Faculteit Economie en
Bedrijfswetenschappen

Department of Decision Sciences
and Information Management

KU Leuven

Essays on Empirical Software Engineering

Proefschrift voorgedragen tot
het behalen van de graad van
Doctor in de Toegepaste
Economische Wetenschappen

door

Karel Dejaeger

Nummer 407 2012

Committee

Prof. dr. Marleen Willekens (Chair)	KU Leuven
Prof. dr. Bart Baesens (Promotor)	KU Leuven
Prof. dr. Monique Snoeck (Co-promotor)	KU Leuven
Prof. dr. Jan Vanthienen	KU Leuven
Prof. dr. ir. David Martens	Universiteit Antwerpen
Prof. dr. Bojan Cukic	West Virginia University
Dr. Thomas Ostrand	Rutgers University

Daar de proefschriften in de reeks van de Faculteit Economie en Bedrijfswetenschappen het persoonlijk werk zijn van hun auteurs, zijn alleen deze laatsten daarvoor verantwoordelijk.

Dankwoord

Live as if you were to die tomorrow.
Learn as if you were to live forever.
Mahatma Gandhi, 1869 – 1948

Hoewel er maar 1 auteur op de cover van deze doctoraatstekst staat, zijn er een heel aantal personen zonder wie deze tekst nooit zou zijn tot stand gekomen. Ik zou dan ook graag eenieder willen bedanken die me geholpen heeft bij het tot stand brengen van deze tekst. Hierbij gaat bijzondere dank uit naar verschillende personen.

In het voorjaar van 2009, tussen alle andere job aanbiedingen door, ontving ik een mailtje van Bart Baesens, toen nog professor Baesens voor mij. Of ik reeds plannen had voor komend jaar, en indien niet, of een doctoraat onder hem iets voor mij zou zijn. Enkele gesprekken later was ik overtuigd van deze uitdaging, en aldus vatte ik mijn doctoraat aan. De afgelopen 3 jaren hebben me geleerd dat dit een allerbeste keuze was; Bart heb ik leren kennen als een promotor die steeds klaar staat voor zijn medewerkers, die meedenkt waar nodig, en steeds een goede oplossing aanbiedt wanneer er zich een schijnbaar onoverkomelijk probleem stelt. Ook stond hij steeds open voor team building activiteiten, waaronder een voetbalmatch, de zomer barbecues, pizza events en kerstfeestjes. Onze bureaus lagen letterlijk recht tegenover elkaar, en hierdoor moest ik steeds op mijn qui-vive zijn, waarvoor dank!

Ik had het geluk in mijn doctorale commissie ook een aantal andere personen te mogen verwelkomen. Professor Monique Snoeck heb ik leren kennen als iemand die erg begaan is met haar (doctoraats) studenten, en ondanks een drukke agenda als vice-decaan onderwijs, altijd een opening vond om samen te zitten voor onderzoek of thesis begeleiding. Zo had ik onder meer het plezier om samen te kunnen werken op het updaten van het 'slaagkansen model voor economisten in spe' dat (hopelijk) studenten op weg zet naar een vruchtbare carrière aan onze schitterende faculteit. Ook professor Jan Vanthienen heeft voor mij een belangrijke rol gespeeld de afgelopen 3 jaren. Het feit dat zijn deur open stond voor eenieder die vragen had, heb ik sterk geapprecieerd, en de goede samenwerking op vlak van onderzoek, bekroond met een DSS paper, en studie begeleiding (weka sessies) zal ik steeds onthouden. Professor David Martens is een waar genoegen om in je commissie te hebben; daar hij zelf recent nog doctoraatsstudent was, weet hij maar al te goed de mogelijke valkuilen en verlokkingen eigen aan een doctoraatstraject, en was hij steeds bereid dit te delen met anderen. Ook de OR conferentie te Egham, waarbij wij, doctoraatsstudenten, de prijs in de wacht sleepten voor het team met onder andere David en Bart zal steeds een aangename herinnering blijven.

