [1]   0 156
```

```
range(results)
```

```
## [1] 0.1104278 0.9840854
```

```
plot(jdt2$cbo,jdt2$bugs)
curve(predict(glmLogit, data.frame(cbo=x), type = "response"),add=TRUE)
```



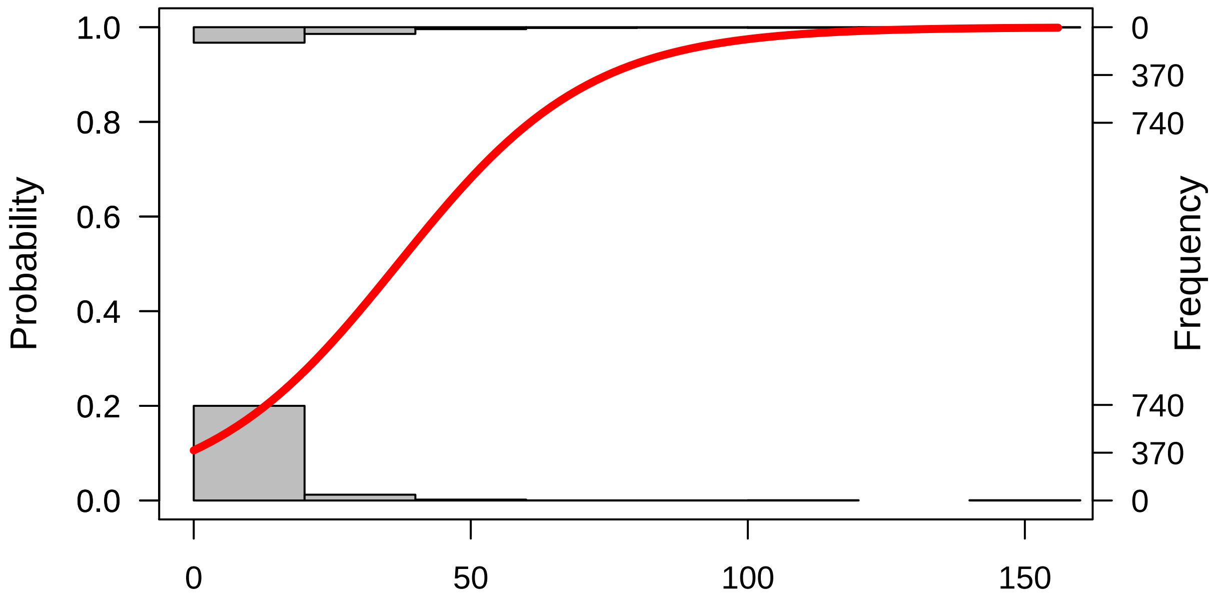
```r
# points(jdtTrain$cbo,fitted(glmLogit))
```

Another type of graph:

```r
library(popbio)
```

```
##
## Attaching package: 'popbio'

## The following object is masked from 'package:caret':
##
##     sensitivity
```

```r
logi.hist.plot(jdt2$cbo,jdt2$bugs,boxp=FALSE,type="hist",col="gray")
```



6.9 The caret package

There are hundreds of packages to perform classification task in R, but many of those can be used throught the 'caret' package which helps with many of the data mining process task as described next.

The caret packagehttp://topepo.github.io/caret/ provides a unified interface for modeling and prediction with around 150 different models with tools for:

```
+ data splitting

+ pre-processing

+ feature selection

+ model tuning using resampling

+ variable importance estimation, etc.
```

Website: http://caret.r-forge.r-project.org

JSS Paper: www.jstatsoft.org/v28/i05/paper

Book: Applied Predictive Modeling

6.10 Ensembles

Ensembles or meta-learners combine multiple models to obtain better predictions. They are typically classified as Bagging, Boosting and Stacking (Stacked generalization).

- Bagging (Breiman, 1996) (also known as Bootstrap aggregating) is an ensemble technique in which a base learner is applied to multiple equal size datasets created from the original data using bootstraping. Predictions are based on voting of the individual predictions. An advantage of bagging is that it does not require any modification to the learning algorithm and takes advantage of the instability of the base classifier to create diversity among individual ensembles so that individual members of the ensemble perform well in different regions of the data. Bagging does *not* perform well with classifiers if their output is robust to perturbation of the data such as nearest-neighbour (NN) classifiers.
- Boosting techniques generate multiple models that complement each other inducing models that improve regions of the data where previous induced models preformed poorly. This is achieved by increasing the weights of instances wrongly classified, so new learners focus on those instances. Finally, classification is based on a weighted voted among all members of the ensemble. In particular, AdaBoost (Adaptive Boosting) is a popular boosting algorithm for classification (Freund et al., 1999). The set of training examples is assigned an equal weight at the beginning and the weight of instances is either increased or decreased depending on whether the learner classified that instance incorrectly or not. The following iterations focus on those instances with higher weights. AdaBoost can be applied to any base learner.
- *Stacking* (Stacked generalisation) which combines different types of models

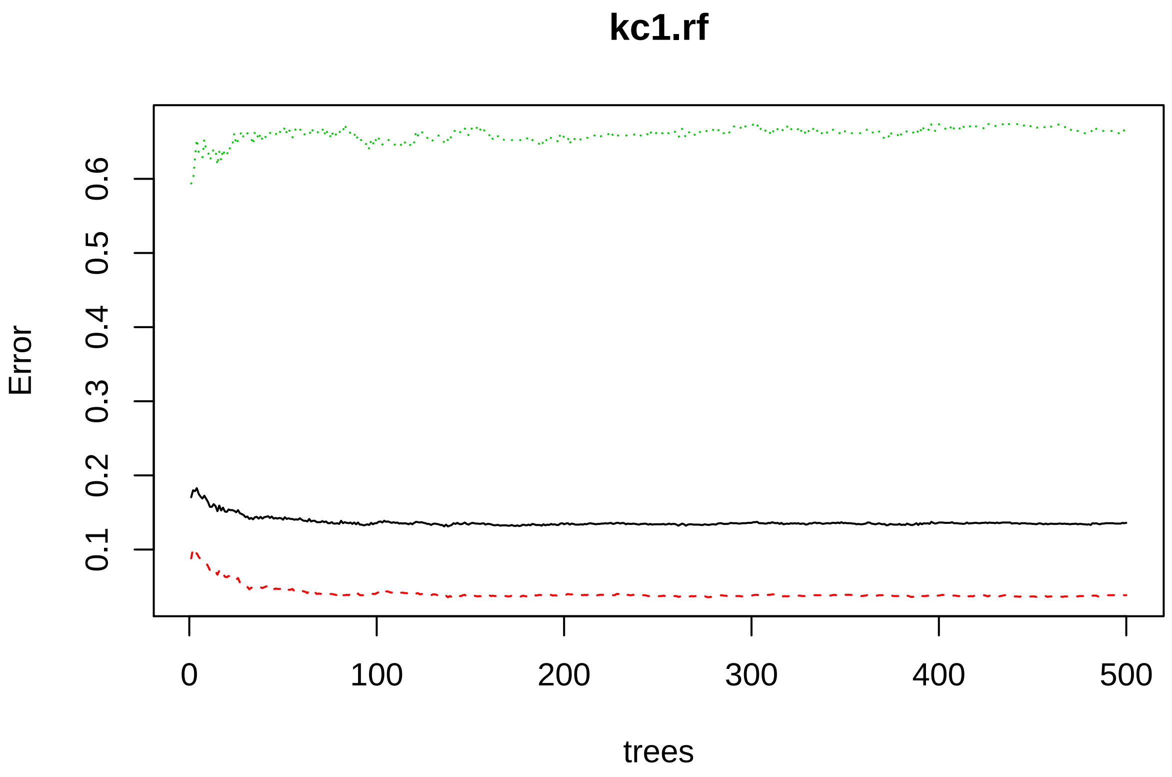
An very propular ensemble is Rotation Forests [40] combine randomly chosen subsets of attributes (random subspaces) and bagging approaches with principal components feature generation to construct an ensemble of decision trees. Principal Component Analysis is used as a feature selection technique combining subsets of attributes which are used with a bootstrapped subset of the training data by the base classifier.

```r
# Load library
library(randomForest)
library(foreign) # To load arff file
```

```
#library(party) # Build a decision tree
#library(caret)

kc1 <- read.arff("./datasets/defectPred/D1/KC1.arff")

kc1.rf <- randomForest(kc1$Defective ~ . ,data = kc1, na.action=na.omit)
print(kc1.rf)
```

```
##
## Call:
##  randomForest(formula = kc1$Defective ~ ., data = kc1, na.action = na.omit)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 13.6%
## Confusion matrix:
##      N   Y class.error
## N 1703  68  0.03839639
## Y  217 108  0.66769231
```

```
plot(kc1.rf)
```



A problem with ensembles is that their models are difficult to interpret (they behave as blackboxes) in comparison to decision trees or rules which provide an explanation of their decision making process.

Chapter 7

Regression tecniques in Software defect prediction

The purpose of this Section is to show the basic ways of estimating software defects by means of several regression models. We apply the techniques on the Equinox dataset. In what follows we use the same data points for training and testing, for the sake of clarity.

7.1 Zero Poison

When estimating the number of defects but we have many zeroes, we can use different types of zero Poison regression.

7.1.1 Load packages

```
## Loading required package: pscl
```

```
## Classes and Methods for R developed in the
```

```
## Political Science Computational Laboratory
```

```
## Department of Political Science
```

```
## Stanford University
```

```
## Simon Jackman
```

```
## hurdle and zeroinfl functions by Achim Zeileis
```

```
## Loading required package: boot
```

```
##
## Attaching package: 'boot'
```

```
## The following object is masked from 'package:lattice':
##
##     melanoma
```

```
##
## Attaching package: 'corrplot'
## The following object is masked from 'package:pls':
##
##     corrplot
```

7.1.2 The number of Software Defects. Count Data

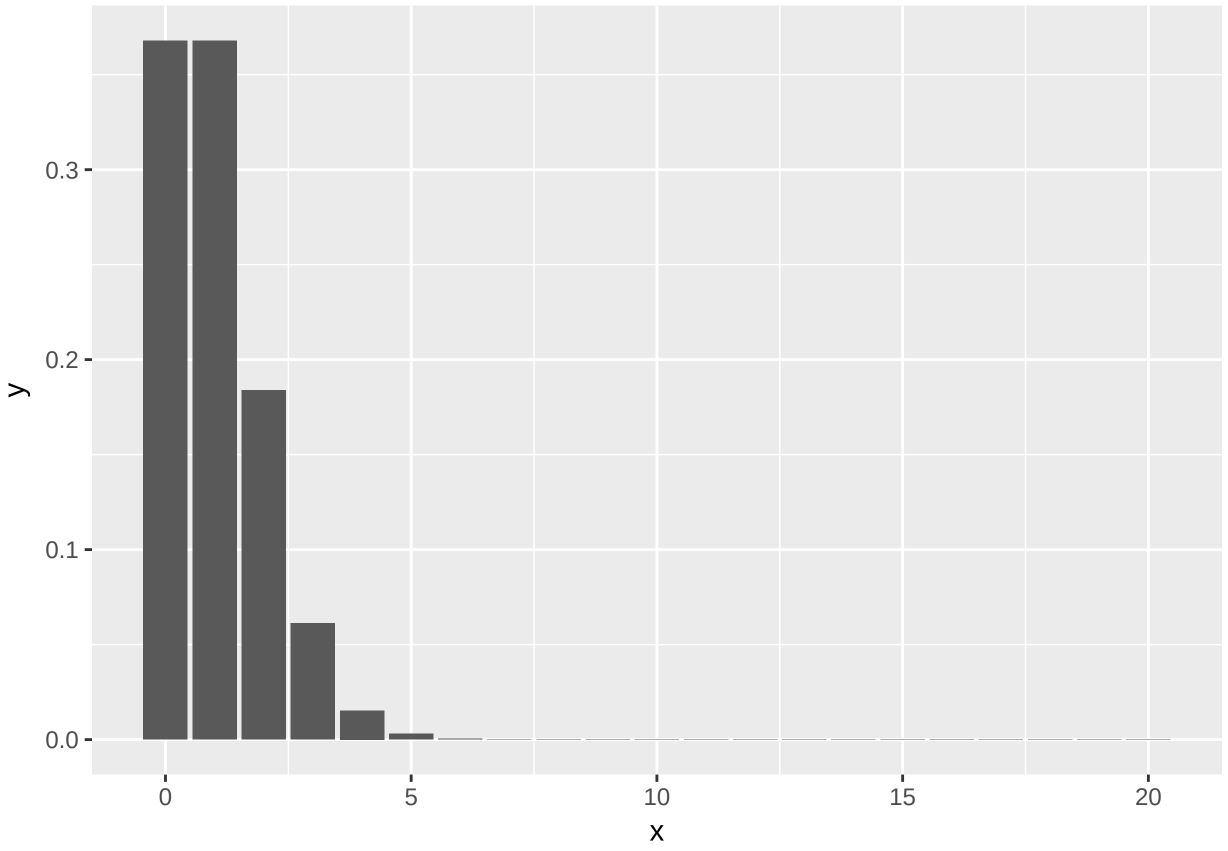
The number of software defects found in a software product can be assimilated to the “count data” concept that is used in many disciplines, because the outcome,number of defects, of whatever software process is a count. There are several ways of analyzing count data. The classical Poisson, negative binomial regression model can be augmented with zero-inflated poisson and zero-inflated negative binomial models to cope with the excess of zeros in the count data. Zero-inflated means that the response variable -software defects- contains more zeros than expected, based on the Poisson or negative binomial distribution. A simple histogram may show the trend. Count variables appear in different areas and have common properties: their range is a non-negative integer(0, 1, ….); and their values are skewed towards zero, because few values are high.

7.1.3 Normal Regression

The normal regression provided by the general linear model (glm in R) is not appropriate for count data because of the non-normality of the residuals. Usually the variance of the residuals increase with their value.

7.1.4 Poisson Regression

This model can be used if there is no excess in the number of zeros. The poisson model is the most common and it is a probability distribution that is controlled by a single parameter, λ



7.1.5 Negative binomial

The negative binomial takes into account overdispersion in the count outcome. When there are more high values or zeros than expected the mean and the variance of the sample data are different.

7.1.6 Zero-Inflated Poisson Regression ZIP

This is a model that can deal with the excess of zeros by explicitly modeling the part that generates the false zeros. There may be different sources for the excess of zeros. The count process is modeled by a Poisson model.

7.1.7 Zero-Inflated Negative Binomial ZINB

In this case, besides taking into account the excess of zeros the count process is modeled by a Negative Binomial model that allows for overdispersion from the non-zero counts.

7.1.8 Read Data

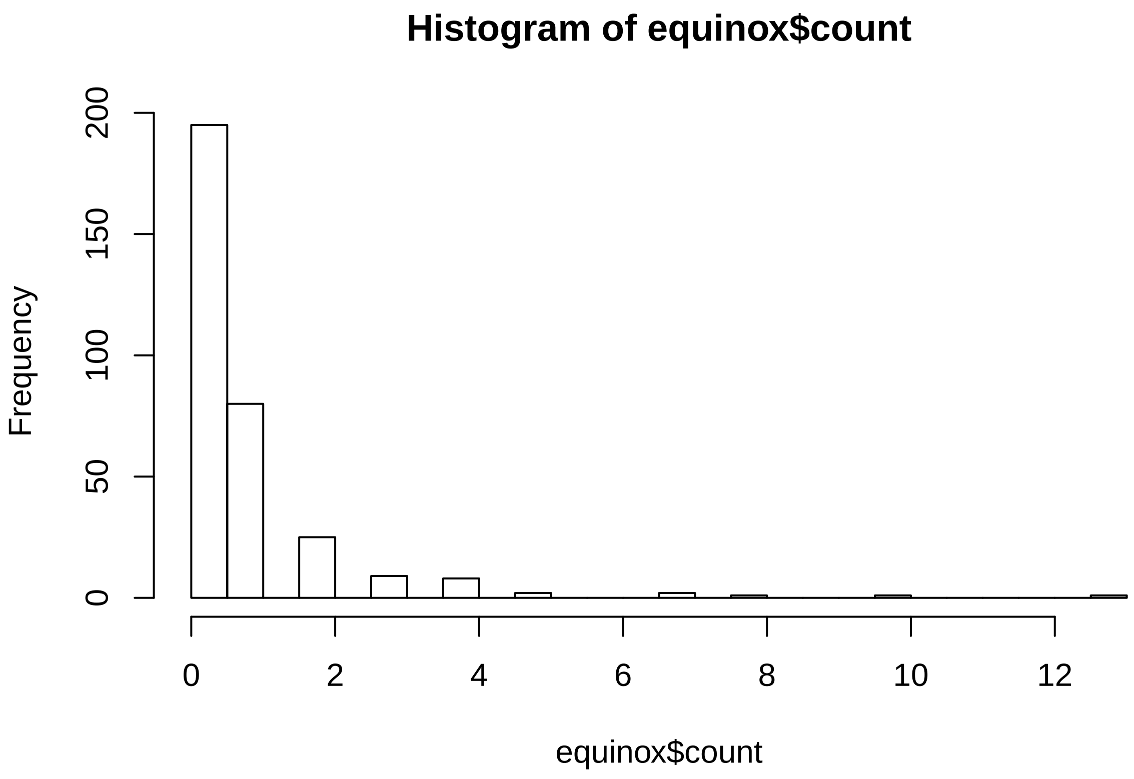
```r
# setwd("~/DocProjects/zeropoisson")
auxfile <- read.table("./datasets/defectPred/BPD/single-version-ck-oo.csv", header = TRUE, sep = ";")
myvars <- c('bugs', 'wmc', 'rfc', 'cbo', 'lcom', 'numberOfLinesOfCode')
```

```
# wmd   weighted methods per class Despite its long name, WMC is simply the method count for a class.
# rfc   response for a class set of methods that can potentially be executed in response to a message r
# cbo   coupling between objects number of classes to which a class is coupled
# lcom  lack of cohesion

equinox <- auxfile[myvars]
names(equinox)[names(equinox) == 'bugs'] <- 'count'
```

7.1.9 Plot histogram

- The first histogram shows the high number of modules with no defects.
- The second histogram shows the distribution of the non-zero values.