I had also the utmost pleasure to welcome Prof. Bojan Cukic and Dr. Thomas Ostrand to my commission. Both can be regarded as experts in the domain of empirical software engineering, and cannot be thanked enough for their willing support in this endeavor. Special thanks goes out for the hospitable reception at the West Viriginia University, homestead of Prof. Cukic, and Prof. Tim Menzies, editor in chief of some of my articles related to fault prediction.

Het schrijven van een dankwoord is zowel een aangename afsluiter van een succesvol doctoraat, als tevens een risicovolle onderneming aangezien niemand van de collega's, zonder dewelke mijn verblijf in HOG 03.120 nooit zo aangenaam zou geweest zijn, mag vergeten worden. Vooreerst zou ik de ZAP staf willen danken; als student nog Prof. Put en Prof. Lemahieu, staan deze nu te boek als bijzonder fijne, en aangename mensen die ik via mijn opvolging zeker nog zal ontmoeten. Ook de mede doctoraatsstudenten moeten zeker vermeld worden. Een aantal van hen kende ik reeds voor aanvang: Tom, Jochen en Filip. De meest markante persoonlijkheid is zonder twijfel deze laatste; gezeten in het aanpalende bureau, was Filip vaak een klankbord voor niet onderzoeksgerelateerde zaken. De wijncursus die we samen volgden was zeker een hoogtepunt, de discussies over high-level onderzoek een ander. Jochen, ondertussen reeds verhuist naar Australië, was de process miner van dienst die, ondanks zijn handicap/dankzij zijn kracht als Anderlecht supporter, het ook tot in zaalvoetbal team Dejaeger heeft geschopt. Tom tenslotte stond altijd klaar met raad en daad als het ging over vechtsporten, en karate in het bijzonder. Uiteraard zijn er nog een heel aantal andere collega's die mijn doctoraatsjaren gekleurd hebben. Mijn bureau heb ik in de loop der jaren gedeeld met Wouter, die nu het mooie weer maakt bij de Dexia bad bank, en met Helen. Deze laatste is actief in het onderzoeksdomein van data quality, alwaar het bekende *six axis framework* aldus opgeld kan maken. Het heeft even geduurd voor sommige collega's het merkten, maar de vorm van haar buik geeft tegenwoordig onmiskenbaar aan dat er binnenkort een kleine Helen in onze faculteit zal rond hossen. Philippe, onze Antwerpenaar van dienst, verdient ook een speciale vermelding; benoemd tot chef koffiedrager, was hij steevast bereid tot een koffiepauze om eens de gedachten te verzetten. Ook met Seppe en Thomas heb ik steeds goed kunnen samen werken, getuige onze gemeenschappelijke pennenvruchten. Mister ticketmatic Alex moet ik dan weer feliciteren met zijn immer relaxte houding, zelfs ten tijde van IWT deadlines. Als je Jonas kon spotten, bleek ook hij een erg leuke collega te zijn, die zeker over impact factoren en journals steeds iets interessant te zeggen had. Ook Pieter, Gayane, Flavius en Willem (die helaas voortijdig gestopt is) stonden steeds paraat voor een babbeltje, of voor assistentie bij een of ander (LaTeX) probleem. Het woord opvolging is al een aantal malen gevallen doorheen dit dankwoord, en deze is zeker gegarandeerd door de nieuwe collega's die ik reeds heb mogen ontmoeten gedurende september. Aimée, Veronique en Libo, jullie gaan dat zeker super doen!

Anderzijds zijn er ook vanuit de familie Dejaeger reeds inspanningen in deze richting gedaan. Broer Stijn is sinds een tweetal jaren in mijn voetsporen getreden als handelsingenieur in de beleidsinformatica en ik hoop dat hij deze studies met evenveel plezier als mij kan afronden. Hem moet ik specifiek danken om mij te introduceren binnen 'zijn' voetbalwereld, en voor zijn ondersteuning bij mijn academische en trainers carrière. Mijn zus Marian houdt onze eer dan weer hoog binnen de medische wereld, en is echt een zus om trots op te zijn. Zij en haar verloofde Steven zorgden voor de nodige *rotaract* noot de afgelopen 3 jaren. Tenslotte zijn er dan nog mijn ouders en grootouders die ook een vitale

ondersteunende rol hebben gespeeld doorheen mijn gehele academische loopbaan, en dit nog steeds blijven doen. Het zijn dan misschien geen 5 pagina's van dankbetuigingen aan jullie adres, maar ik weet wel zeker: zonder al jullie steun had onderliggend werk zeker nooit het daglicht gezien. Daarom draag ik dit werk dan ook graag op aan mijn beide ouders.