```
hist(equinox$count,breaks="FD")
```

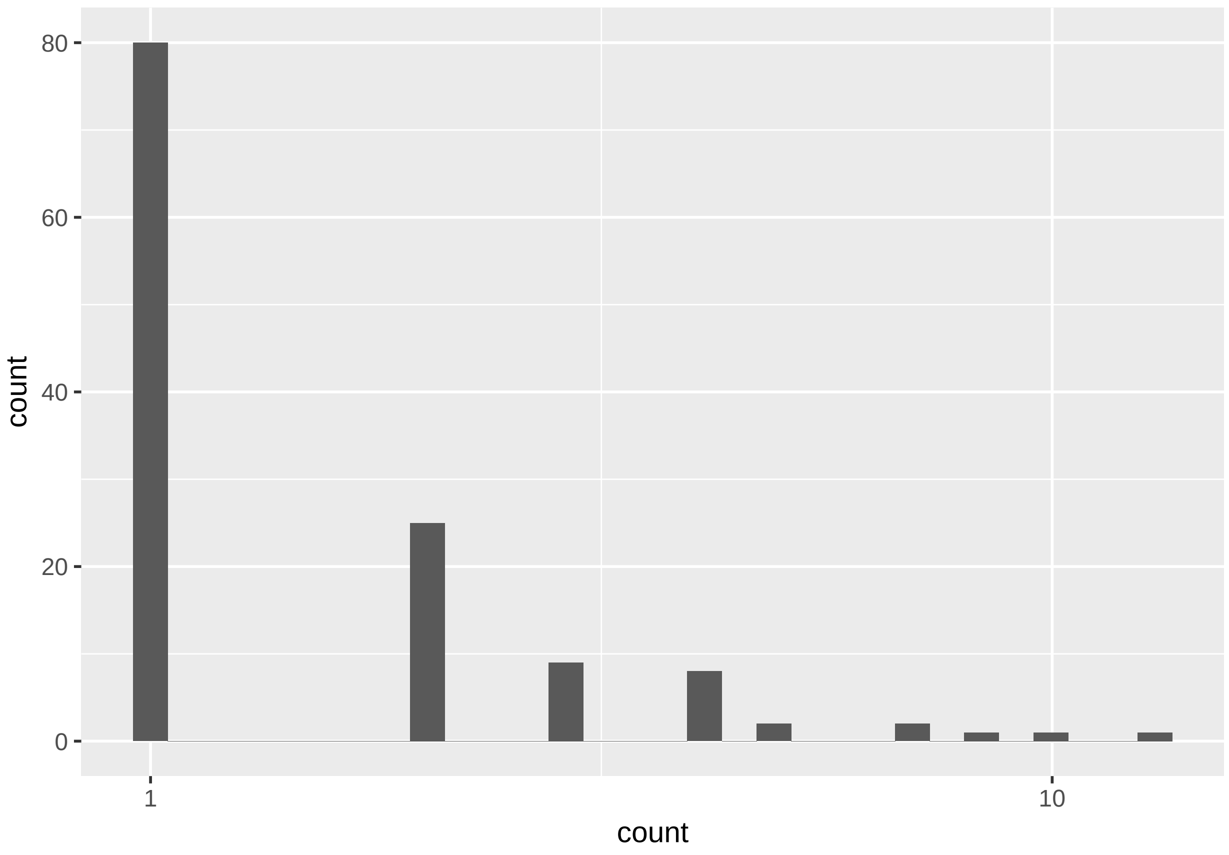


```
## histogram with x axis in log10 scale
ggplot(equinox, aes(count)) +
  geom_histogram() +
  scale_x_log10()
```

```
## Warning: Transformation introduced infinite values in continuous x-axis
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

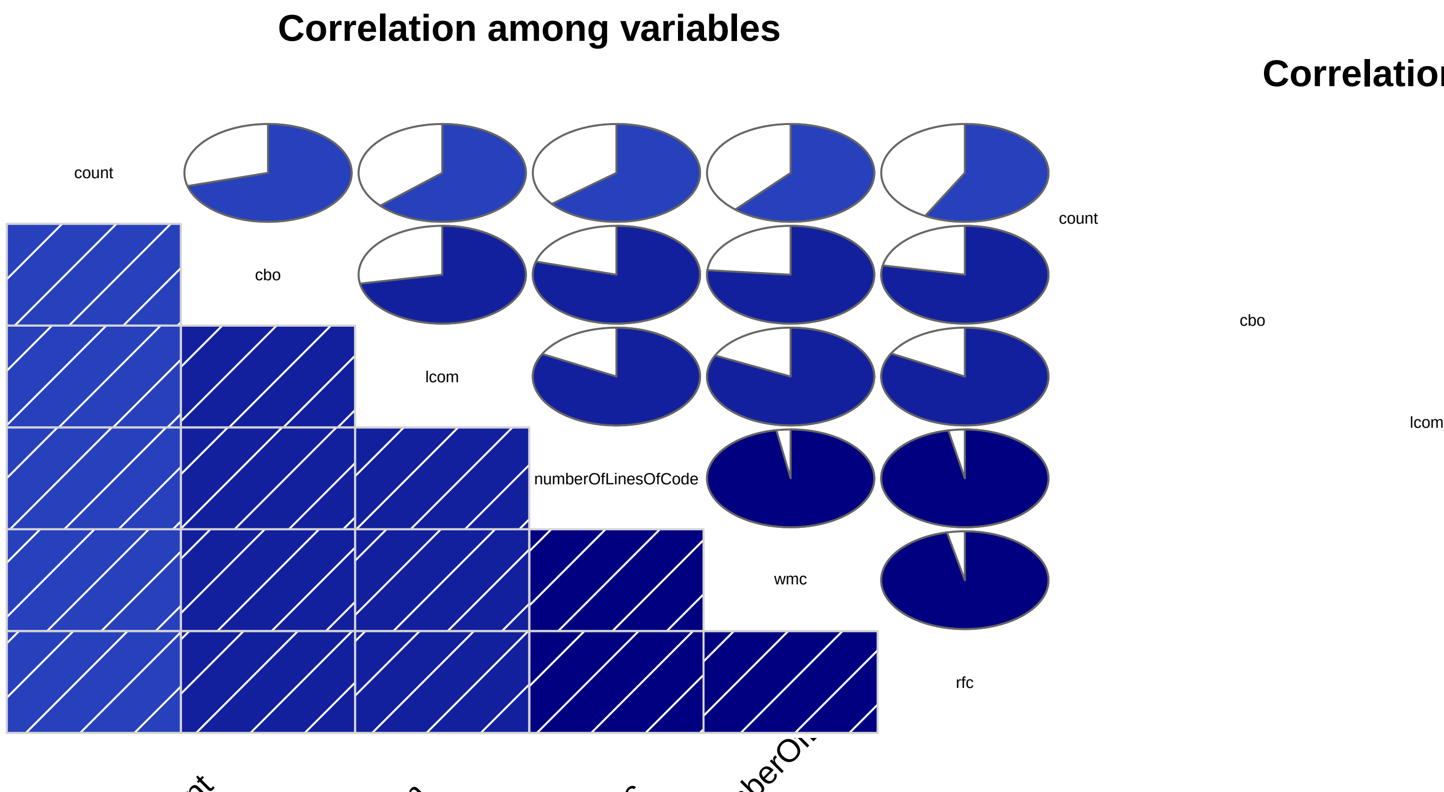
```
## Warning: Removed 195 rows containing non-finite values (stat_bin).
```

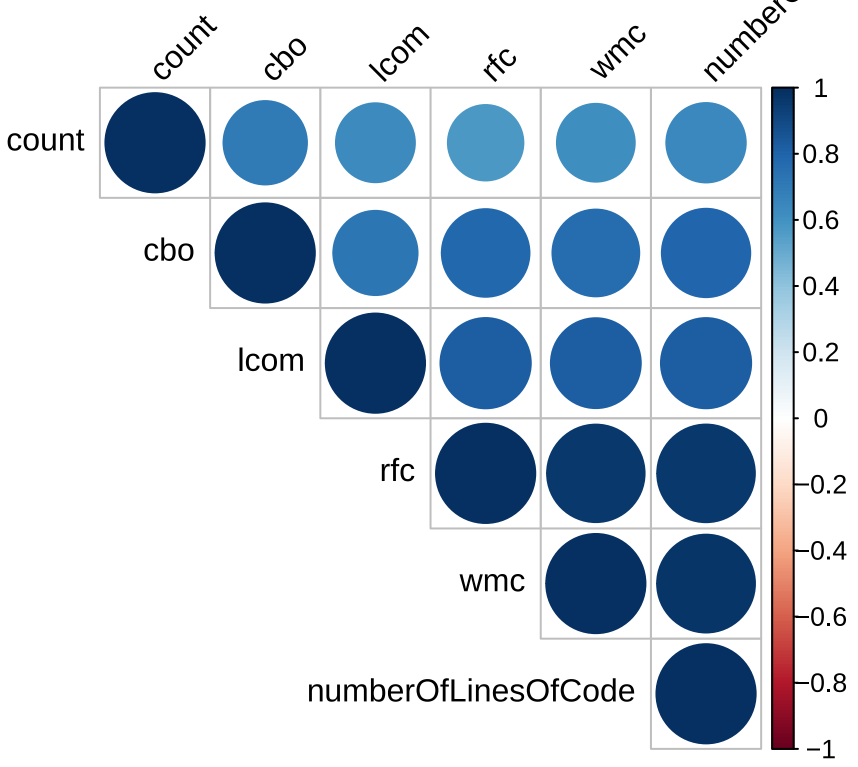


7.1.10 Correlation among variables

The correlation plots and the tables show a strong correlation among some variables.

```
##                         count       wmc       rfc       cbo      lcom
## count               1.0000000 0.6151012 0.5786541 0.7065440 0.6327475
## wmc                 0.6151012 1.0000000 0.9656904 0.7647778 0.8208613
## rfc                 0.5786541 0.9656904 1.0000000 0.7804111 0.8283116
## cbo                 0.7065440 0.7647778 0.7804111 1.0000000 0.7208646
## lcom                0.6327475 0.8208613 0.8283116 0.7208646 1.0000000
## numberOfLinesOfCode 0.6404813 0.9725794 0.9683453 0.7932072 0.8263641
##                     numberOfLinesOfCode
## count                         0.6404813
## wmc                           0.9725794
## rfc                           0.9683453
## cbo                           0.7932072
## lcom                          0.8263641
## numberOfLinesOfCode           1.0000000
```





```
##      count             wmc              rfc              cbo
##  Min.   : 0.0000   Min.   :  0.00   Min.   :   0.00   Min.   : 0.000
##  1st Qu.: 0.0000   1st Qu.:  2.00   1st Qu.:   5.00   1st Qu.: 2.000
##  Median : 0.0000   Median : 12.00   Median :  18.00   Median : 6.000
##  Mean   : 0.7531   Mean   : 32.64   Mean   :  58.34   Mean   : 9.673
##  3rd Qu.: 1.0000   3rd Qu.: 36.00   3rd Qu.:  61.50   3rd Qu.:13.250
##  Max.   :13.0000   Max.   :534.00   Max.   :1009.00   Max.   :77.000
##       lcom        numberOfLinesOfCode
##  Min.   :   0.0   Min.   :   0.00
##  1st Qu.:   1.0   1st Qu.:   7.75
##  Median :  15.0   Median :  45.00
##  Mean   : 124.2   Mean   : 122.02
##  3rd Qu.:  66.0    3rd Qu.: 130.00
##  Max.   :2775.0   Max.   :1805.00
```

7.1.11 Classical Regression

- First we fit a classical regression model using the variables `cbo`, `lcom`, `wmc`, `rfc`. Please note that the last section that compares models with different combinations of variables.
- The parameters of interest are the intercept and the slope coefficients

```
##
## Call:
## glm(formula = count ~ cbo + lcom + wmc + rfc, family = gaussian,
##     data = equinox)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.5540  -0.4498  -0.0248   0.3129   6.2884
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.0816095  0.0749013  -1.090    0.277
## cbo          0.0750713  0.0078977   9.505  < 2e-16 ***
## lcom         0.0012674  0.0002786   4.549 7.68e-06 ***
## wmc          0.0154600  0.0034539   4.476 1.06e-05 ***
## rfc         -0.0094888  0.0018973  -5.001 9.42e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.9355133)
##
##     Null deviance: 686.25  on 323  degrees of freedom
## Residual deviance: 298.43  on 319  degrees of freedom
## AIC: 904.84
##
## Number of Fisher Scoring iterations: 2
```

7.2 Poisson Regression

- Regression coefficients must be interpreted with the log transformation.

```
##
## Call:
## glm(formula = count ~ cbo + lcom + wmc + rfc, family = poisson,
##     data = equinox)
##
## Deviance Residuals:
```

```
##     Min       1Q   Median       3Q      Max
## -3.0940  -0.8936  -0.7937   0.5785   2.7547
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.1797490  0.1011751 -11.660  < 2e-16 ***
## cbo          0.0518008  0.0056195   9.218  < 2e-16 ***
## lcom         0.0001574  0.0001744   0.903    0.367
## wmc          0.0114261  0.0018959   6.027 1.67e-09 ***
## rfc         -0.0060023  0.0011625  -5.163 2.43e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 588.43  on 323  degrees of freedom
## Residual deviance: 316.67  on 319  degrees of freedom
## AIC: 632.15
##
## Number of Fisher Scoring iterations: 5
```

7.3 Compare to Null Model (intercept)

- The chi-squared test on the difference of log likelihoods is used to compare the poisson model to the intercept. It yields a high significant p-value; thus, the model is statistically significant.
- All of the predictors in both the count and inflation portions of the model are statistically significant. This model fits the data significantly better than the null model, i.e., the intercept-only model.
- Since we have five predictor variables in the full model, the degrees of freedom for chi-squared test is 5.

```
# five variables in the full model --> df = 5
nullmodel <- update(defects_pois, . ~ 1)

pchisq(2 * (logLik(defects_pois) - logLik(nullmodel)), df = 5, lower.tail=FALSE)
```

```
## 'log Lik.' 1.171544e-56 (df=5)
```

7.4 Negative Binomial

```
# negative binomial regression
defects_negbinom <- glm.nb(count ~ cbo + lcom + wmc + rfc,
data = equinox)
```

```
## Warning in glm.nb(count ~ cbo + lcom + wmc + rfc, data = equinox):
## alternation limit reached
```

```
# summary of results
summary(defects_negbinom)
```

```
##
## Call:
## glm.nb(formula = count ~ cbo + lcom + wmc + rfc, data = equinox,
##     init.theta = 6.219388589, link = log)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4508  -0.8535  -0.7505   0.5116   2.5068
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.2780140  0.1140133 -11.209  < 2e-16 ***
## cbo          0.0583198  0.0071047   8.209 2.24e-16 ***
## lcom         0.0001011  0.0002150   0.470    0.638
## wmc          0.0118841  0.0024655   4.820 1.43e-06 ***
## rfc         -0.0061535  0.0014455  -4.257 2.07e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for Negative Binomial(6.2194) family taken to be 1)
##
##     Null deviance: 507.19  on 323  degrees of freedom
## Residual deviance: 278.33  on 319  degrees of freedom
## AIC: 628.55
##
## Number of Fisher Scoring iterations: 1
##
##
##               Theta:  6.22
##           Std. Err.:  3.14
## Warning while fitting theta: alternation limit reached
##
##  2 x log-likelihood:  -616.551
```

```
# overdispersion
summary(defects_negbinom)$theta
```

```
## [1] 6.219389
```

7.4.1 Zero Inflated Poisson

```
# zero-inflated Poisson regression
# the | seperates the count model from the logistic model
defects_zip <- zeroinfl(count ~ cbo + lcom + wmc + rfc | numberOfLinesOfCode, data =
equinox, link = "logit", dist = "poisson", trace = FALSE)
# summary of results
summary(defects_zip)
```

```
##
## Call:
## zeroinfl(formula = count ~ cbo + lcom + wmc + rfc | numberOfLinesOfCode,
##     data = equinox, dist = "poisson", link = "logit", trace = FALSE)
##
## Pearson residuals:
##     Min      1Q  Median      3Q     Max
## -2.3683 -0.6880 -0.2784  0.4692  3.8402
##
```

```
## Count model coefficients (poisson with log link):
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.8641763  0.1240264  -6.968 3.22e-12 ***
## cbo          0.0433784  0.0055946   7.754 8.93e-15 ***
## lcom         0.0002456  0.0001123   2.188   0.0287 *
## wmc          0.0097291  0.0019739   4.929 8.27e-07 ***
## rfc         -0.0051994  0.0010795  -4.816 1.46e-06 ***
##
## Zero-inflation model coefficients (binomial with logit link):
##                     Estimate Std. Error z value Pr(>|z|)
## (Intercept)           1.6749     0.6869   2.438   0.0147 *
## numberOfLinesOfCode  -0.1657     0.1118  -1.483   0.1380
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Number of iterations in BFGS optimization: 28
## Log-likelihood: -296.5 on 7 Df
```

7.4.2 Zero Inflated Negative Binomial

```
# zero-inflated negative binomial regression
# the | seperates the count model from the logistic model
defects_zinb <- zeroinfl(count ~ cbo + lcom + wmc + rfc | numberOfLinesOfCode, data =
equinox, link = "logit", dist = "negbin", trace = FALSE, EM = FALSE)
```

```
## Warning in sqrt(diag(vc)[np]): NaNs produced
```

```
# summary of results
summary(defects_zinb)
```

```
## Warning in sqrt(diag(object$vcov)): NaNs produced
```

```
##
## Call:
## zeroinfl(formula = count ~ cbo + lcom + wmc + rfc | numberOfLinesOfCode,
##     data = equinox, dist = "negbin", link = "logit", trace = FALSE,
##     EM = FALSE)
##
## Pearson residuals:
##     Min      1Q  Median      3Q     Max
## -1.8809 -0.6721 -0.2745  0.4487  3.7922
##
## Count model coefficients (negbin with log link):
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.9088131  0.1282227  -7.088 1.36e-12 ***
## cbo          0.0450277  0.0059527   7.564 3.90e-14 ***
## lcom         0.0002357         NA      NA       NA
## wmc          0.0100136  0.0022232   4.504 6.67e-06 ***
## rfc         -0.0052856  0.0013063  -4.046 5.20e-05 ***
## Log(theta)   2.6591002         NA      NA       NA
##
## Zero-inflation model coefficients (binomial with logit link):
##                     Estimate Std. Error z value Pr(>|z|)
## (Intercept)           1.7143     0.7618   2.250   0.0244 *
## numberOfLinesOfCode  -0.1894     0.1332  -1.422   0.1551
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Theta = 14.2834
## Number of iterations in BFGS optimization: 42
## Log-likelihood: -295.8 on 8 Df
```

7.4.3 Comparing models with Vuong test

- The Vuong test compares the zero-inflated model with an ordinary Poisson regression model.
- The result of the test statistic is significant, indicating that the zero-inflated model is superior to the standard Poisson model.

```
vuong(defects_zip, defects_pois)
```

```
## Vuong Non-Nested Hypothesis Test-Statistic:
## (test-statistic is asymptotically distributed N(0,1) under the
##  null that the models are indistinguishible)
## -------------------------------------------------------------
##               Vuong z-statistic             H_A    p-value
## Raw                    3.143494 model1 > model2 0.00083472
## AIC-corrected          2.713456 model1 > model2 0.00332927
## BIC-corrected          1.900524 model1 > model2 0.02868219
```

```
vuong(defects_zip, defects_negbinom)
```

```
## Vuong Non-Nested Hypothesis Test-Statistic:
## (test-statistic is asymptotically distributed N(0,1) under the
##  null that the models are indistinguishible)
## -------------------------------------------------------------
##               Vuong z-statistic             H_A   p-value
## Raw                    2.468425 model1 > model2 0.0067854
## AIC-corrected          2.050671 model1 > model2 0.0201495
## BIC-corrected          1.260961 model1 > model2 0.1036615
```

7.4.4 Compute Confidence Intervals for intercept and variables. ZIP version

- We may confidence intervals for all the parameters and the exponentiated parameters using bootstrapping.
- We may compare these results with the regular confidence intervals based on the standard errors.

```
dput(round(coef(defects_zip, "count"), 4))
```

```
## structure(c(-0.8642, 0.0434, 2e-04, 0.0097, -0.0052), .Names = c("(Intercept)",
## "cbo", "lcom", "wmc", "rfc"))
```

```
dput(round(coef(defects_zip, "zero"), 4))
```

```
## structure(c(1.6749, -0.1657), .Names = c("(Intercept)", "numberOfLinesOfCode"
## ))
```

```
# change list of parameters 1, 3, 5 etc

### FINAL MODEL SELECTED ZERO INFLATED POISSON
```

```
f <- function(data, i) {
  require(pscl)
  m <- zeroinfl(count ~ cbo + lcom + wmc + rfc | numberOfLinesOfCode,
    data = data[i, ], dist = "poisson",
    start = list(count = c(-0.8642, 0.0434, 2e-04, 0.0097, -0.0052), zero = c(1.6749, -0.1657)))
  as.vector(t(do.call(rbind, coef(summary(m)))[, 1:2]))
}

set.seed(10)
(res <- boot(equinox, f, R = 1200, parallel = "snow", ncpus = 4))
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = equinox, statistic = f, R = 1200, parallel = "snow",
##     ncpus = 4)
##
##
## Bootstrap Statistics :
##           original        bias     std. error
## t1*  -0.8641784331 -2.303129e-02 1.316306e-01
## t2*   0.1240279149  1.058718e-02 3.483776e-02
## t3*   0.0433783993  5.727146e-04 7.267716e-03
## t4*   0.0055945857  1.619122e-03 2.941890e-03
## t5*   0.0002456421 -1.532408e-05 2.769170e-04
## t6*   0.0001122920  6.739868e-05 1.380387e-04
## t7*   0.0097291337  4.351203e-04 2.957174e-03
## t8*   0.0019739001  5.735321e-04 1.386695e-03
## t9*  -0.0051993761 -1.748856e-04 1.561095e-03
## t10*  0.0010795051  3.518046e-04 5.972876e-04
## t11*  1.6750251424  6.975388e-01 5.261811e+00
## t12*  0.6869578264  9.237353e-01 2.058666e+01
## t13* -0.1657573309 -9.363010e-02 5.468317e-01
## t14*  0.1117748277  6.984320e-02 2.075220e+00
```

```
## basic parameter estimates with percentile and bias adjusted CIs
parms <- t(sapply(c(1, 3, 5, 7, 9, 11, 13), function(i) {
  out <- boot.ci(res, index = c(i, i + 1), type = c("perc", "bca"))
  with(out, c(Est = t0, pLL = percent[4], pUL = percent[5],
    bcaLL = bca[4], bcaUL = bca[5]))
}))

print("Confidence intervals for ZIP, BOOTED")
```

```
## [1] "Confidence intervals for ZIP, BOOTED"
```

```
## add row names
names(coef(defects_zip))
```

```
## [1] "count_(Intercept)"        "count_cbo"
## [3] "count_lcom"               "count_wmc"
## [5] "count_rfc"                "zero_(Intercept)"
## [7] "zero_numberOfLinesOfCode"
```

```
nrow(parms)
```

```
## [1] 7
```

```
row.names(parms) <- names(coef(defects_zip))
## print results
parms
```

```
##                                    Est          pLL           pUL
## count_(Intercept)        -0.8641784331 -1.155311776 -0.6405252226
## count_cbo                 0.0433783993  0.030104286  0.0594077144
## count_lcom                0.0002456421 -0.000363527  0.0007406458
## count_wmc                 0.0097291337  0.005179901  0.0187585385
## count_rfc                -0.0051993761 -0.009329260 -0.0028504289
## zero_(Intercept)          1.6750251424  0.712867236  5.0130270331
## zero_numberOfLinesOfCode -0.1657573309 -0.594861582 -0.0682408298
##                                 bcaLL        bcaUL
## count_(Intercept)        -1.105649479 -0.565552024
## count_cbo                 0.026293665  0.058188886
## count_lcom               -0.000324895  0.000781505
## count_wmc                 0.005126268  0.018329267
## count_rfc                -0.009010388 -0.002658944
## zero_(Intercept)          0.377152297  3.928757887
## zero_numberOfLinesOfCode -0.547781601 -0.058402161
```

```
print("Confidence intervals for ZIP , NORMAL BASED APPROXIMATION")
```

```
## [1] "Confidence intervals for ZIP , NORMAL BASED APPROXIMATION"
```

```
## compare with normal based approximation
confint(defects_zip)
```

```
##                                  2.5 %        97.5 %
## count_(Intercept)        -1.107264e+00 -0.6210889471
## count_cbo                 3.241321e-02  0.0543435607
## count_lcom                2.555382e-05  0.0004657304
## count_wmc                 5.860348e-03  0.0135978860
## count_rfc                -7.315162e-03 -0.0030835806
## zero_(Intercept)          3.286748e-01  3.0212006348
## zero_numberOfLinesOfCode -3.847732e-01  0.0532838577
```

- Estimate the incident risk ratio (IRR) for the Poisson model and odds ratio (OR) for the logistic (zero inflation) model.

```
## exponentiated parameter estimates with percentile and bias adjusted CIs
expparms <- t(sapply(c(1, 3, 5, 7, 9, 11, 13), function(i) {
  out <- boot.ci(res, index = c(i, i + 1), type = c("perc", "bca"), h = exp)
  with(out, c(Est = t0, pLL = percent[4], pUL = percent[5],
    bcaLL = bca[4], bcaUL = bca[5]))
}))

## add row names
row.names(expparms) <- names(coef(defects_zip))
## print results
expparms
```

```
##                                 Est        pLL          pUL      bcaLL
## count_(Intercept)        0.4213976 0.3149594   0.5270156 0.3309958
## count_cbo                1.0443330 1.0305620   1.0612078 1.0266424
## count_lcom               1.0002457 0.9996365   1.0007409 0.9996752
## count_wmc                1.0097766 1.0051933   1.0189356 1.0051394
## count_rfc                0.9948141 0.9907141   0.9971536 0.9910301
## zero_(Intercept)         5.3389294 2.0398324 150.3599485 1.4581264
## zero_numberOfLinesOfCode 0.8472518 0.5516390   0.9340355 0.5782311
##                              bcaUL
## count_(Intercept)         0.5680465
## count_cbo                 1.0599152
## count_lcom                1.0007818
## count_wmc                 1.0184983
## count_rfc                 0.9973446
## zero_(Intercept)         50.8437847
## zero_numberOfLinesOfCode  0.9432705
```

7.5 Compare Models Fit. AIC and BIC

- There are several measures that can be used to assess model fit. Here we use information-based measures.
- The traditional Akaike's information criterion (AIC) selects the model that has the smallest AIC value.
- The Schwartz's Bayesian Information criterion (BIC) also minimizes model complexity.

```
#Model Fit
# AIC values
AIC(defects_normal)
```

```
## [1] 904.8354
```

```
AIC(defects_pois)
```

```
## [1] 632.1547
```

```
AIC(defects_negbinom)
```

```
## [1] 628.5507
```

```
AIC(defects_zip)
```

```
## [1] 606.9155
```

```
AIC(defects_zinb)
```

```
## [1] 607.5639
```

```
# AIC weights
compare_models <- list( )
compare_models[[1]] <- defects_normal
compare_models[[2]] <- defects_pois
compare_models[[3]] <- defects_negbinom
compare_models[[4]] <- defects_zip
compare_models[[5]] <- defects_zinb
compare_names <- c("Typical", "Poisson", "NB", "ZIP", "ZINB")
names(compare_models) <- compare_names
compare_results <- data.frame(models = compare_names)
compare_results$aic.val <- unlist(lapply(compare_models, AIC))
compare_results$aic.delta <- compare_results$aic.val-min(compare_results$aic.val)
```

```
compare_results$aic.likelihood <- exp(-0.5* compare_results$aic.delta)
compare_results$aic.weight <-
compare_results$aic.likelihood/sum(compare_results$aic.likelihood)

# BIC values
print("Schwarz's Bayesian criterion")
```

```
## [1] "Schwarz's Bayesian criterion"
```

```
AIC(defects_normal, k = log(nrow(equinox)))
```

```
## [1] 927.5198
```

```
AIC(defects_pois, k = log(nrow(equinox)))
```

```
## [1] 651.0584
```

```
AIC(defects_negbinom, k = log(nrow(equinox)))
```

```
## [1] 651.2352
```

```
AIC(defects_zip, k = log(nrow(equinox)))
```

```
## [1] 633.3807
```

```
AIC(defects_zinb, k = log(nrow(equinox)))
```

```
## [1] 637.8098
```

7.6 Compare prediction of Defects

```
# observed zero counts
# actual
sum(equinox$count < 1)
```

```
## [1] 195
```

```
# typical
sum(dnorm(0, fitted(defects_normal)))
```

```
## [1] 97.76806
```

```
# poisson
sum(dpois(0, fitted(defects_pois)))
```

```
## [1] 188.7356
```

```
# nb
sum(dnbinom(0, mu = fitted(defects_negbinom), size = defects_negbinom$theta))
```

```
## [1] 195.8165
```

```
# zip
sum(predict(defects_zip, type = "prob")[,1])
```

```
## [1] 195.7924
```

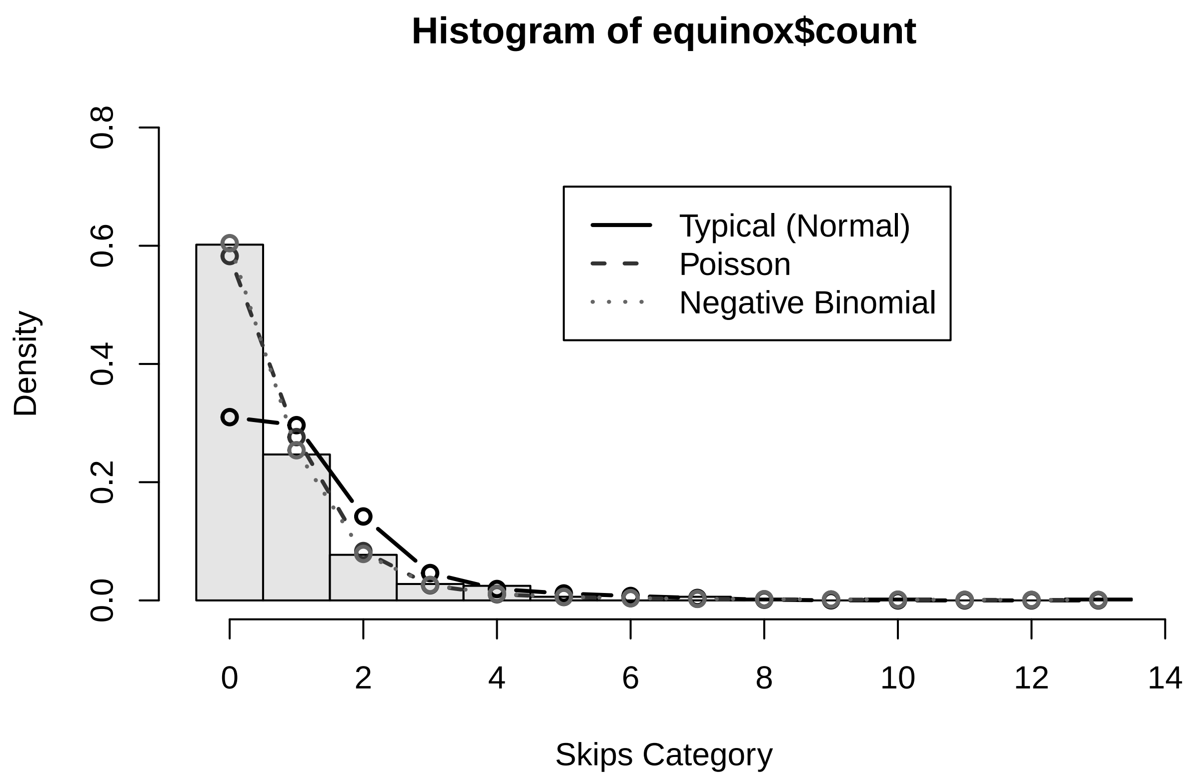
```
# zinb
sum(predict(defects_zinb, type = "prob")[,1])
```

```
## [1] 198.2048
```

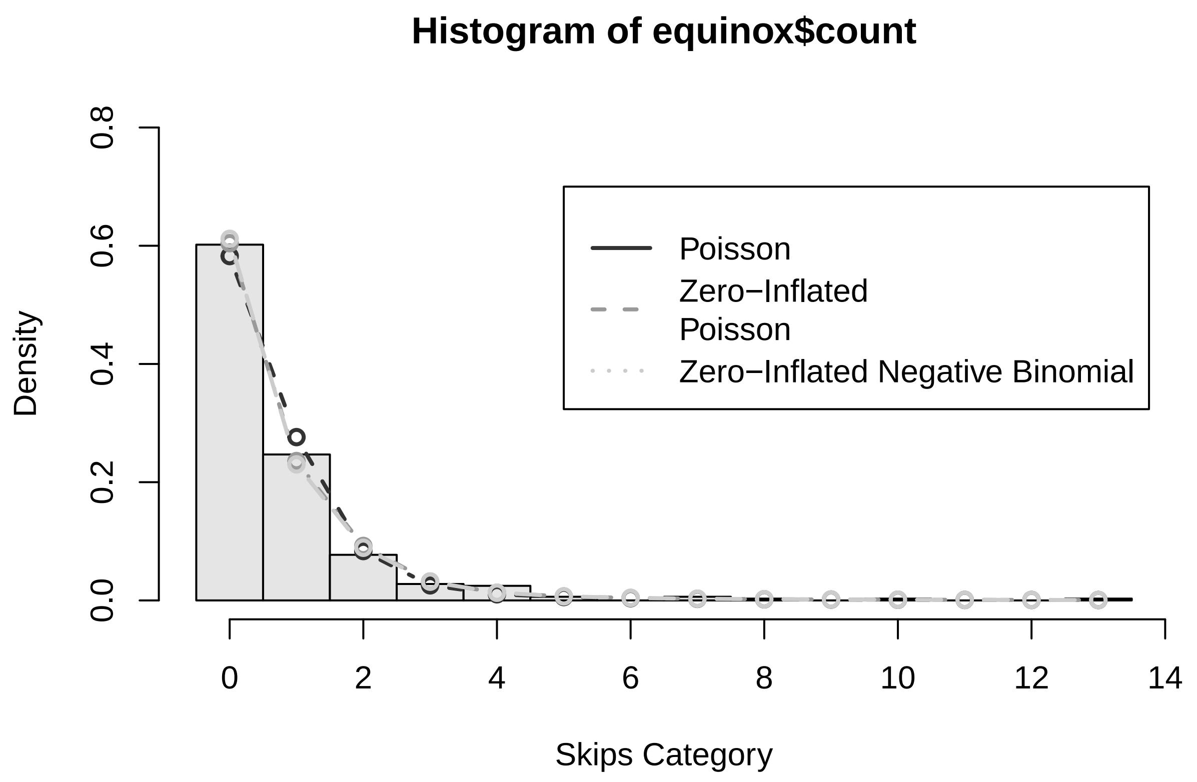
7.7 Plot predictions

```
# histogram plot with fitted probabilities
# predicted values for typical regression
normal.y.hat <- predict(defects_normal, type = "response")
normal.y <- defects_normal$y
normal.yUnique <- 0:max(normal.y)
normal.nUnique <- length(normal.yUnique)
phat.normal <- matrix(NA, length(normal.y.hat), normal.nUnique)
dimnames(phat.normal) <- list(NULL, normal.yUnique)
for (i in 1:normal.nUnique) {
phat.normal[, i] <- dnorm(mean = normal.y.hat, sd = sd(residuals(defects_normal)),x =
normal.yUnique[i])
}
# mean of the normal predicted probabilities for each value of the outcome
phat.normal.mn <- apply(phat.normal, 2, mean)
# probability of observing each value and mean predicted probabilities for
#count regression models
phat.pois <- predprob(defects_pois)
phat.pois.mn <- apply(phat.pois, 2, mean)
phat.nb <- predprob(defects_negbinom)
phat.nb.mn <- apply(phat.nb, 2, mean)
phat.zip <- predprob(defects_zip)
phat.zip.mn <- apply(phat.zip, 2, mean)
phat.zinb <- predprob(defects_zinb)
phat.zinb.mn <- apply(phat.zinb, 2, mean)
```

```
# histogram 1
hist(equinox$count, prob = TRUE, col = "gray90", breaks=seq(min(equinox$count)-0.5,
max(equinox$count)+.5, 1), xlab = "Skips Category", ylim=c(0,.8))
rangex <- length(phat.normal.mn)-1
# overlay predicted values
lines(x = seq(0, rangex, 1), y = phat.normal.mn, type = "b", lwd=2, lty=1, col="black")
lines(x = seq(0, rangex, 1), y = phat.pois.mn, type = "b", lwd=2, lty=2, col="gray20")
lines(x = seq(0, rangex, 1), y = phat.nb.mn, type = "b", lwd=2, lty=3, col="gray40")

# legend
legend(5, 0.7, c("Typical (Normal)","Poisson", "Negative Binomial"), lty=seq(1:3), col =
c("black","gray20","gray40"), lwd=2)
```