Karel, November 2012

Contents

Samenvatting

Data is overal om ons heen. Supermarkten houden bij welke artikelen door wie wanneer zijn gekocht, een bankkaart betaling geeft aanleiding tot een nieuwe observatie in de database van je favoriete financiële instelling, en ook de details van ieder GSM gesprek worden opgeslagen door telecommunicatie operatoren. Door de toenemende informatisering kan verwacht worden dat deze vloedgolf van data niet snel zal opdrogen. Data captatie is echter maar een eerste stap, en het gehele proces van data captatie en verrijking, samen met de extractie van verborgen patronen uit deze data en bijhorende validatie wordt aangeduid met de term Knowledge Discovery in Databases (KDD). Machine learning en data mining zijn twee andere termen die vaak worden gehanteerd in dit opzicht en refereren naar de gereedschapskist die de onderzoeker tot zijn beschikking heeft in de zoektocht naar verborgen patronen in de data. In dit doctoraat zullen we dieper ingaan op de rol die machine learning kan spelen binnen software engineering, waarbij we ook steeds oog hebben voor de andere stappen binnen het KDD proces. Software engineering is op zijn beurt het onderzoeksdomein dat focust op het opleveren van software artefacten en alle aspecten die hiermee gerelateerd zijn. Het is op het kruispunt van deze twee zeer actieve onderzoeksdomeinen dat we dit doctoraat kunnen positioneren.

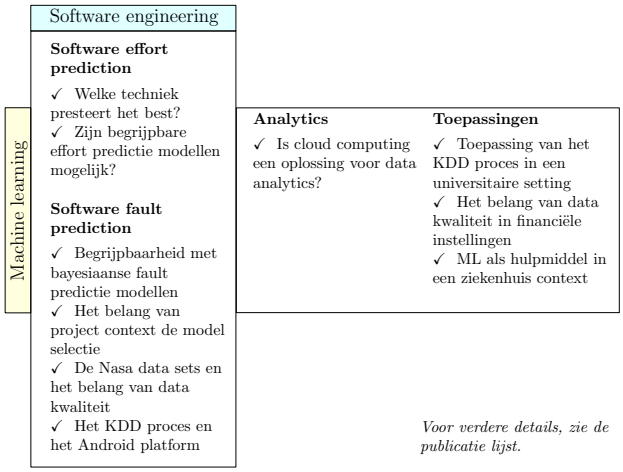
Motivatie

Er is een boutade uit 1986 die gaat als volgt:

> 'Bridges are normally built on-time, on-budget and do not fall down.
> On the other hand, software never comes in on-time or on-budget.
> In addition, it always breaks down.'

Helaas zit er vandaag nog steeds een kern van waarheid in deze uitspraak, zoals internationaal onderzoek ook herhaaldelijk heeft aangewezen. In 1994 stelde een onderzoeksrapport van de Standish Group dat commerciële software projecten gemiddeld met 189% over budget gaan. Meer recente schattingen van deze organisatie uit 2008 geven aan dat de kosten gemiddeld 54% hoger uitvallen dan gebudgetteerd, terwijl 24% van alle software projecten zelfs volledig falen. Ook dichter bij huis vinden we voorbeelden van projecten die grote vertragingen en budget overschrijdingen opliepen zoals het project 'Speer' van het Nederlandse leger, of zelfs projecten die voortijdig gestopt werden, zoals 'Feniks' dat de Belgische justitie het informatica tijdperk moest binnenloodsen. Deze vaststelling wordt als uitgangspunt genomen voor onderliggende tekst, waarin we het gebruik van machine learning onderzoeken ter ondersteuning van het software ontwikkelproces. Specifiek gaan we in op twee deeldomeinen van software engineering: software effort estimation en software fault prediction.

Overzicht van de onderzoeksvragen



Het inschatten van de benodigde ontwikkelingstijd

Een eerste luik van dit doctoraat behelst het inschatten van de benodigde ontwikkeltijd van nieuwe software projecten, zie ook hoofdstuk 3. Als we de eerder geciteerde cijfers in acht nemen, kunnen we stellen dat dit aspect zeker nog onze aandacht verdiend. Een correcte inschatting maken van de ontwikkeltijd is immers cruciaal voor de planning en budgettering van projecten. Historisch gezien is het steunen op de mening van één of meerdere experts de oudste en nog steeds vaak gehanteerde strategie bij het maken van zulke inschattingen. Er zijn echter een aantal duidelijke nadelen hieraan verbonden, zoals de neiging van mensen om extreme gevallen als norm te beschouwen, en de problemen die mogelijks ontstaan indien de expert het bedrijf verlaat. Als reactie hierop zijn er een aantal alternatieven voorgesteld die op meer objectieve wijze informatie van reeds voltooide projecten in rekening nemen, zoals het cocomo model dat op basis van de hoeveelheid te ontwikkelen code en andere projectkarakteristieken een schatting maakt. Andere methoden steunen op het gebruik van functiepunten, een concept dat de hoeveelheid gevraagde 'functionaliteit' in een software project kwantificeert. Recentelijk werd ook het gebruik van machine learning onderzocht, maar tot op heden was het onduidelijk in hoeverre dit een toegevoegde waarde kon bieden aan het gamma van reeds bestaande methoden. In dit eerste luik onderzoeken we dan ook of, en zo ja, welke machine learning techniek de meest geschikte is voor het schatten van de ontwikkelinspanning, en dit op een voorheen ongeziene schaal. Ook de begrijpbaarheid van modellen is een aspect dat onze aandacht verdiend aangezien deze modellen als input dienen tot budgetteringsprocessen en aldus een directe impact kunnen hebben op de gepercipieerde prestaties van ontwikkelaars. Dit werd ook in meer detail besproken in een conferentie publicatie die echter, wegens plaatsgebrek, niet werd opgenomen in deze doctorale tekst.

Het screenen op fouten

Software testing is een activiteit die tot 60% van het totale ontwikkelbudget kan kosten, en verwaarlozing hiervan kan leiden tot grote, onverwachte uitgaven na het opleveren van het project. Het screenen van de broncode om zo de meer foutengevoelige gebieden te identificeren kan een belangrijke bijdrage leveren tot het reduceren van deze uitgaven. Het tweede luik van dit doctoraat (hoofdstukken 4, 5 en 6) benadert dit probleem door opnieuw gebruik te maken van diverse machine learning technieken. Er dient immers opgemerkt te worden dat bijvoorbeeld expertpanels vaak geen goed zicht hebben op de volledige broncode van grotere projecten en dat het bovendien als te tijdsintensief wordt ervaren om op voldoende fijne granulariteit deze code te screenen. Hierdoor kan machine learning als het belangrijkste hulpmiddel gezien worden binnen software fault prediction. In een eerste deel onderzoeken we of meer begrijpbare bayesiaanse modellen van waarde kunnen zijn, en houden we ook rekening met de context waarin een project wordt ontwikkeld. Binnen dit onderzoeksdomein wordt er vaak gebruik gemaakt van data die gecollecteerd werd door de NASA. In een tweede deel beschouwen we van nabij deze data en sporen we mogelijke kwaliteitsproblemen op die aanwezig zijn in deze data. Tenslotte beschouwen we een voorbeeld van het volledige KDD proces binnen software engineering. Meer bepaald rapporteren we de resultaten van een case studie van het Android platform waarin we onder andere onderzoeken of het mogelijk is om data overheen meerdere releases te gebruiken voor predictie doeleinden.