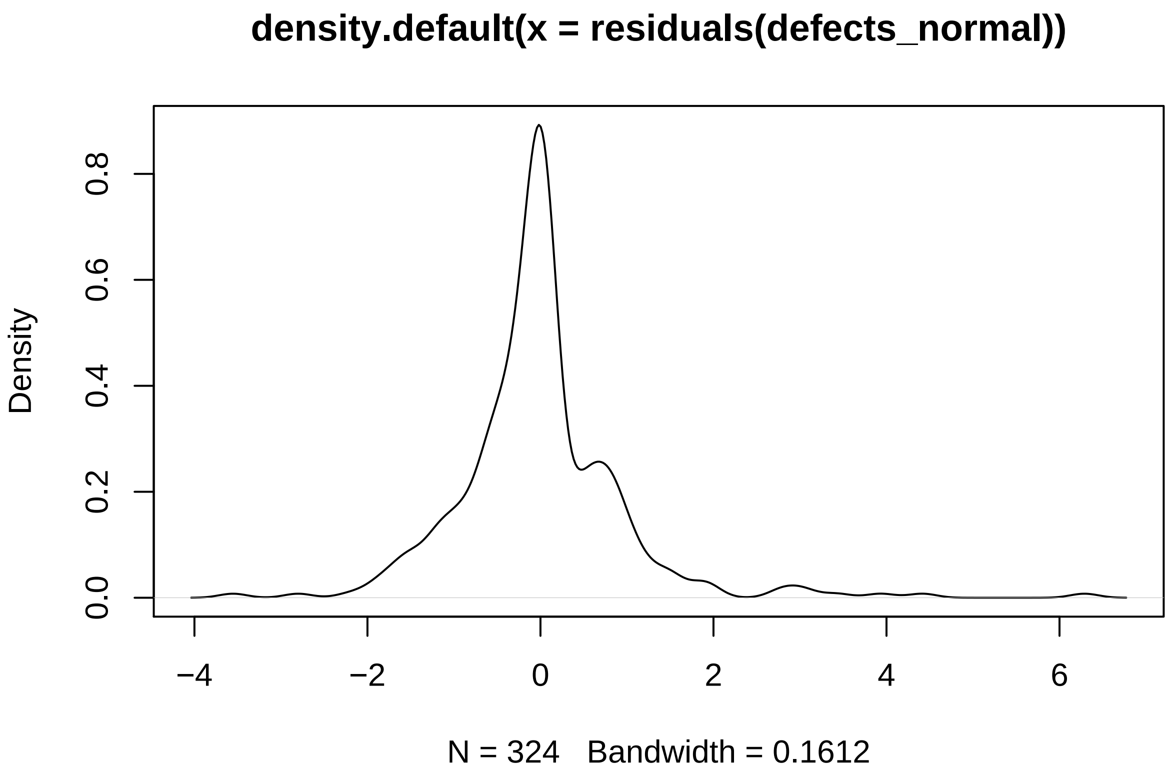
```r
# histogram 2
hist(equinox$count, prob = TRUE, col = "gray90", breaks=seq(min(equinox$count)-0.5,
max(equinox$count)+.5, 1), xlab = "Skips Category", ylim=c(0,.8))
rangex <- length(phat.normal.mn)-1
# overlay predicted values

lines(x = seq(0, rangex, 1), y = phat.pois.mn, type = "b", lwd=2, lty=2, col="gray20")
lines(x = seq(0, rangex, 1), y = phat.zip.mn, type = "b", lwd=2, lty=4, col="gray60")
lines(x = seq(0, rangex, 1), y = phat.zinb.mn, type = "b", lwd=2, lty=5, col="gray80")
# legend
legend(5, 0.7, c("Poisson",  "Zero-Inflated
Poisson", "Zero-Inflated Negative Binomial"), lty=seq(1:3), col =
c("gray20","gray60","gray80"), lwd=2)
```



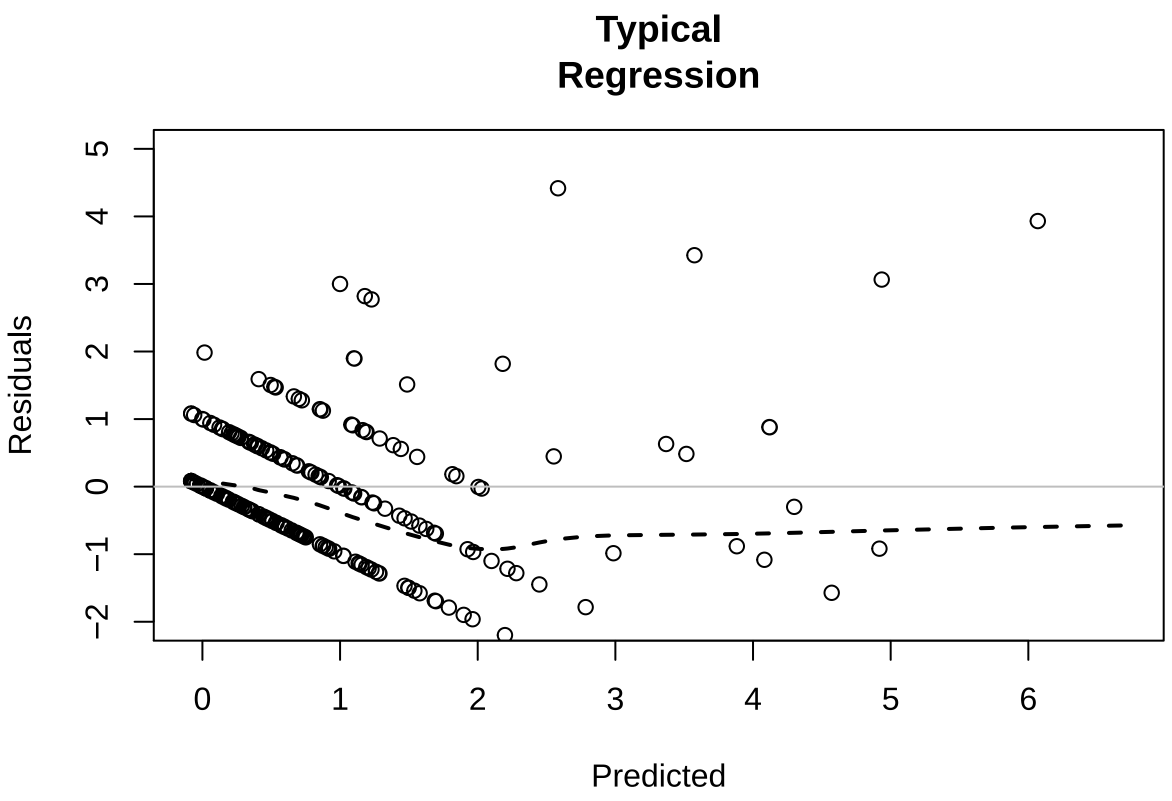
7.8 Regression diagnostics

- Residual plots for the models

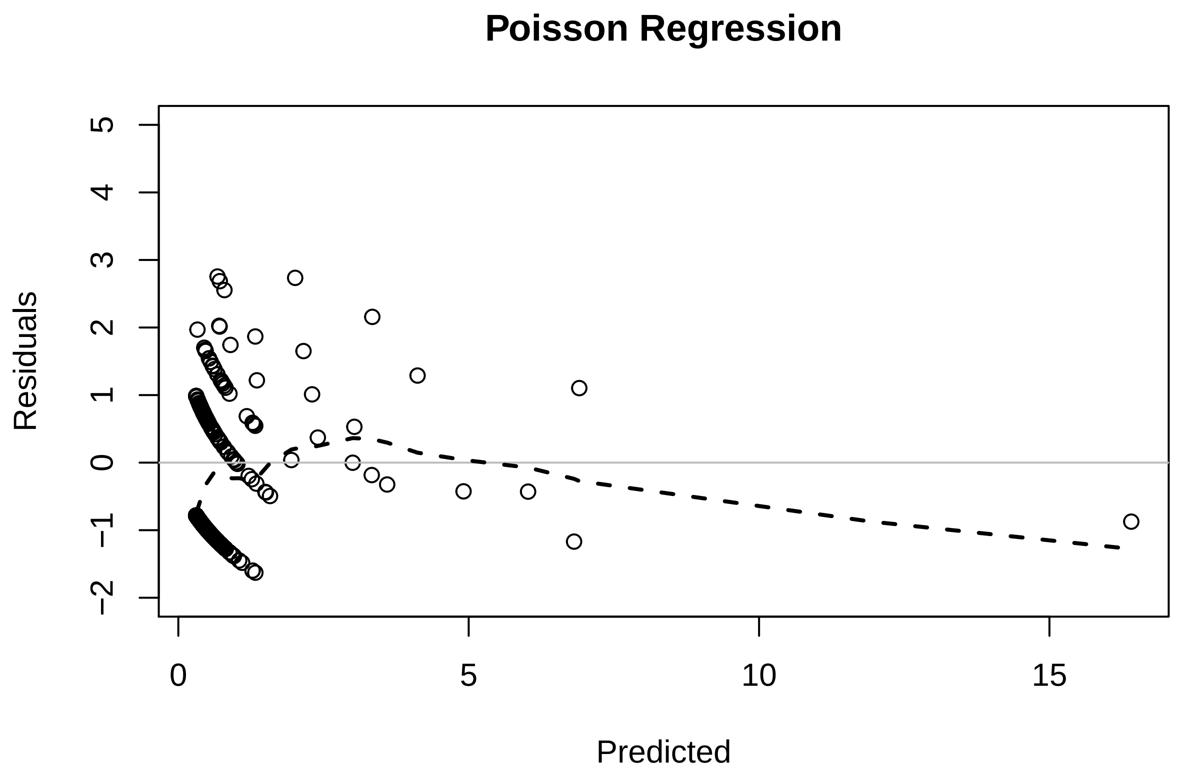
```
# Diagnostics
# normal residuals density plot
plot(density(residuals(defects_normal)))
```



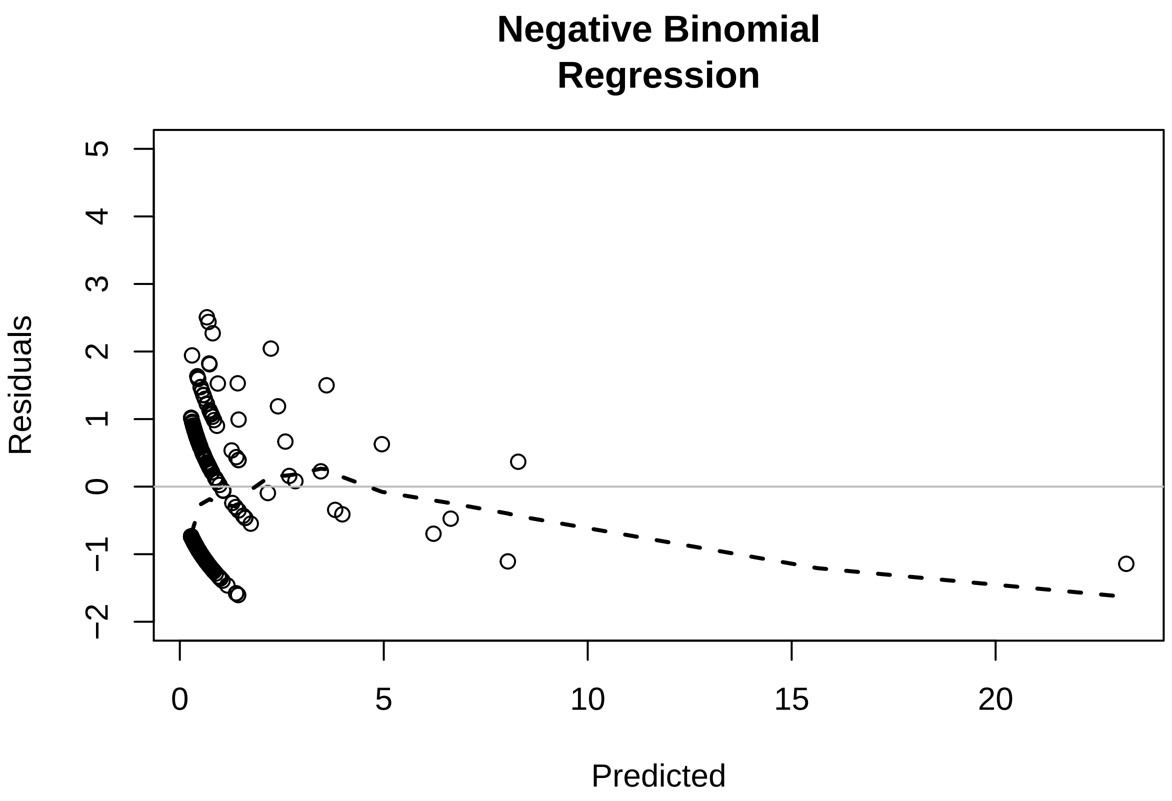
```
# predicted vs. residual plots
# typical
plot(predict(defects_normal, type="response"), residuals(defects_normal), main="Typical
Regression", ylab="Residuals", xlab="Predicted", ylim=c(-2,5))
abline(h=0,lty=1,col="gray")
lines(lowess(predict(defects_normal,type="response"),residuals(defects_normal)), lwd=2, lty=2)
```



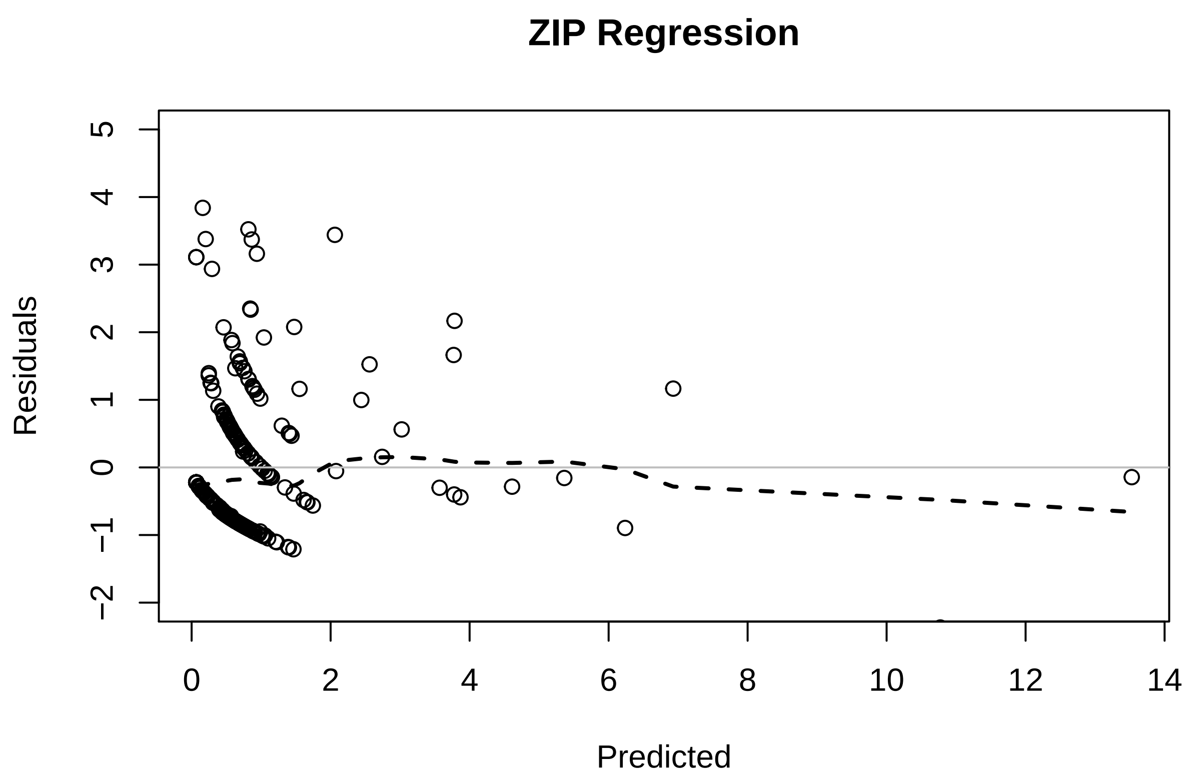
```r
# poisson
plot(predict(defects_pois,type="response"),residuals(defects_pois), main="Poisson Regression",
ylab="Residuals", xlab="Predicted", ylim=c(-2,5))
abline(h=0,lty=1,col="gray")
lines(lowess(predict(defects_pois,type="response"),residuals(defects_pois)),lwd=2, lty=2)
```



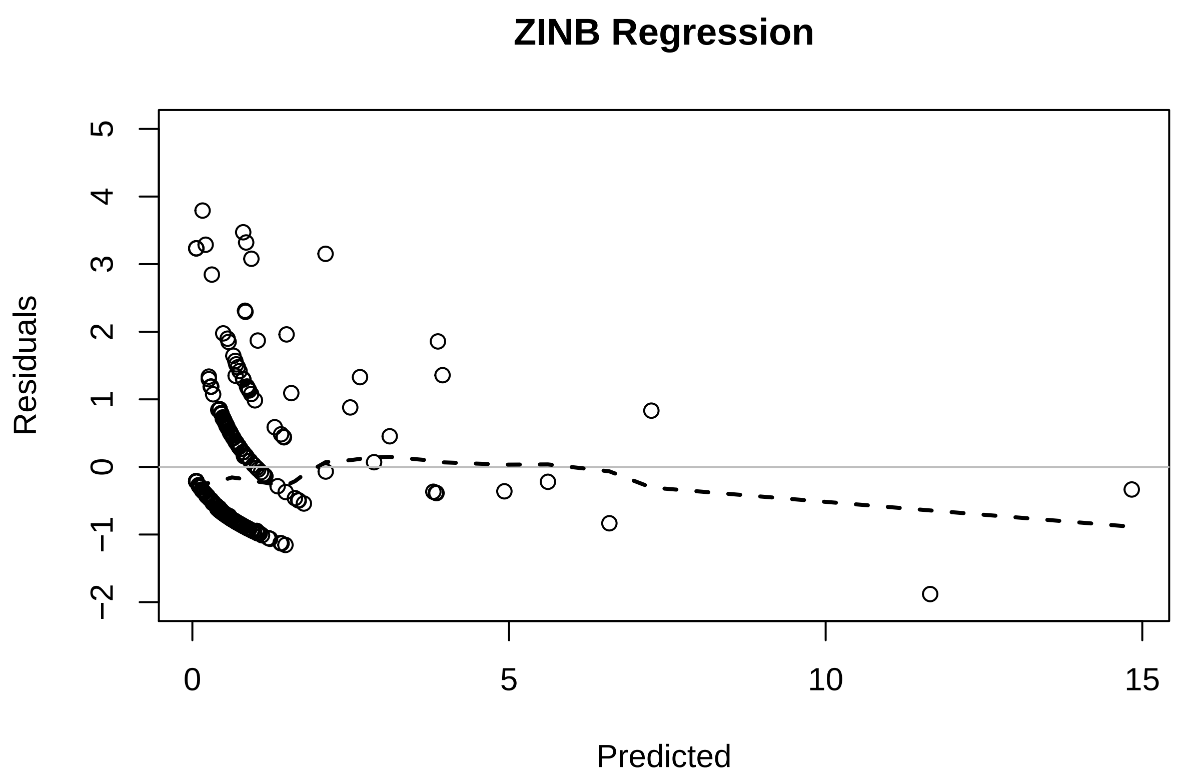
```
# negative binomial
plot(predict(defects_negbinom,type="response"),residuals(defects_negbinom), main="Negative Binomial
Regression", ylab="Residuals", xlab="Predicted", ylim=c(-2,5))
abline(h=0,lty=1,col="gray")
lines(lowess(predict(defects_negbinom,type="response"),residuals(defects_negbinom)), lwd=2, lty=2)
```



```
# ZIP
plot(predict(defects_zip,type="response"),residuals(defects_zip), main="ZIP Regression",
ylab="Residuals", xlab="Predicted", ylim=c(-2,5))
abline(h=0,lty=1,col="gray")
lines(lowess(predict(defects_zip,type="response"),residuals(defects_zip)),lwd=2, lty=2)
```

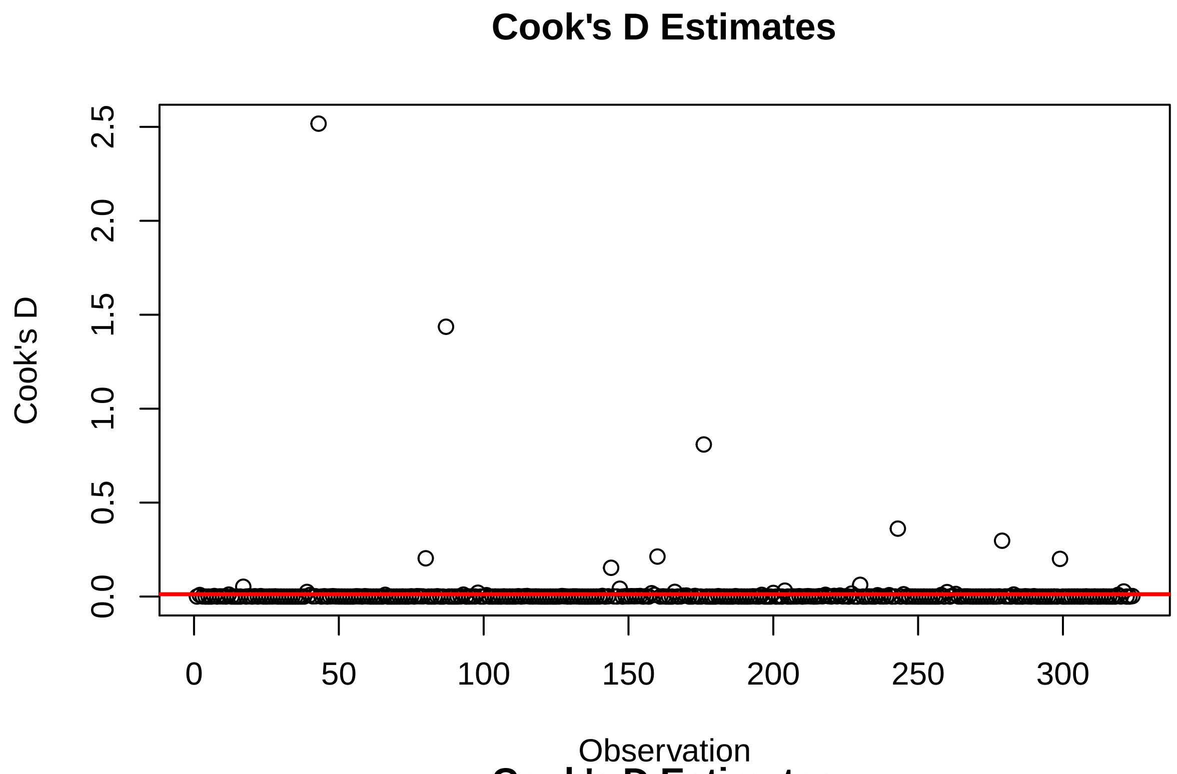


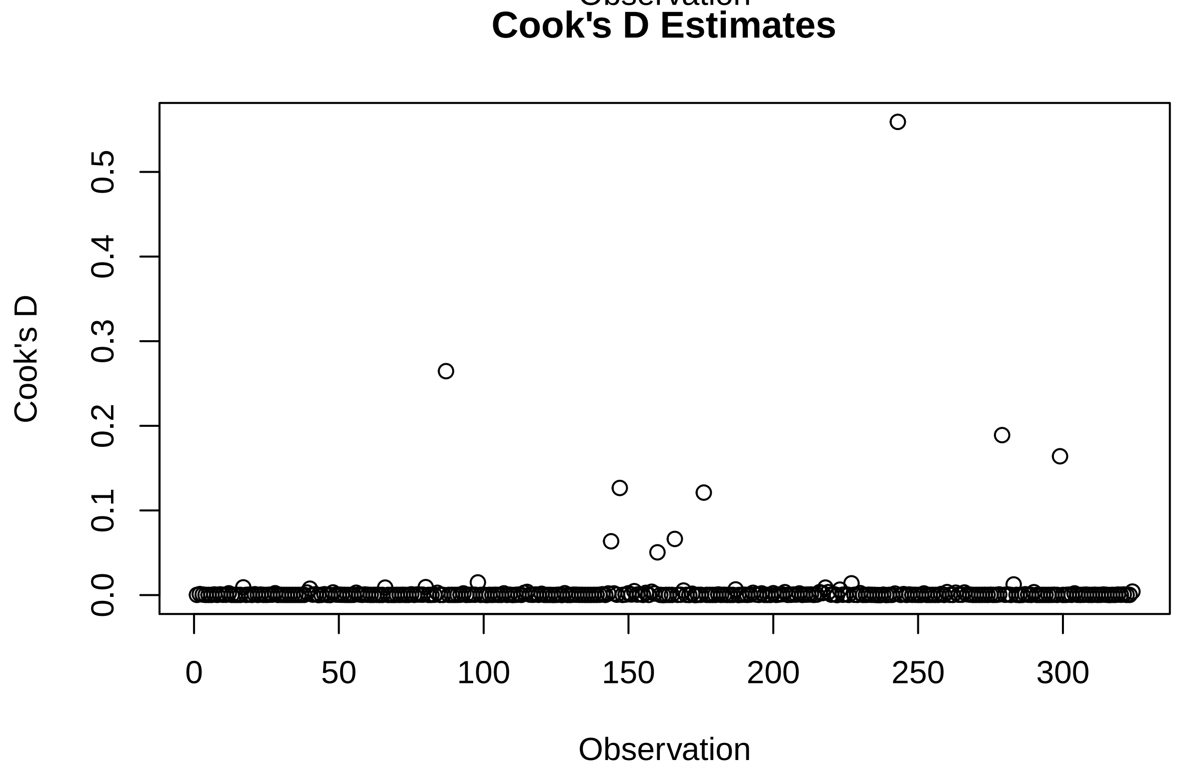
```
# ZINB
plot(predict(defects_zinb,type="response"),residuals(defects_zinb), main="ZINB Regression",
ylab="Residuals", xlab="Predicted", ylim=c(-2,5))
abline(h=0,lty=1,col="gray")
lines(lowess(predict(defects_zinb,type="response"),residuals(defects_zinb)),lwd=2, lty=2)
```

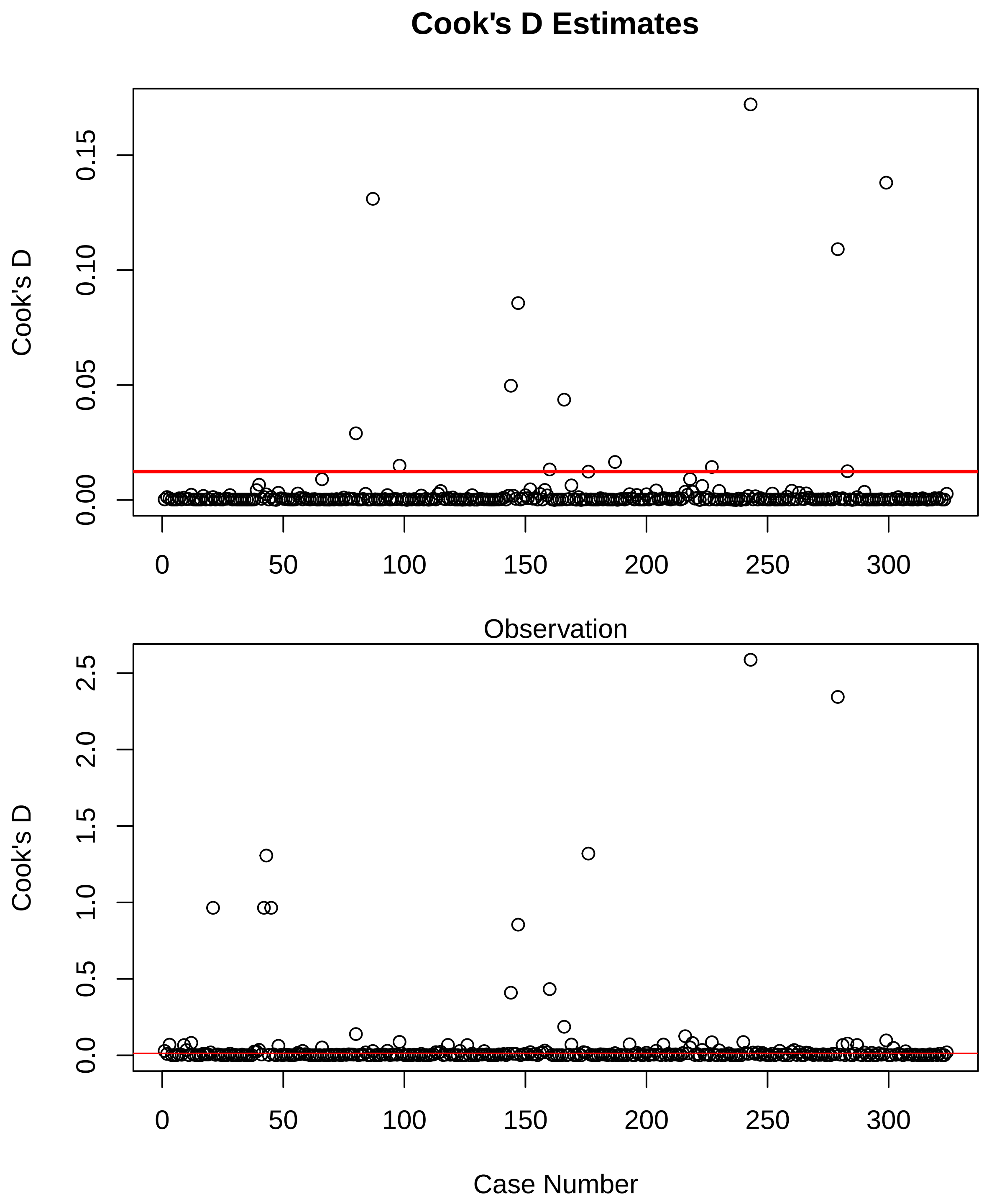


7.8.1 Cook's D for the ZIP

Cook's D Cook's distance or Cook's D is a commonly used for detecting highly influential observations.







7.8.2 ZIP Model variable selection for the Equinox using AIC and BIC

```
# calculate and combine AIC, AIC weights, and BIC
results <- data.frame(models = model.names)
kpar <- log(nrow(equinox))
results$bic.val <- unlist(lapply(cand.models, AIC, k = kpar))
results$bic.rank <- rank(results$bic.val)
```

```
results$aic.val <- unlist(lapply(cand.models, AIC))
results$aic.delta <- results$aic.val-min(results$aic.val)
results$aic.likelihood <- exp(-0.5* results$aic.delta)
results$aic.weight <- results$aic.likelihood/sum(results$aic.likelihood)
# sort models by AIC weight
results <- results[rev(order(results[, "aic.weight"])),]
results$cum.aic.weight <- cumsum(results[, "aic.weight"])
results
```

```
##                  models  bic.val bic.rank  aic.val aic.delta
## 6      cbo_lcom_wmc_rfc 633.3807        1 606.9155  0.000000
## 5  LOC+cbo+lcom+wmc+rfc 638.6139        2 608.3680  1.452498
## 2               LOC+cbo 641.6528        3 622.7491 15.833574
## 3          LOC+cbo+lcom 646.0690        4 623.3845 16.468999
## 4      LOC+cbo+lcom+wmc 650.0853        5 623.6201 16.704643
## 7          lcom_wmc_rfc 683.1122        6 660.4278 53.512284
## 1                   LOC 686.4209        7 671.2979 64.382419
## 9                   wmc 697.7706        8 682.6476 75.732128
## 8               wmc_rfc 701.9812        9 683.0775 76.162002
## 10                  rfc 709.0393       10 693.9163 87.000801
##    aic.likelihood   aic.weight cum.aic.weight
## 6    1.000000e+00 6.735886e-01      0.6735886
## 5    4.837200e-01 3.258282e-01      0.9994168
## 2    3.645717e-04 2.455714e-04      0.9996624
## 3    2.653397e-04 1.787298e-04      0.9998411
## 4    2.358484e-04 1.588648e-04      1.0000000
## 7    2.398585e-12 1.615659e-12      1.0000000
## 1    1.046009e-14 7.045794e-15      1.0000000
## 9    3.589033e-17 2.417531e-17      1.0000000
## 8    2.894885e-17 1.949962e-17      1.0000000
## 10   1.282378e-19 8.637951e-20      1.0000000
```

```
# # final model
defects_final_zip <- zeroinfl(count ~ cbo + lcom + wmc + rfc| numberOfLinesOfCode, data = equinox, link
defects_final_zip
```

```
##
## Call:
## zeroinfl(formula = count ~ cbo + lcom + wmc + rfc | numberOfLinesOfCode,
##     data = equinox, dist = "poisson", link = "logit", trace = FALSE)
##
## Count model coefficients (poisson with log link):
## (Intercept)          cbo         lcom          wmc          rfc
##  -0.8641763    0.0433784    0.0002456    0.0097291   -0.0051994
##
## Zero-inflation model coefficients (binomial with logit link):
##         (Intercept)  numberOfLinesOfCode
##              1.6749              -0.1657
```

7.9 References and R code used

1. A. Alexander Beaujean, Grant B. Morgan, Tutorial on Using Regression Models with Count Outcomes using R. Practical Assessment, Research & Evaluation, Volume 21, Number 2, February 2016

2. Achim Zeileis, Christian Kleiber and Simon Jackman, Regression Models for Count Data in R

3. ZERO-INFLATED POISSON REGRESSION | R DATA ANALYSIS EXAMPLES. UCLA: Statistical Consulting Group, from http://stats.idre.ucla.edu/r/dae/zinb/ (accessed May 20, 2017)

4. Taghi M. Khoshgoftaar , Kehan Gao & Robert M. Szabo, (2005) Comparing software fault predictions of pure and zero-inflated Poisson regression models, International Journal of Systems Science, 36:11, 705-715, http://dx.doi.org/10.1080/00207720500159995

5. Taghi M. Khoshgoftaar , Kehan Gao & Robert M. Szabo, An Application of Zero-Inflated Poisson Regression for Software Fault Prediction, 2001

6. Marco D'Ambros, Michele Lanza, Romain Robbes.Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empir Software Eng (2012) 17:531–577 DOI 10.1007/s10664-011-9173-9

7. A.F. Zuur et al., Chapter 11 Zero-Truncated and Zero-Inflated Models for Count Data in Mixed Effects Models and Extensions in Ecology with R, Springer, 2009

Part IV

Evaluation

Chapter 8

Evaluation of Models

Once we obtain the model with the training data, we need to evaluate it with some new data (testing data).

> **No Free Lunch theorem** In the absence of any knowledge about the prediction problem, no model can be said to be uniformly better than any other

8.1 Building and Validating a Model

We cannnot use the the same data for training and testing (it is like evaluating a student with the exercises previouly solved in class, the sudent's marks will be "optimistic" and we do not know about student capability to generalise the learned concepts).

Therefore, we should, at a minimun, divide the dataset into *training* and *testing*, learn the model with the training data and test it with the rest of data as explained next.

8.1.1 Holdout approach

Holdout approach consists of dividing the dataset into *training* (typically approx. 2/3 of the data) and *testing* (approx 1/3 of the data). + Problems: Data can be skewed, missing classes, etc. if randomly divided. Stratification ensures that each class is represented with approximately equal proportions (e.g., if data contains aprox 45% of positive cases, the training and testing datasets should mantain similar proportion of positive cases).

Holdout estimate can be made more reliable by repeating the process with different subsamples (repeated holdout method)

The error rates on the different iterations are averaged (overall error rate)

- Usually, part of the data points are used for building the model and the remaining points are used for validating the model. There are several approaches to this process.
- *Validation Set approach*: it is the simplest method. It consists of randomly dividing the available set of oservations into two parts, a *training set* and a *validation set* or hold-out set. Usually 2/3 of the data points are used for training and 1/3 is used for testing purposes.

8.1.2 Cross Validation (CV)

k-fold Cross-Validation involves randomly dividing the set of observations into k groups, or folds, of approximately equal size. One fold is treated as a validation set and the method is trained on the remaining $k-1$

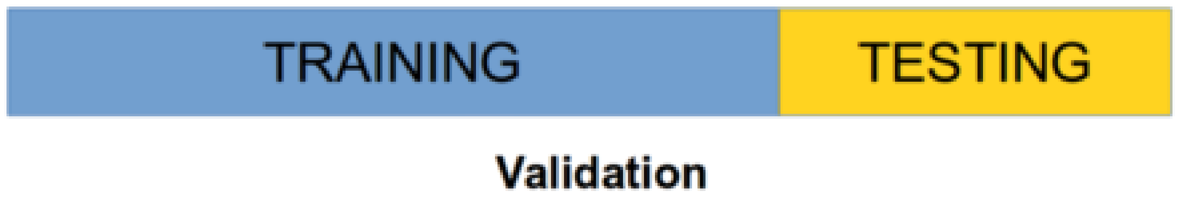


Figure 8.1: Hold out validation

folds. This procedure is repeated k times. If k is equal to n we are in the previous method.