Besluit

Veel van de aspecten die behandeld zijn in dit doctoraat kunnen gezien worden als symptomen van het feit dat software ontwikkeling nog steeds een grotendeels mensgedreven proces is, en hoewel er pogingen ondernomen worden om de menselijke factor deels uit te schakelen (zie bijvoorbeeld het gebruik van code generatoren), is het onwaarschijnlijk dat dit zal lukken op korte termijn. Tot het zover is, zullen de onderwerpen van dit doctoraat hun waarde behouden, en wellicht zelfs aan belang winnen door de steeds toenemende informatisering van onze samenleving. Echter, zoals Professor Tim Menzies me ook vertelde op mijn bezoek naar Morgan Town, ‘the search for a theory which can summarize everything in software engineering is unlikely to be fruitful; instead, we should focus on specific aspects which can make a difference’.

Bridges are normally built on-time,
on-budget and do not fall down.
On the other hand, software never
comes in on-time or on-budget.
In addition, it always breaks down.

Alfred Spector, 1986

1 Introduction

1.1 Context and problem definition

Alongside with the advent of the first programmable electronic computing devices in the early forties, the first software was developed. The first such machine was the Colossus Mark I which was developed in the UK to decode encrypted messages during WW II. Being very complex and expensive devices, these computing devices or computers were typically purpose-built machines requiring several skilled engineers to operate and maintain the computer[1]. The past 65 years have witnessed the evolution from purpose-specific, room-filling machines to portable and versatile computers, also coinciding with an enormous growth of the software development industry. In order to put the importance of the software development industry into perspective, the following numbers should be considered. Gartner, an information technology research and advisory firm, estimated that the worldwide enterprize software development market netted $ 176.3 billion in 2008 while in 2010, the total revenue already accounted for $ 244 billion. For 2011, Gartner forecasts a further increase of 9.5% for a total of $ 267 billion dollar [116]. Moreover, market researcher Datamonitor concluded that the global worldwide software industry valued $ 303.8 billion in 2008. Over time, the software development sector has also witnessed several large mergers and acquisitions (M&A's), strengthening the position of large developers. Table 1.1 provides data on a number of large M&A's, listing the estimated value of the target company in the last column. The institute of Mergers, Acquisitions and Alliances, a specialized research institution, acknowledged over 40,000 M&A's over the past 22 years with a total known value of close to $ 1,500 billion[2]. Another example is the Initial Public Offering of the social networking company Facebook Inc. on the 18^{th} of May 2012, with a peak market capitalization of over $ 104 billion. Furthermore, as software development remains to a large extend a creative process which is only partially automatable, a large number of people are employed in this sector. The US Bureau of Labor Statis-

[1] Note that the first programmable computing devices predates the Colossus Mark I, being perceived by Charles Babbage in the 19^{th} century. However, these machines were mechanical in nature and were never actually built due to funding problems [333].

[2] www.imaa-institute.org

Buyer	Target	M&A completion date	Total worth
Microsoft	Hotmail	31/12/1997	$ 500 million
Microsoft	Visio Corporation	1/7/2000	$ 1.375 billion
Ebay	Paypal	14/10/2002	$ 1.5 billion
Oracle	Peoplesoft	13/12/2004	$ 10.3 billion
Symantec	Veritas	16/12/2004	$ 13.5 billion
Yahoo	Flickr	20/9/2007	$ 1.6 billion
Activision	Vivendi Games	9/7/2008	$ 18.8 billion
Oracle	Sun microsystems	20/4/2009	$ 7.4 billion

Table 1.1: Examples of large M&A's in software / ICT industry

tics stated that a total of 333,620 persons were employed in the US as computer programmer in the period of May 2010[3] while the National Bureau of Statistics of China estimated that a total of 1,150,000 students graduated from Chinese technical schools in 2009 [207]. Research indicated that the occupation of software engineer is amongst the fastest growing occupations in the United States [135].

It should be acknowledged that software engineering in fact covers a broad range of activities and could be defined as 'the systematic application of scientific and technological knowledge, through the medium of sound engineering principles, to the production of computer programs, and to the requirements definition, functional specification, design description, program implementation, and test methods that lead up to this code' [194][4]. Empirical software engineering can be regarded as a subdomain hereof in which quantitative data are collected, processed and analyzed with the purpose of supporting the development of software. Software developing companies need to take several resources into account such as programmers' time, computer equipment and office space. Typically, considering the tremendous increase in storage capacity and computing power, the latter two are subordinate, requiring project management to focus on the planning and support of software developers. As a response hereto, various research tracks aiming at assisting project management and individual developers have been investigated, including software effort prediction, software fault prediction, bug pattern mining and reliability modeling. In this dissertation, the focus lies on two important topics in the field of empirical software engineering: software effort prediction and software fault prediction.