- 1st step: split dataset ($\mathcal{D}$) into k subsets of approximatelly equal size $C_1, \ldots, C_k$
- 2nd step: we construct a dataset $D_i = D - C_i$ used for training and test the accuracy of the classifier D_i on C_i subset for testing

Having done this for all k we estimate the accuracy of the method by averaging the accuracy over the k cross-validation trials

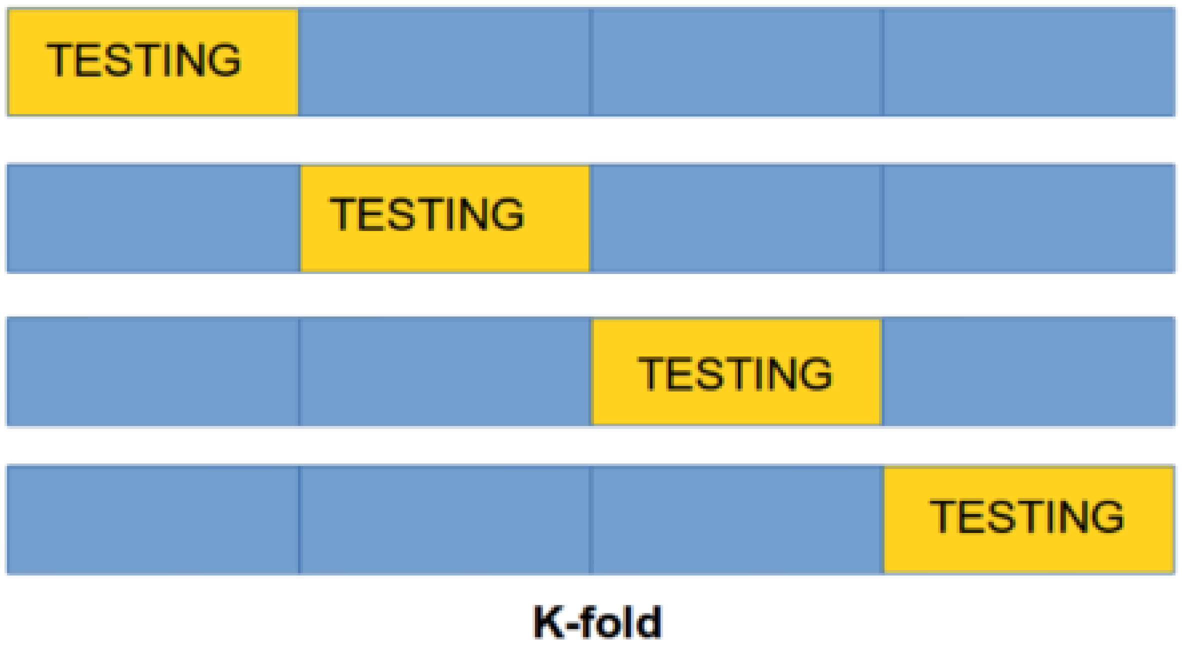


Figure 8.2: k-fold

- *Leave-One-Out Cross-Validation*: This is a special case of CV. Instead of creating two subsets for training and testing, a single observation is used for the validation set, and the remaining observations make up the training set. This approach is repeated n times (the total number of observations) and the estimate for the test mean squared error is the average of the n test estimates.

8.2 Evaluation of Classification Models

The confusion matrix (which can be extended to multiclass problems). The following table shows the possible outcomes for binary classification problems:

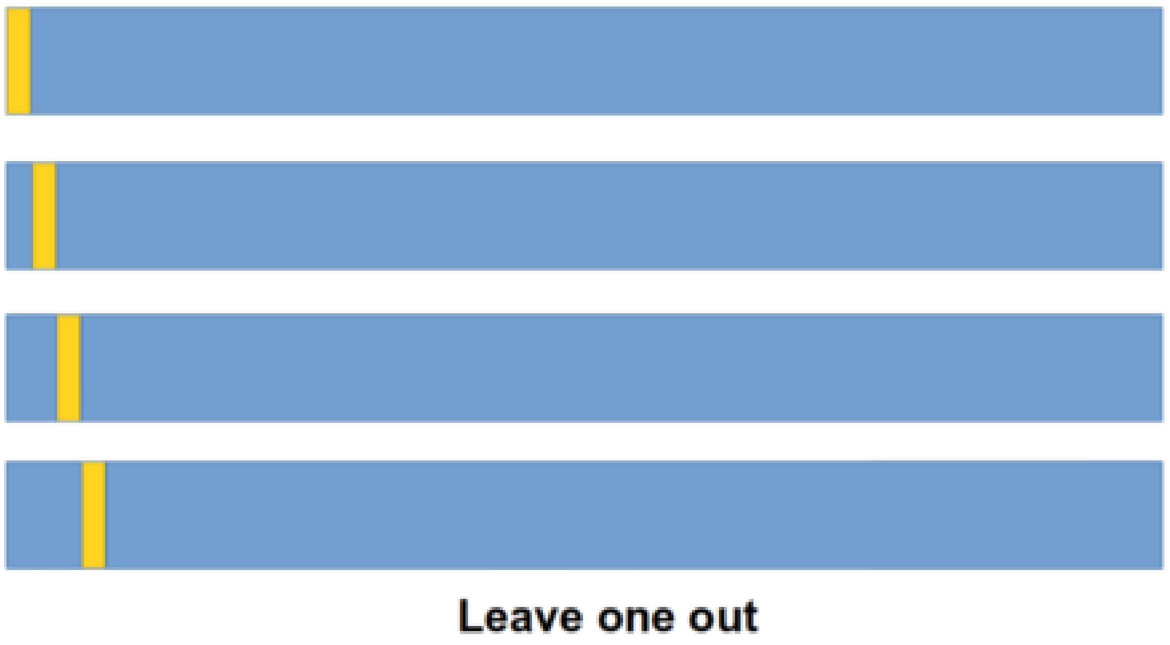


Figure 8.3: Leave One Out

	$PredPos$	$PredNeg$
$ActPos$	TP	FN
$ActNeg$	FP	TN

where *True Positives* (TP) and *True Negatives* (TN) are respectively the number of positive and negative instances correctly classified, *False Positives* (FP) is the number of negative instances misclassified as positive (also called Type I errors), and *False Negatives* (FN) is the number of positive instances misclassified as negative (Type II errors).

From the confusion matrix, we can calculate:

- *True positive rate*, or *recall* ($TP_r = recall = r = TP/TP + FN$) is the proportion of positive cases correctly classified as belonging to the positive class.
- *False negative rate* ($FN_r = FN/TP+FN$) is the proportion of positive cases misclassified as belonging to the negative class.
- *False positive rate* ($FP_r = FP/FP+TN$) is the proportion of negative cases misclassified as belonging to the positive class.
- *True negative rate* ($TN_r = TN/FP + TN$) is the proportion of negative cases correctly classified as belonging to the negative class.

There is a tradeoff between FP_r and FN_r as the objective is minimize both metrics (or conversely, maximize the true negative and positive rates). It is possible to combine both metrics into a single figure, predictive *accuracy*:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

to measure performance of classifiers (or the complementary value, the *error rate* which is defined as $1 - accuracy$)

- Precision, fraction of relevant instances among the retrieved instances,

$$\frac{TP}{TP + FP}$$

- Recall$ (*sensitivity* probability of detection, PD) is the fraction of relevant instances that have been retrieved over total relevant instances, $\frac{TP}{}$

8.2.1 Prediction in probabilistic classifiers

A probabilistic classifier estimates the probability of each of the posible class values given the attribute values of the instance $P(c|x)$. Then, given a new instance, x, the class value with the highest a posteriori probability will be assigned to that new instance (the *winner takes all* approach):

$\psi(x) = argmax_c(P(c|x))$

8.3 Other Metrics used in Software Engineering with Classification

In the domain of defect prediction and when two classes are considered, it is also customary to refer to the *probability of detection*, (pd) which corresponds to the True Positive rate (TP_{rate} or *Sensitivity*) as a measure of the goodness of the model, and *probability of false alarm* (pf) as performance measures~Menzies et al. (2007b).

The objective is to find which techniques that maximise pd and minimise pf. As stated by Menzies et al., the balance between these two measures depends on the project characteristics (e.g. real-time systems vs. information management systems) it is formulated as the Euclidean distance from the sweet spot $pf = 0$ and $pd = 1$ to a pair of (pf, pd).

$$balance = 1 - \frac{\sqrt{(0 - pf^2) + (1 - pd^2)}}{\sqrt{2}}$$

It is normalized by the maximum possible distance across the ROC square $(\sqrt{2}, 2)$, subtracted this value from 1, and expressed it as a percentage.

8.4 Graphical Evaluation

8.4.1 Receiver Operating Characteristic (ROC)

The *Receiver Operating Characteristic* (*ROC*)(Fawcett, 2006) curve which provides a graphical visualisation of the results.

The Area Under the ROC Curve (AUC) also provides a quality measure between positive and negative rates with a single value.

A simple way to approximate the AUC is with the following equation: $AUC = \frac{1+TP_r-FP_r}{2}$

8.4.2 Precision-Recall Curve (PRC)

Similarly to ROC, another widely used evaluation technique is the Precision-Recall Curve (PRC), which depicts a trade off between precision and recall and can also be summarised into a single value as the Area Under the Precision-Recall Curve (AUPRC)~**?**.

%AUPCR is more accurate than the ROC for testing performances when dealing with imbalanced datasets as well as optimising ROC values does not necessarily optimises AUPR values, i.e., a good classifier in AUC space may not be so good in PRC space. %The weighted average uses weights proportional to class frequencies in the data. %The weighted average is computed by weighting the measure of class (TP rate, precision, recall ...) by the proportion of instances there are in that class. Computing the average can be sometimes be misleading. For instance, if class 1 has 100 instances and you achieve a recall of 30%, and

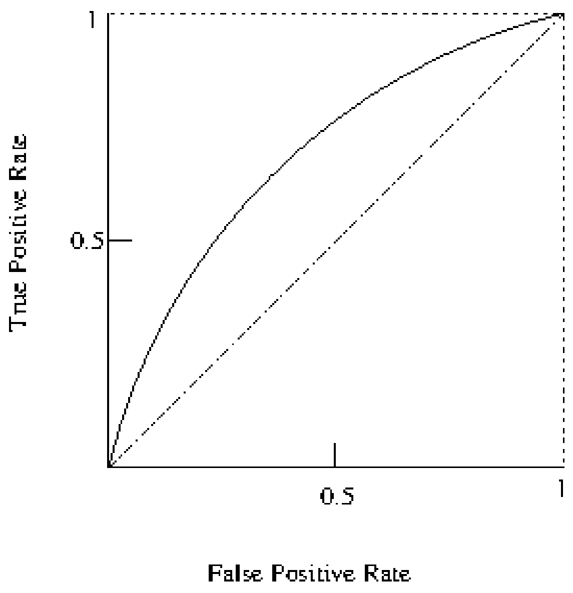


Figure 8.4: Receiver Operating Characteristic

class 2 has 1 instance and you achieve recall of 100% (you predicted the only instance correctly), then when taking the average (65%) you will inflate the recall score because of the one instance you predicted correctly. Taking the weighted average will give you 30.7%, which is much more realistic measure of the performance of the classifier.

8.5 Numeric Prediction Evaluation

RSME

Mean Square Error $= MSE = \frac{(p_1-a_1)^2+...+(p_n-a_n)^2}{n}$

$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - y_i)^2$

$RMSD = \sqrt{\frac{\sum_{t=1}^{n}(\hat{y}_t-y)^2}{n}}$

Mean-absolute error MAE

$\frac{|p_1-a_1|+...+|p_n-a_n|}{n}$

Relative absolute error:

$RAE = \frac{\sum_{i=1}^{N}|\hat{\theta}_i-\theta_i|}{\sum_{i=1}^{N}|\bar{\theta}-\theta_i|}$

Root relative-squared error:

$RAE = \sqrt{\frac{\sum_{i=1}^{N}|\hat{\theta}_i-\theta_i|}{\sum_{i=1}^{N}|\bar{\theta}-\theta_i|}}$

where $\hat{\theta}$ is a mean value of θ.

Relative-squared error $\frac{(p_1-a_1)^2+...+(p_n-a_n)^2}{(a_1-\hat{a})^2+...+(a_n-\hat{a})^2}$ ($\hat{a}$ is the mean value over the training data)

Relative Absolut Error

Correlation Coefficient

Correlation coefficient between two random variables X and Y is defined as

$$\rho(X,Y) = \frac{\mathbf{Cov}(X,Y)}{\sqrt{\mathbf{Var}(X)\mathbf{Var}(Y)}}.$$

The {*sample correlation coefficient*} *r between two samples* x_i *and* y_j *is defined as* $r = S_{xy}/\sqrt{S_{xx}S_{yy}}$.

Example: Is there any linear relationship between the effort estimates (p_i) *and actual effort* (a_i)*?*

$a \| 39, 43, 21, 64, 57, 47, 28, 75, 34, 52$

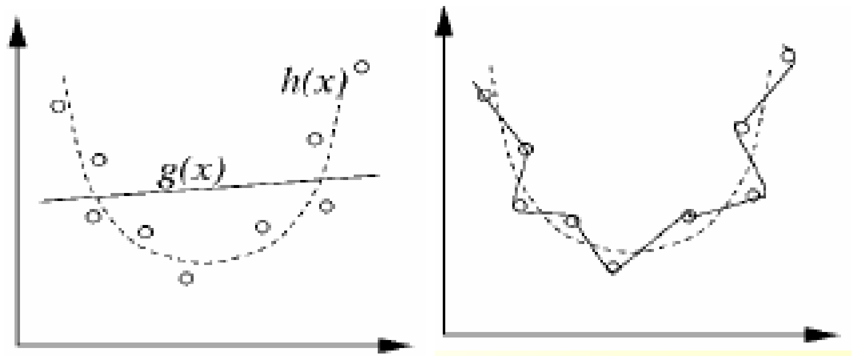
$p \| 65, 78, 52, 82, 92, 89, 73, 98, 56, 75$

```
p<-c(39,43,21,64,57,47,28,75,34,52)
a<-c(65,78,52,82,92,89,73,98,56,75)
#
cor(p,a)
```

```
## [1] 0.8397859
```

R^2

8.6 Underfitting vs. Overfitting



For example, increasing the tree size, decreases the training and testing errors. However, at some point after (tree complexity), training error keeps decreasing but testing error increases. Many algorithms have parameters to determine the model complexity (e.g., in decision trees is the prunning parameter)

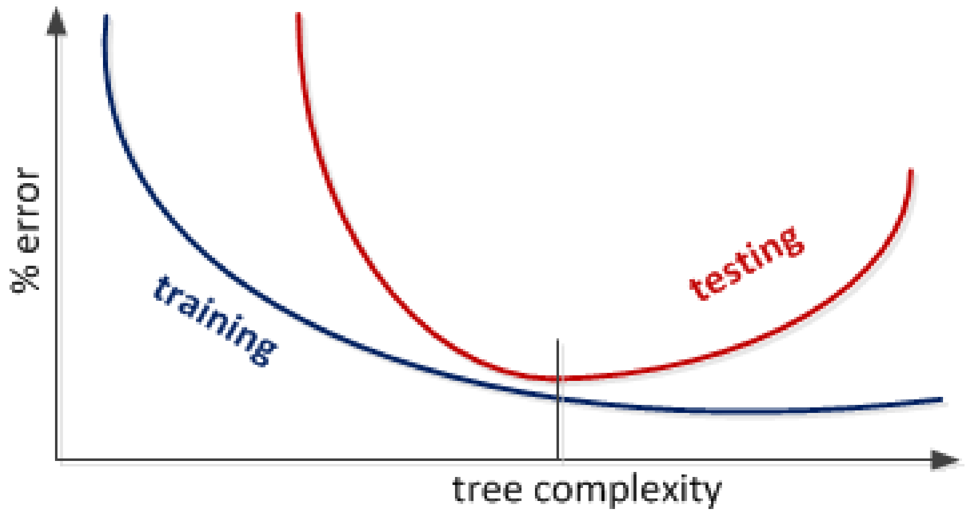


Figure 8.5: Overfitting in trees

Part V

Problems and Issues in Defect Prediction

Chapter 9

Data Mining Issues in Defect Prediction

There are many challenges related to Software Mining and fault prediction such as:

- *fault prediction without fault data*
- *limited fault data*
- *noisy data*
- *cross-company vs within-company.*

Catal highlights some of these issues (Catal, 2012).

Chapter 10

Outliers, missing values, inconsistencies data noise

Although this statistical problem is well known in the literature, it is not always properly reported.

For example preprocessing of the NASA datasets hosted at PROMISE as some concerns about their quality have been raised (Gray et al., 2011b). And these are by far the most used datasets.

Some example of works/techniques to address noise and data quality in software engineering datasets are been proposed such as the works by (Khoshgoftaar and Van Hulse, 2009).

Chapter 11

Redundant and irrelevant attributes and instances

It is also well known that the existence of irrelevant and redundant features in the datasets has a negative impact on most data mining algorithms, which assume a certain level of balance between the class attributes.

Feature Selection has been applied and studied by the software engineering community (Khoshgoftaar et al., 2012a),

Futhermore, feature selection algorithms do not perform well with unbalanced datasets (defect prediction datasets tend to be are highly unbalanced), resulting in a selection of metrics that are not adequate for the learning algorithms See (Wang et al., 2012)(Rodriguez et al., 2007).

However instance selection that needs further research. For example, the JM1 dataset rom NASA's defect prediction datasets in PROMISE has 8,000 repeated instances

*A few exceptions for effort estimation include (**?**),(Chen et al., 2005).*

Chapter 12

Overlapping or class separability

When dealing with classification, we may also face the problem of overlapping between classes in which a region of the data space contains samples from different values for the class.

Many samples from the NASA dataset contained in the PROMISE repository are contradictory or inconsistent, many instances have the same values for all attributes with the exception of the class, making the induction of good predictive models difficult.

Chapter 13

Data shifting

The data shift problem happens when the test data distribution differs from the training distribution. Turhan discusses the dataset shift problem in software engineering (effort estimation and defect prediction) (Turhan, 2012). This can also happen when divide the data into k-folds for cross validation (Raudys and Jain, 1991)

Chapter 14

Imbalance datasets

This happens when samples of some classes vastly outnumber the cases of other classes. Under this situation, when the imbalanced data is not considered, many learning algorithms generate distorted models for which the impact of some factors can be hidden and the prediction accuracy can be misleading (He and Garcia, 2009). This is due to the fact that most data mining algorithms assume balanced datasets. The imbalance problem is known to affect many machine learning algorithms such as decision tress, neural networks or support vectors machines.

Chapter 15

Evaluation metrics and the evaluation of models

Arisholm et al. (Arisholm et al., 2010) compare different data mining techniques (classification tree algorithm (C4.5), a coverage rule algorithm (PART), logistic regression, back–propagation neural work and support vector machines) over 13 releases of a Telecom middleware software developed in Java using three types metrics: (i) object oriented metrics, (ii) delta measures, amount of change between successive releases, and (iii) process measures from a configuration management system. The authors concluded that although there are no significant differences regarding the techniques used, large differences can be observed depending on the criteria used to compare them. The authors also propose a cost–effectiveness measure based on the AUC and number of statements so that larger modules are more expensive to test. The same approach of considering module size in conjunction with the AUC as evaluation measure has been explored by Mende and Koschke (Mende and Koschke, 2010). These can be considered as cost-sensitive classification approaches.

There is also another controversy regarding the AUC measure, which is widely used to compare classifiers particularly when data is imbalanced. Hand anlysed that the AUC is not a coherent measure and proposed the $H - measure$ *(Hand, 2009).*

Another controversy concerns the use of the t-test for comparison of multiple algorithms and datasets. As this is a parametric test it does not seem to be the most appropriate (Demšar, 2006).