The first considers software projects as a whole, defining project-wide characteristics to come up with an upfront estimate of the total required software

[3]www.bls.gov, Standard Occupational Classification (SOC) code 151131.

[4]The term *software engineering* was first used in academic literature during the NATO software engineering conferences held in 1968 in Garmish, Germany and in 1969 in Brussels, Belgium.

development time, typically expressed in terms of man months or an equivalent metric. Several estimation approaches have been developed and the investigation of these methods is often referred to as software effort or software cost prediction research.

The second topic, software fault prediction, zooms in on the actual source code being developed during the project to identify fault prone code to guide testing efforts. By looking at source code characteristics (e.g. static code features) and taking other information such as developer and/or code churn metrics into account, mathematical models identifying fault prone modules can be constructed.

In the following sections, both topics are first positioned in a typical project development life cycle. Then, the paradigm of open source software development is elaborated upon and its relevance to both topics is discussed. Next, the topic of software effort prediction is further introduced followed by software fault prediction. The specific research questions are also presented at the end of each section. This chapter concludes with an overview of the results published in the context of this dissertation.

1.2 Software development models

In order to keep larger software projects manageable, a development methodology can be followed. A prevalent methodology is the waterfall model proposed by W. Royce in 1970 [270]. The waterfall model is a development methodology which starts by collecting and analyzing the project requirements, followed by a design and a coding phase. Sometimes, the waterfall model constitutes an additional preliminary design phase in which the entire process is explored in miniature to verify critical design areas. The waterfall model also acknowledges the importance of a (separate) testing step and the need for post deployment maintenance activities. These last two steps are acknowledged to be a major expense in the overall development budget. The waterfall model has often been criticized as being too inflexible and too heavy weight, mandating the completion of each phase before commencing the next. For instance, it is assumed that during requirement analysis, all desired user functionalities are to be captured; i.e. the set of requirements is considered to be frozen. In many cases however, upon receiving a functional prototype, the end user requirements turn out to be not fully understood or are subjected to change. Moreover, design choices made early in the project might cause implementation issues later during development. While Royce already acknowledged that software design is seldom confined to the successive steps outlined by his development model, a number of alternative development methodologies have been proposed, including Rational Unified Process (RUP) [191], Spiral development [37] and eXtreme Programming (XP) [28]. However, the waterfall model is still often adopted in larger projects, as is suggested by recent data collection efforts [148]. The waterfall model is shown in Fig. 1.1 and the topics discussed in this dissertation are positioned within this model. It can be seen that effort estimation is typically performed at the

beginning of the project, after completing the requirements phase. The results of the requirement phase can serve as an input to estimate the project size and complexity using for instance a functional sizing based approach which are subsequently used as inputs for software effort modeling [5]. Fault prediction is typically effected in between the coding and testing phase. However, e.g. during the coding phase, the output of fault prediction models can already be used to indicate fault prone regions in the source code, while estimating development effort can be delayed and/or repeated later during project development. Note that irrespective of the development methodology, an upfront estimation of the development effort is often a managerial requirement for staff allocation and productivity benchmarking with other projects and/or other software developers [71]. Software testing is also a crucial, but sometimes neglected activity as often no direct added value is generated. Testing can take up to 60% of the total development cost and in some cases, e.g. for mission critical software, this percentage is reported to be even higher [130, 270].

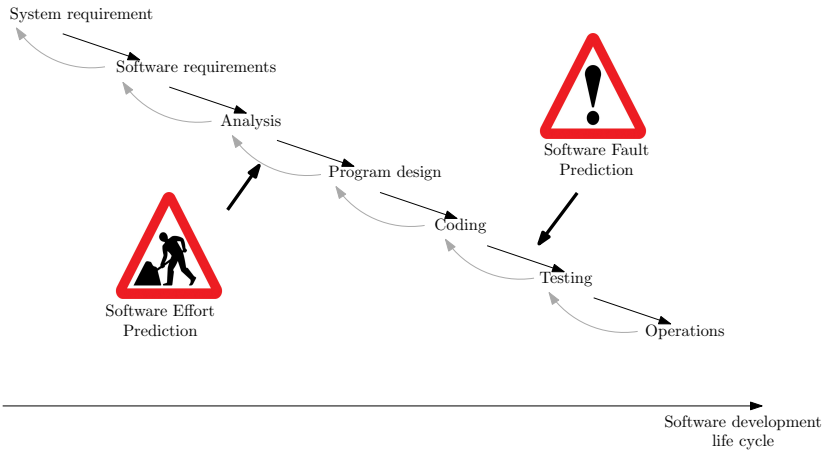


Figure 1.1: Positioning of both research topics in the waterfall model

1.2.1 Open source software

A paradigm which is gaining momentum in the software development industry is that of open source software (OSS) in which the source code is made publicly available free of charge (and consequently, commercial software is sometimes referred to as closed source software or CSS). Depending on the software licence, one is allowed to copy, modify and redistribute the source code for (non) commercial purposes. The open source initiative (OSI)[5] lists a large number of OSS licenses, the GNU general public licence (GPL) being one of the most frequently used. This particular licence was issued by a second organization, the

[5] www.opensource.org

free software foundation[6] which, similar to the OSI, supports the development of OSS and GNU projects in particular. The GPL states that anybody may adapt or even resell source code, on the premise that others are granted the same rights and all contributors are mentioned. Other licences might impose additional restrictions.

Provider	# Projects	# Developers
Berlios	4.750	50.512
www.berlios.de		
sample hosted projects:	*SIM-IM, FreeNX*	
Eclipse Foundation	*Unknown*	*Unknown*
www.eclipse.org		
sample hosted projects:	*Eclipse, Mylyn*	
github	2.550.801	927.117
www.github.com		
sample hosted projects:	*jQuery, reddit*	
GNU Savannah	3.363	52.969
www.savannah.gnu.org		
sample hosted projects:	*emacs, GNU octave*	
Javaforge	540	27.000
www.javaforge.com		
sample hosted projects:	*KETTLE, OpenModelSphere*	
SourceForge	303.839	*Unknown*
www.sourceforge.net		
sample hosted projects:	*Emule, Azureus, FileZilla*	
Tigris.org	692	*Unknown*
www.tigris.org		
sample hosted projects:	*Scarab, Subversion, ArgoUML*	

Table 1.2: Overview of OSS hosting sites

Typically, OSS is developed by geographically distributed teams of volunteers who do not receive any direct compensation. As such, projects often lack any form of central project management and violate also other well established software development practices such as the development of software in small teams, the reenforcement of specific development guidelines, or the need for requirement analyses [131, 307]. It should be noted that *the* open source software development model does not exist. One of the first discussions on the differences between the development of various OSS projects can be found in the work of E. Raymond, who compares the development of Linux with the development

[6] www.fsf.org

practices of other open source projects. He states that the development of some open source projects resembles the building of a cathedral. In these projects, there is a small and collaborating group of programmers who develop software without releasing any beta version to the public. These programmers lean more towards how commercial software is built, adhering for instance to a waterfall development methodology using tight organizational structures and centralized planning. On the other hand, the Linux project accepts contributions from anybody, releasing all modifications and versions to the public. This sort of development is compared to a bazaar where numerous developers with different agendas and approaches are cooperating with each other [266]. The open source community offers a number of collaboration platforms which can be used to host software projects. Table 1.2 lists a number of such OSS platforms. Note that larger OSS projects often do not use these platforms but instead are hosted on separate websites dedicated to the project. Examples hereof are the GCC compiler for the GNU project, the Andoid platform which is supported by the Android Open Source Project, the development of Linux and both the FreeBSD and OpenBSD desktop environments.