Also currently there is a tendency to rely less on p-values and more on confidence intervales and Equivalence Hipothesis Testing (Dolado et al., 2016)

Dicussion:

(Menzies et al., 2007b) Data Mining Static Code Attributes to Learn Defect Predictors

(Zhang and Zhang, 2007) Comments on "Data Mining Static Code Attributes to Learn Defect Predictors"

(Menzies et al., 2007a) Problems with Precision: A Response to Comments on Data Mining Static Code Attributes to Learn Defect Predictors

Chapter 16

Cross project defect prediction

Most studies find classifiers in single defect prediction datasets. Some exceptions include:

(Hosseini et al., 2017) A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction

(Zimmermann et al., 2009) Cross-project Defect Prediction

(Canfora et al., 2013) Multi-objective Cross-Project Defect Prediction

(Panichella et al., 2014) Cross-project defect prediction models: L'Union fait la force

This paper concluded that different classifiers are not equivalent and they can complement each other. This is also confirmed by Bowes et al. (Bowes et al., 2015)

Part VI

Examples

Chapter 17

Feature Subsect Selection in Defect Prediction

Feature Subsect Selection (FSS) is important in different ways:

- *A reduced volume of data allows different data mining or searching techniques to be applied.*
- *Irrelevant and redundant attributes can generate less accurate and more complex models. Furthermore, data mining algorithms can be executed faster.*
- *We can avoid the collection of data for those irrelevant and redundant attributes in the future.*

FSS algorithms search through candidate feature subsets guide by a certain evaluation measure which captures the goodness of each subset. An optimal (or near optimal) subset is selected when the search stops. There are two possibilities when applying FSS:

- *The* filter model *relies on general characteristics of the data to evaluate and select feature subsets without involving any data mining algorithm.*
- *The* wrapper model *requires one predetermined mining algorithm and uses its performance as the evaluation criterion. It searches for features better suited to the mining algorithm aiming to improve mining performance, but it also tends to be more computationally expensive than filter model.*

We applied Feature Subset Selection (FSS) to several datasets publicly available (PROMISE repository), and different classifiers to improve the detection of faulty modules. (Rodriguez et al., 2007) Detecting Fault Modules Applying Feature Selection to Classifiers

Khoshgoftaar et al have studied FS extensively together with imbalance, noise etc. For example in:

(Khoshgoftaar et al., 2012b) Exploring an iterative feature selection technique for highly imbalanced data sets

Chapter 18

Imbalanced data

Most publicly available datasets in software defect prediction are highly imbalanced, i.e., samples of non-defective modules vastly outnumber the defective ones.

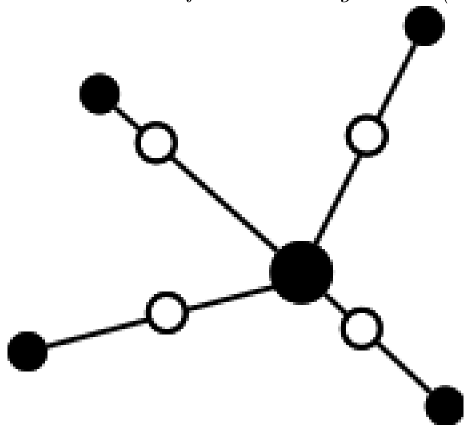
Data mining algorithms generate poor models because they tend to optimize the overall accuracy or AUC but perform badly in classes with very few samples (minority class which is usually the one we are interested in).

It is typically addressed by preprocessing the datasets with sampling techniques [Afzal2012IJSEKE] or considering cost in the data mining algorithms (making the algorithms more robust). This problem happens in many of the defect prediction datasets (e.g. the PROMISE repository has around 60 defect prediction datasets). The previous problems, redundant and irrelevant attributes, overlapping, data shifting and small datasets are made worse when datasets are imbalanced (Fernández et al., 2011) and ad-hoc approaches may be needed (Khoshgoftaar et al., 2012b) in feature selection.

Different approaches include:

- Sampling*: Sampling techniques are classified as oversampling or undersampling and are based on adding or removing instances of the training dataset*
- *Random OverSampling (ROS) replicates instances from the minority class towards a more balanced distribution*
- *Random Under-Sampling (RUS) removes instances from the majority class*

More intelligent approaches that random sampling include SMOTE (Synthetic Minority Over-sampling Technique) that generates new instances based on a number of nearest neighbours (k-NN). There are other



variations of SMOTE (Borderline SMOTE)

- Cost-Sensitive Classifiers *(CSC) penalises differently the type of errors*

- Ensembles*: Bagging (Bootstrap aggregating), boosting and stacking (Stacked generalization) which combines different types of models*
- Robust algorithms*: algorithms or modification to algorithmsdesigned to work with unbalanced data (Ibarguren et al., 2017)*

(Seiffert et al., 2007) An empirical study of the classification performance of learners on imbalanced and noisy software quality data

Chapter 19

Subgroup Discovery

Subgroup Discovery (SD) aims to find subgroups of data that are statistically different given a property of interest. (Klösgen, 1996) (Wrobel, 1997) (Wrobel, 2001)

A comprehensive survey by (Herrera et al., 2011).

SD lies between predictive (finding rules given historical data and a property of interest) and descriptive tasks (discovering interesting patterns in data). An important difference with classification tasks is that the SD algorithms only focus on finding subgroups (e.g., inducing rules) for the property of interest and do not necessarily describe all instances in the dataset.

In general, subgroups are represented through rules *with the form* $\{Cond \rightarrow Class\}$ *having as consequent* $(Class)$ *a specific value of an attribute of interest. The antecedent* $(Cond)$ *is usually composed of a conjunction of attribute–value pairs through relational operators. Discrete attributes can have the form of* $att = val$ *or* $att \neq val$ *and for continuous attributes ranges need to be defined, i.e.,* $val_1 \leq att \leq val_2$.

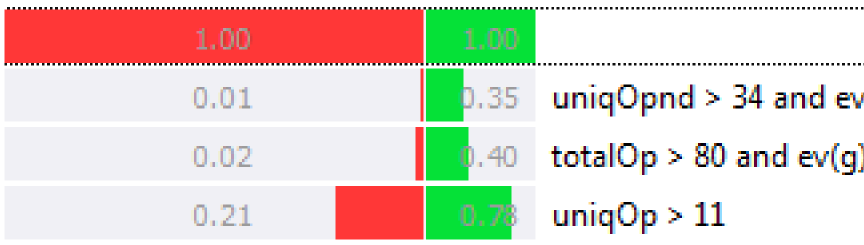
	Classification	*Subgroup Discovery*
Induction Type	*Predictive*	*Descriptive*
Output	*Set of classification rules*	*Individual rules to describe subgroups*
Purpose	*To learn a classification model*	*To find interesting and interretable patterns*

It is worth noting that we do not cover all examples

This has been applied to deal with unbalanced data (Rodriguez et al., 2012)

Classical algorithms include the SD, CN2-SD algorithms etc.

Using the CN2-SD algorithm with the KC2 dataset:



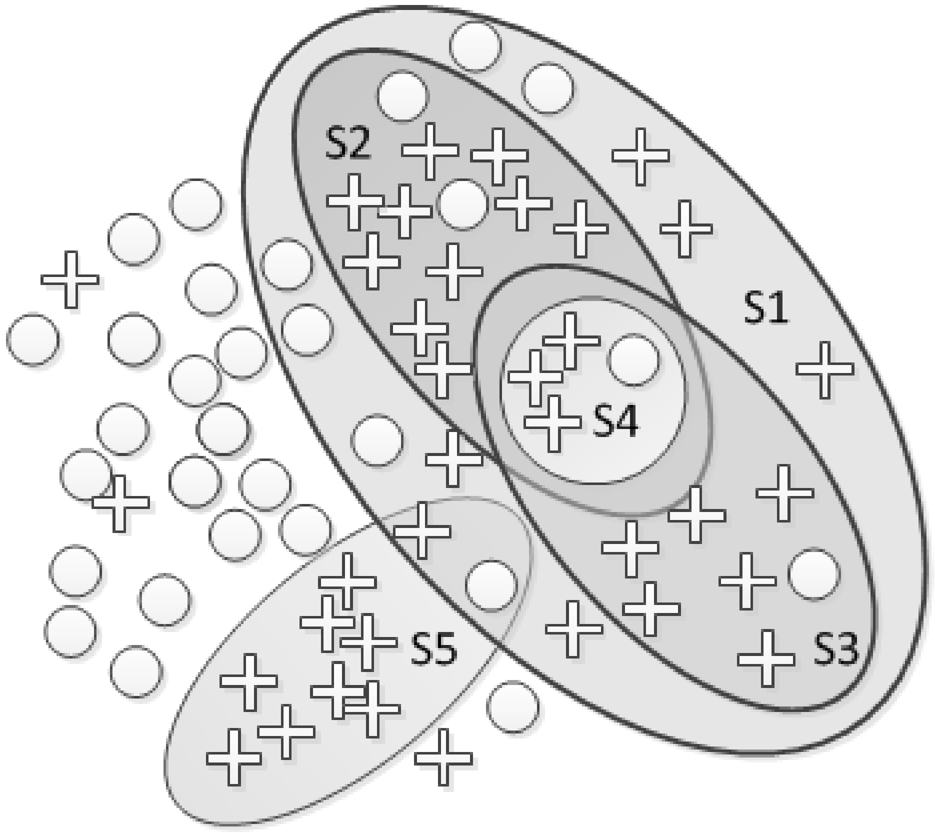


Figure 19.1: SD Rules

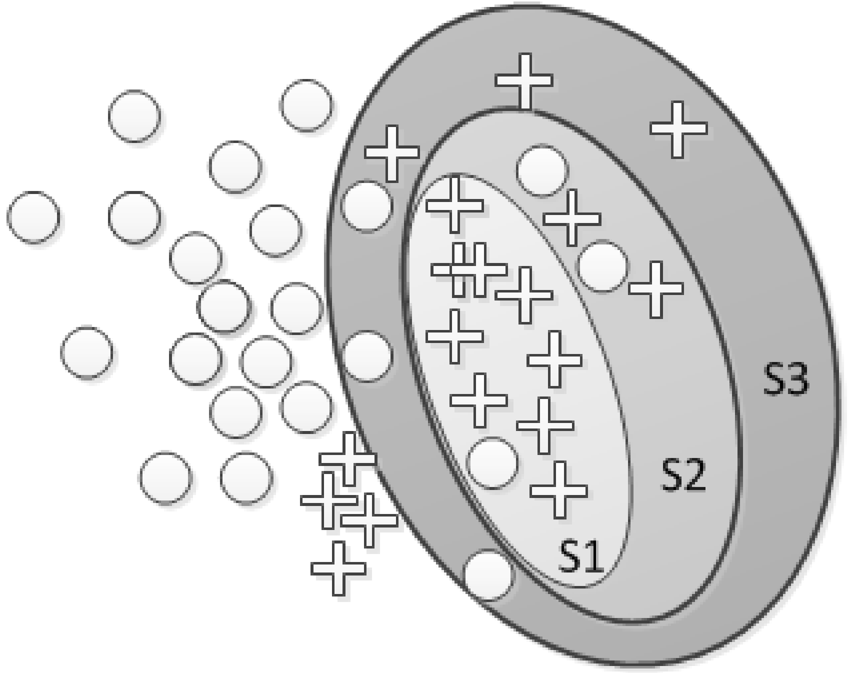
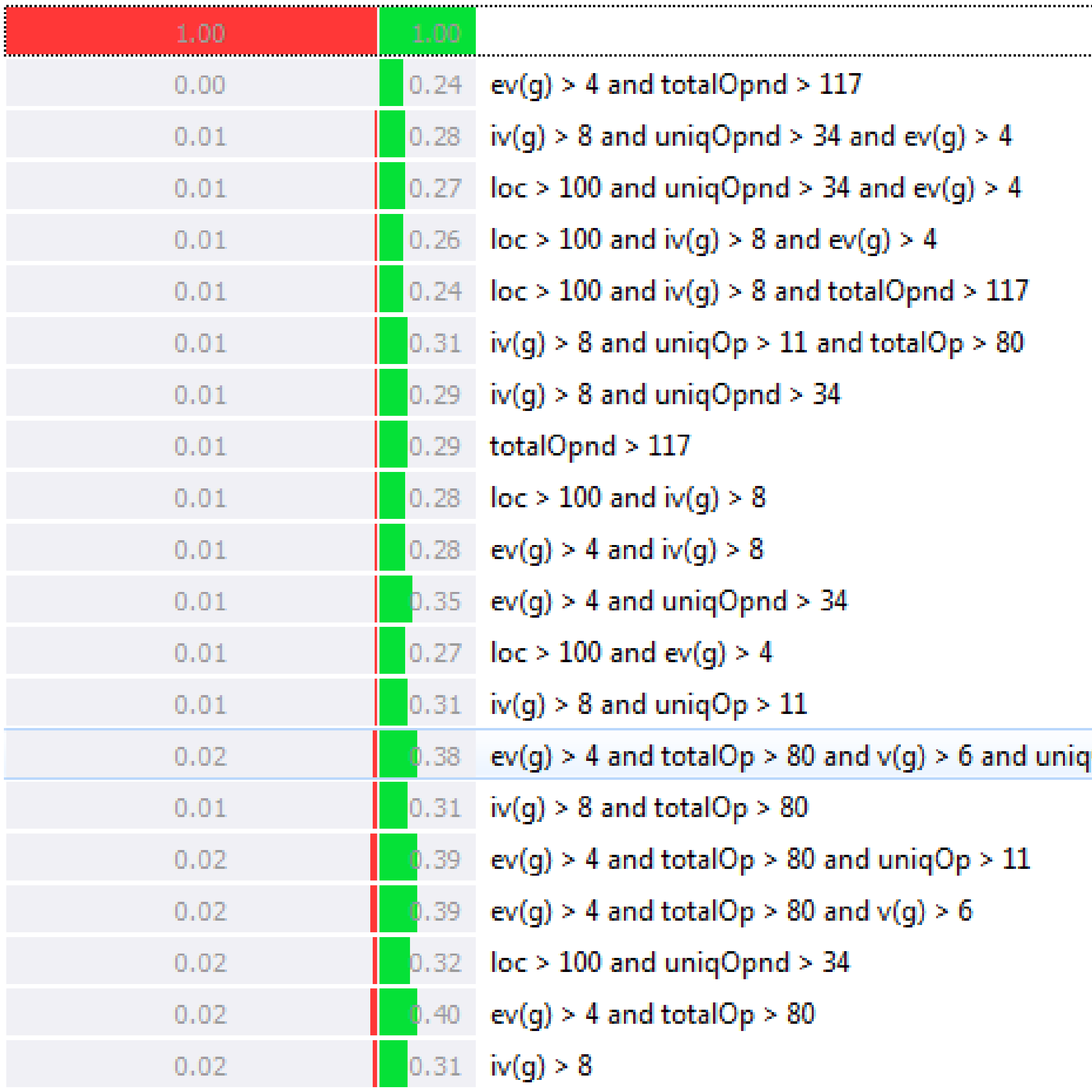


Figure 19.2: SD Gen Vs Spc



1.00	1.00	
0.00	0.24	ev(g) > 4 and totalOpnd > 117
0.01	0.28	iv(g) > 8 and uniqOpnd > 34 and ev(g) > 4
0.01	0.27	loc > 100 and uniqOpnd > 34 and ev(g) > 4
0.01	0.26	loc > 100 and iv(g) > 8 and ev(g) > 4
0.01	0.24	loc > 100 and iv(g) > 8 and totalOpnd > 117
0.01	0.31	iv(g) > 8 and uniqOp > 11 and totalOp > 80
0.01	0.29	iv(g) > 8 and uniqOpnd > 34
0.01	0.29	totalOpnd > 117
0.01	0.28	loc > 100 and iv(g) > 8
0.01	0.28	ev(g) > 4 and iv(g) > 8
0.01	0.35	ev(g) > 4 and uniqOpnd > 34
0.01	0.27	loc > 100 and ev(g) > 4
0.01	0.31	iv(g) > 8 and uniqOp > 11
0.02	0.38	ev(g) > 4 and totalOp > 80 and v(g) > 6 and uniq
0.01	0.31	iv(g) > 8 and totalOp > 80
0.02	0.39	ev(g) > 4 and totalOp > 80 and uniqOp > 11
0.02	0.39	ev(g) > 4 and totalOp > 80 and v(g) > 6
0.02	0.32	loc > 100 and uniqOpnd > 34
0.02	0.40	ev(g) > 4 and totalOp > 80
0.02	0.31	iv(g) > 8

And with the SD algorithm:

Chapter 20

Semi-supervised learning

Semi-Supervised Learning (SSL) lies between supervised and unsupervised techniques, where a small number of instances in a dataset are labeled but a lot of unlabeled data is also available.

- *Supervised all data labelled*
- Semi-supervised *both labelled and unlabelled data*
- *Unsupervised no class attribute (all unlabelled)*

Given a dataset, $\mathcal{D}$:

- $\mathcal{D} = \mathcal{L} \bigcup \mathcal{U}$
- *Learner* $f : \mathcal{X} \mapsto \mathcal{Y}$
- *Labeled data* $\mathcal{L} = (X_i, Y_i) = \{(x_{1:l}, y_{1:l})\}$
- *Unlabeled data* $\mathcal{U} = X_u = \{x_{l+1:n}\}$ *\(avalilable during training, usually $l << n$)*
- *Test data* $X_{test} = \{x_{n+1:...}\}$

In SSL, threre are two distinct goals:

- Inductive. *Predict the labels on future test data, i.e., learning models are applied to future test data (not avalilable during training).* $\{(x_{1:l}, y_{1:l}), x_{l+1:n}, x_{n+1:...}\}$
- Transductive. *It is concerned with predicting the labels on the unlabeled instances provided with the training sample.* $\{(x_{1:l}, y_{1:l}), x_{l+1:n}\}$

Self-training Algorithm

Input: Labeled data $\mathcal{L}$, unlabeled data $\mathcal{U}$, and a supervised learning algorithm $\mathcal{A}$.

1. *Learn a classifier f using labeled data $\mathcal{L}$ with f.*
2. *Label unlabeled data $\mathcal{U}$ with f.*
3. *Add new labeled data to $\mathcal{L}$ and removed them from $\mathcal{U}$*

For each class C, select examples which c labels as C with high confidence, and add it to the labeled data.

Repeat 1–3 until it converges or no more unlabeled example left.

(Deocadez et al., 2017) Preliminary Study on Applying Semi-Supervised Learning to App Store Analysis In Appstore, classifying reviews into: bugs, request *and* other.

In defect prediction:

(Lu et al., 2012) Software defect prediction using semi-supervised learning with dimension reduction

(Li et al., 2012) Sample-based software defect prediction with active and semi-supervised learning

Chapter 21

Learning from Crowds

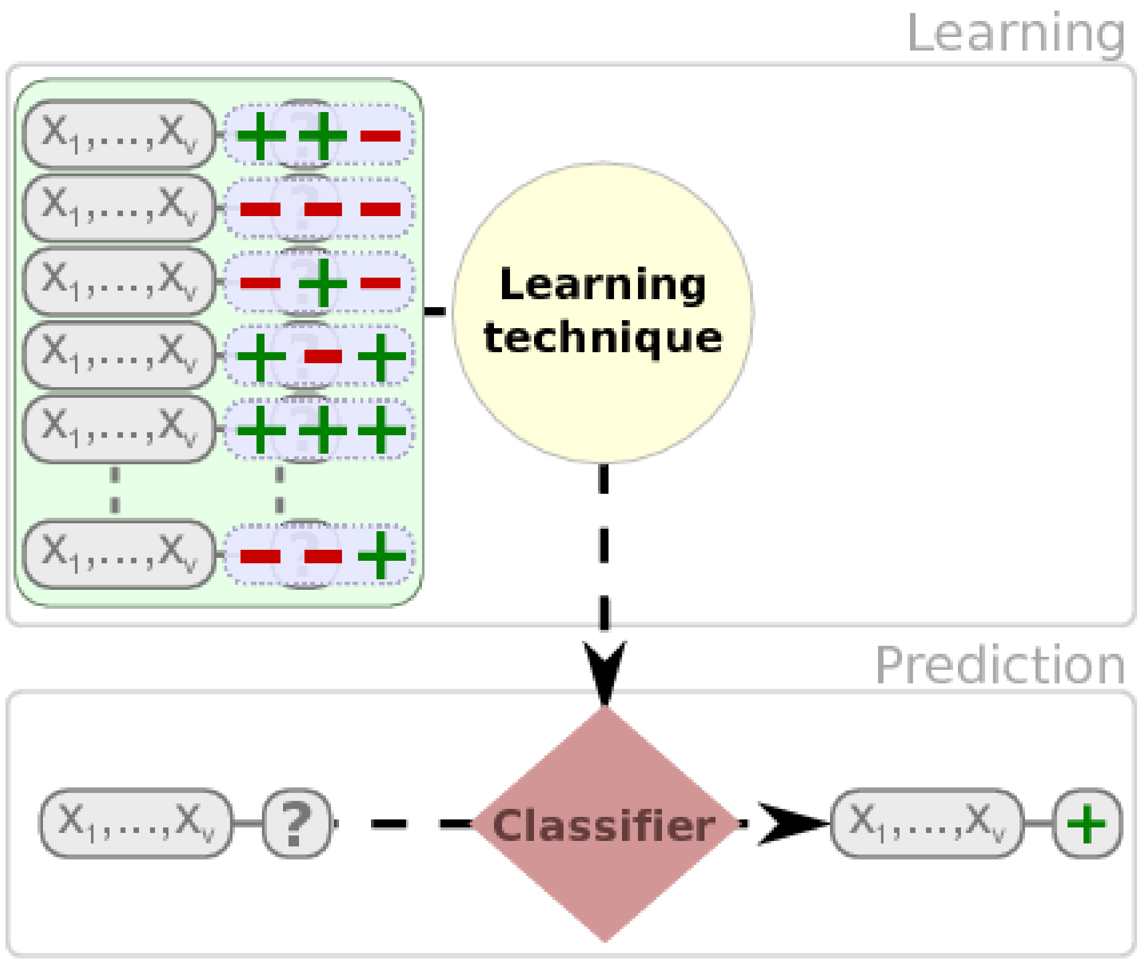


Figure 21.1: Learning from Crowds

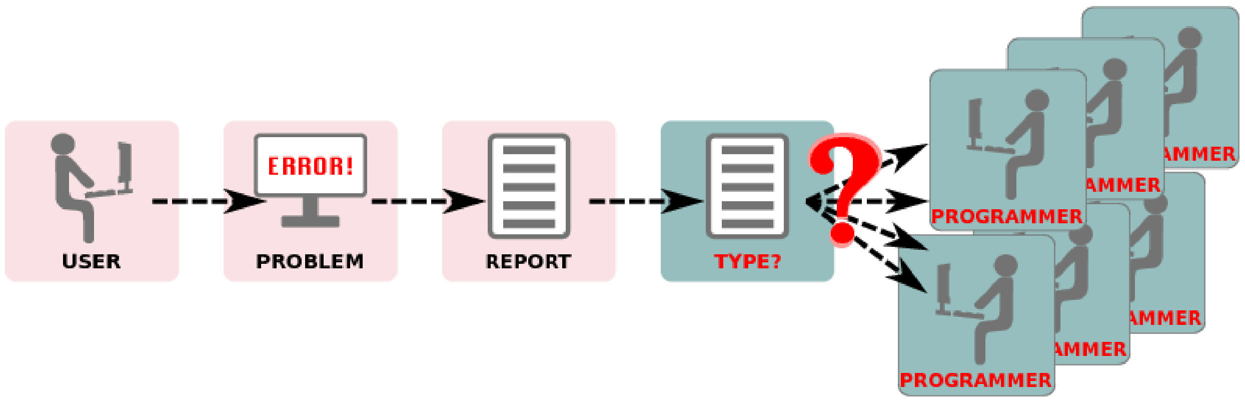


Figure 21.2: ODC Classification

Chapter 22

Multi-objective Rules for defect prediction

We are applying MOQAR (multi-objective evolutionary schemes to discover quantitative association rules) to defect prediciton rules

(Martínez-Ballesteros et al., 2016) Improving a Multi-objective Evolutionary Algorithm to Discover Quantitative Association Rules

Chapter 23

Settings Thresholds for Defect Prediction

23.1 Use of Mean and Standard Deviation

Some of the methods proposed in the literature suggest that the threshold for a given measure X should be based on its distribution.

More specifically, both Erni and Lewerentz [10] and Lanza and Marinescu [19] describe methods to define thresholds based on the mean value and the standard deviation of measures.

As an applicability precondition, both proposals assume that X follows a normal distribution.

According to [10], the interval $[\mu\sigma, \mu + \sigma]$ is regarded as the central range of normal *values for X, where μ and σ are the average and the standard deviation of the distribution of X, respectively.*

The proposal of [19] takes $\mu - \sigma$ as the "low" threshold, $\mu + \sigma$ as the "high" threshold, and $1.5(\mu + \sigma)$ as the "very high"" threshold. In addition, the proposal suggests the use of "meaningful"" thresholds, which can be derived based on

1. *commonly-used fraction thresholds (e.g., 0.75) and*
2. *thresholds with generally-accepted meaning, $\mu - \sigma$ and $\mu + \sigma$.*

23.2 Use of Weighted Benchmark Data

Alves et al. use data from multiple different software systems to derive thresholds that are expected to "(i) bring out the metric's variability between systems and (ii) help focus on a reasonable percentage of the source code volume" (Alves et al., 2010).

23.3 Use of Quantiles

Quantiles can also be used to set thresholds

23.4 Some further literature

(Benlarbi et al., 2000) Thresholds for object-oriented measures

(Morasca and Lavazza, 2016) Slope-based Fault-proneness Thresholds for Software Engineering Measures

Part VII

Bibliography

Bibliography

Alves, T., C.Ypma, and Visser, J. (2010). Deriving metric thresholds from benchmark data. In IEEE International Conference on Software Maintenance (ICSM'2010)*, pages 1–10.*

Arisholm, E., Briand, L. C., and Johannessen, E. B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. Journal of Systems and Software*, 83(1):2–17.*

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering*, 28(1):4–17.*

Benlarbi, S., Emam, K. E., Goel, N., and Rai, S. (2000). Thresholds for object-oriented measures. In Proceedings 11th International Symposium on Software Reliability Engineering (ISSRE 2000)*, pages 24–38.*

Bowes, D., Hall, T., and Petrić, J. (2015). Different classifiers find different defects although with different level of consistency. In Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'15*, PROMISE'15, pages 3:1–3:10, New York, NY, USA. ACM.*

Breiman, L. (1996). Bagging predictors. Machine Learning*, 24:123–140.*

Canfora, G., Lucia, A. D., Penta, M. D., Oliveto, R., Panichella, A., and Panichella, S. (2013). Multi-objective cross-project defect prediction. In IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 252–261.*

Catal, C. (2011). Software fault prediction: A literature review and current trends. Expert Systems with Applications*, 38(4):4626–4636.*

Catal, C. (2012). Software mining and fault prediction. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(5):420–426.*

Catal, C. and Diri, B. (2009). A systematic review of software fault prediction studies. Expert Systems with Applications*, 36(4):7346–7354.*

Chen, Z., Menzies, T., Port, D., and Boehm, D. (2005). Finding the right data for software cost modeling. IEEE Software*, 22(6):38–46.*

Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering*, 20(6):476–493.*

D'Ambros, M., Lanza, M., and Robbes, R. (2010). An extensive comparison of bug prediction approaches. In Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR07)*, pages 31 – 41. IEEE CS Press.*

D'Ambros, M., Lanza, M., and Robbes, R. (2011). Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Software Engineering*, pages 1–47. 10.1007/s10664-011-9173-9.*

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. Journal of Machine Learning Research*, 7:1–30.*

Deocadez, R., Harrison, R., and Rodriguez, D. (2017). Preliminary study on applying semi-supervised learning to app store analysis. In Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE'17)*, EASE'17, pages 320–323, New York, NY, USA. ACM.*

Dolado, J. J., Rodriguez, D., Harman, M., Langdon, W. B., and Sarro, F. (2016). Evaluation of estimation models using the minimum interval of equivalence. Applied Soft Computing*, 49:956–967.*

Fawcett, T. (2006). An introduction to roc analysis. Pattern Recognition Letters*, 27(8):861–874. ROC Analysis in Pattern Recognition.*

Fernández, A., García, S., and Herrera, F. (2011). Addressing the classification with imbalanced data: Open problems and new challenges on class distribution. In 6th International Conference on Hybrid Artificial Intelligence Systems (HAIS)*, pages 1–10.*

Fernández-Delgado, M., Cernadas, E., Barro, S., and Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems? Journal of Machine Learning Research*, 15:3133–3181.*

Freund, Y., Schapire, R., and Abe, N. (1999). A short introduction to boosting. Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612.*

Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2011a). The misuse of the nasa metrics data program data sets for automated software defect prediction. In 15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011)*, pages 96–103.*

Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2011b). The misuse of the nasa metrics data program data sets for automated software defect prediction. In Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, pages 96 –103.*

Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering*, 38(6):1276–1304.*

Halstead, M. (1977). Elements of software science*. Elsevier Computer Science Library. Operating And Programming Systems Series; 2. Elsevier, New York ; Oxford.*

Hand, D. J. (2009). Measuring classifier performance: a coherent alternative to the area under the roc curve. Machine Learning*, 77(1):103–123.*

He, H. and Garcia, E. (2009). Learning from imbalanced data. IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284.*

Herraiz, I., Izquierdo-Cortazar, D., Rivas-Hernandez, F., Gonzalez-Barahona, J. M., Robles, G., nas Dominguez, S. D., Garcia-Campos, C., Gato, J. F., and Tovar, L. (2009). FLOSSMetrics: Free / libre / open source software metrics. In Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society.*

Herrera, F., Carmona del Jesus, C. J., González, P., and del Jesus, M. J. (2011). An overview on subgroup discovery: Foundations and applications. Knowledge and Information Systems*, 29:495–525.*

Hosseini, S., Turhan, B., and Mantyla, M. (2017). A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. Information and Software Technology*, pages –.*

Howison, J., Conklin, M., and Crowston, K. (2006). FLOSSmole: A collaborative repository for FLOSS research data and analyses. International Journal of Information Technology and Web Engineering*, 1(3).*

Huang, L., Ng, V., Persing, I., Geng, R., Bai, X., and Tian, J. (2011). AutoODC: Automated generation of orthogonal defect classifications. In 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'2011)*, pages 412–415.*

Ibarguren, I., Pérez, J., Muguerza, J., Rodriguez, D., and Harrison, R. (2017). The consolidated tree construction algorithm in imbalanced defect prediction datasets. In Evolutionary Methods and Machine Learning in SE, Testing and SE Repositories. Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC2017)*, pages 96–103.*

Jureczko, M. and Spinellis, D. (2010). Using object-oriented design metrics to predict software defects. In Models and Methodology of System Dependability. Proceedings of RELCOMEX 2010 Fifth International Conference on Dependability of Computer Systems DepCoS*, Monographs of System Dependability, pages 69–81, Wrocław, Poland. Oficyna Wydawnicza Politechniki Wrocławskiej.*

Khoshgoftaar, T. and Van Hulse, J. (2009). Empirical case studies in attribute noise detection. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 39(4):379 –388.*

Khoshgoftaar, T. M., Gao, K., and Napolitano, A. (2012a). An empirical study of feature ranking techniques for software quality prediction. International Journal of Software Engineering and Knowledge Engineering*, 22(2):161–183.*

Khoshgoftaar, T. M., Gao, K., and Napolitano, A. (2012b). Exploring an iterative feature selection technique for highly imbalanced data sets. In IEEE 13th International Conference on Information Reuse and Integration (IRI'2012)*, pages 101–108.*

Klösgen, W. (1996). Explora: a multipattern and multistrategy discovery assistant*. American Association for Artificial Intelligence, Menlo Park, CA, USA.*

Krishnan, S., Strasburg, C., Lutz, R. R., and Goševa-Popstojanova, K. (2011). Are change metrics good predictors for an evolving software product line? In Proceedings of the 7th International Conference on Predictive Models in Software Engineering (Promise'11)*, Promise'11, pages 7:1–7:10, New York, NY, USA. ACM.*

Li, M., Zhang, H., Wu, R., and Zhou, Z.-H. (2012). Sample-based software defect prediction with active and semi-supervised learning. Automated Software Engineering*, 19(2):201–230.*

Lincke, R., Lundberg, J., and Löwe, W. (2008). Comparing software metrics tools. In Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*, ISSTA'08, pages 131–142, New York, NY, USA. ACM.*

Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., and Baldi, P. (2009). Sourcerer: mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery*, 18:300–336. 10.1007/s10618-008-0118-x.*

Lu, H., Cukic, B., and Culp, M. (2012). Software defect prediction using semi-supervised learning with dimension reduction. In 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, pages 314–317.*

Madeyski, L. and Jureczko, M. (2015). Which process metrics can significantly improve defect prediction models? an empirical study. Software Quality Journal*, 23(3):393–422.*

Martínez-Ballesteros, M., Troncoso, A., Martínez-Álvarez, F., and Riquelme, J. C. (2016). Improving a multi-objective evolutionary algorithm to discover quantitative association rules. Knowl. Inf. Syst.*, 49(2):481–509.*

McCabe, T. (1976). A complexity measure. IEEE Transactions on Software Engineering*, 2(4):308–320.*

Mende, T. and Koschke, R. (2010). Effort-aware defect prediction models. In Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*, CSMR'10, pages 107–116, Washington, DC, USA. IEEE Computer Society.*

Menzies, T., Dekhtyar, A., Distefano, J., and Greenwald, J. (2007a). Problems with precision: A response to comments on data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering*, 33(9):637–640.*

Menzies, T., Greenwald, J., and Frank, A. (2007b). Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering.

Morasca, S. and Lavazza, L. (2016). Slope-based fault-proneness thresholds for software engineering measures. In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 12:1–12:10, New York, NY, USA. ACM.*

Moser, R., Pedrycz, W., and Succi, G. (2008). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In 2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190.*

Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. (2012). Change bursts as defect predictors. In 21st IEEE International Symposium on Software Reliability Engineering (ISSRE 2012).

Nussbaum, L. and Zacchiroli, S. (2010). The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 52–61.*

Panichella, A., Oliveto, R., and Lucia, A. D. (2014). Cross-project defect prediction models: L'union fait la force. In 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 164–173.*

Raudys, S. and Jain, A. (1991). Small sample size effects in statistical pattern recognition: recommendations for practitioners. IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(3):252–264.*

Rodriguez, D., Ruiz, R., Cuadrado, J., and Aguilar-Ruiz, J. (2007). Detecting fault modules applying feature selection to classifiers. In IEEE International Conference on Information Reuse and Integration (IRI 2007)*, pages 667–672.*

Rodriguez, D., Ruiz, R., Riquelme, J., and Aguilarâ€"Ruiz, J. (2012). Searching for rules to detect defective modules: A subgroup discovery approach. Information Sciences*, 191(0):14–30. <ce:title>Data Mining for Software Trustworthiness</ce:title>.*

Seiffert, C., Khoshgoftaar, T., Hulse, J. V., and Folleco, A. (2007). An empirical study of the classification performance of learners on imbalanced and noisy software quality data. In 2007 IEEE International Conference on Information Reuse and Integration, IEEE IRI-2007*, pages 651–658.*

Shepperd, M., Song, Q., Sun, Z., and Mair, C. (2013). Data quality: Some comments on the nasa software defect datasets. IEEE Transactions on Software Engineering*, 39(9):1208–1215.*

Shippey, T., Hall, T., Counsell, S., and Bowes, D. (2016). So you need more method level datasets for your software defect prediction? voila! In 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'16)*, ESEM'16, pages 12:1–12:6, New York, NY, USA. ACM.*

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., and Noble, J. (2010). Qualitas corpus: A curated collection of java code for empirical studies. In 2010 Asia Pacific Software Engineering Conference (APSEC2010).

Turhan, B. (2012). On the dataset shift problem in software engineering prediction models. Empirical Software Engineering*, 17:62–74. 10.1007/s10664-011-9182-8.*

Van Antwerp, M. and Madey, G. (2008). Advances in the sourceforge research data archive (srda). In Fourth International Conference on Open Source Systems, IFIP 2.13 (WoPDaSD 2008)*, Milan, Italy.*

Vasa, R. (2010). Growth and Change Dynamics in Open Source Software Systems*. PhD thesis, Faculty of Information and Communication Technologies Swinburne University of Technology Melbourne, Australia.*

Wang, H., Khoshgoftaar, T. M., Wald, R., and Napolitano, A. (2012). A comparative study on the stability of software metric selection techniques. In 11th International Conference on Machine Learning and Applications, ICMLA*, pages 301–307.*

Wrobel, S. (1997). An algorithm for multi-relational discovery of subgroups. In Proceedings of the 1st European Symposium on Principles of Data Mining*, pages 78–87.*

Wrobel, S. (2001). Relational Data Mining*, chapter An algorithm for multi-relational discovery of subgroups, pages 74–101. Springer.*

Zhang, H. (2009). An investigation of the relationships between lines of code and defects. In IEEE International Conference on Software Maintenance*, pages 274–283.*

Zhang, H. and Zhang, X. (2007). Comments on "data mining static code attributes to learn defect predictors". IEEE Transactions on Software Engineering*, 33(9):635–637.*

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., and Murphy, B. (2009). Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE'09, pages 91–100, New York, NY, USA. ACM.*

Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In Proceedings of the Third International Workshop on Predictor Models in Software Engineering (PROMISE'07)*, PROMISE '07, pages 9–, Washington, DC, USA. IEEE Computer Society.*