Especially the development of 'bazaar-like' open source projects is often bug driven, in the sense that users make requests or locate bugs which are then used as input during development. Fig. 1.2 provides an overview of the different actors in this process. First, a user posts a bug or feature request on a forum or directly on the appropriate bug tracker (possibly via a link provided in the application) (**1**). The user has to provide details such as whether it is a feature request or a bug [32]. As this dissertation focuses on fault prediction, only bugs are considered. Typical bug trackers in use by open source projects are Bugzilla, Issuezilla, Trac, MantisBT and Scarab. As users might file reports on bugs already present in the bugtracker, e.g. Bugzilla incorporates an automatic duplicate bug detection process, presenting a list of possible matches upon submission [259]. Once the bug report is filed, a senior developer will assign it to other developers, depending on their area of expertise; this is also referred to as bug triage (**2**) [11, 66]. The assignee will consult the information available on the tracker and possibly consult other developers through mailing lists or other means of communication (**3**). Often, the latest version of the source code can be obtained from a content management system (**4**). Such source code repositories together with bug tracking services are typically offered by OSS platforms. Possible content management systems include CVS, SVN, GIT and Mercurial. After locating the appropriate source code and checking out this code, modifications can be made. These modifications need to be reviewed and the developer will often release an initial version of the modifications (**5**). These modifications will be further scrutinized by other developers, possibly making use of mailing lists or the bug tracking service (**6**). Finally, the update is incorporated into the application and the bug request is closed (**7**) [166].

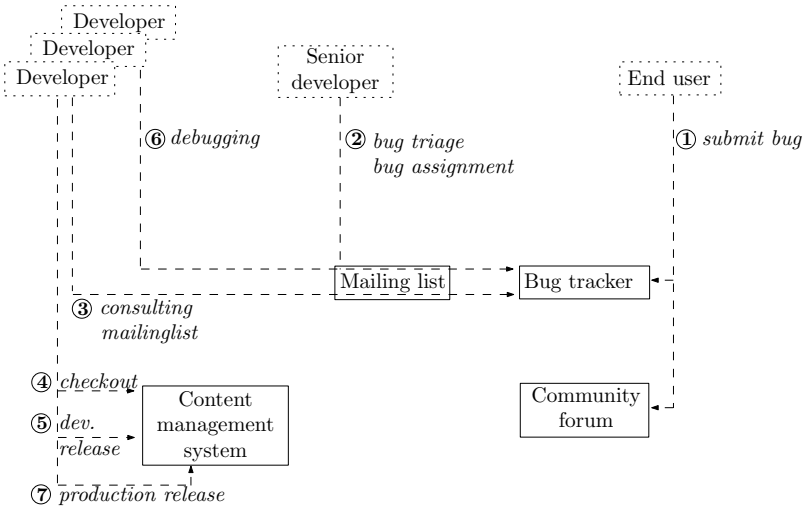


Figure 1.2: Open source software development

1.3 Software effort prediction

In this section, the field of software effort prediction is further elaborated upon by first providing an overview of the key academic research groups in this domain followed by a more in depth look into how software effort prediction is done. Finally, the research objectives are presented.

1.3.1 Research landscape

As the issue of planning and budgeting software projects is often a very challenging one, it has attracted research interest from various domains including software engineering, managerial sciences [1], and cognitive sciences [249]. Due to the large number of publications from various viewpoints, a more systematic and focussed search through the literature seems required; in this thesis, we will focus on how to predict development effort using mathematical techniques, putting less emphasis on related topics such as software size estimation (e.g. function point analysis [2, 5]) or expert driven estimation techniques [160].

To provide an overview of the software effort prediction literature, a query was entered into several academic publication repositories. Academic papers are stored and indexed to allow other researchers to more easily search for relevant papers. Note that the ease of which papers can be located and retrieved has inspired researchers to write comprehensive literature review papers [52, 54, 121, 160, 163, 330]. These papers are often amongst the most valuable papers to other researchers as they discuss the current state of the art in a specific topic. Depending on the domain, typical indexing sites include Science Direct, CiteSeer, ISI Web of Knowledge, PubMed and Scopus. In this introductory

chapter, two such indexes have been queried, i.e. Science Direct[7] and ISI Web of Knowledge[8], and the information retrieved from these indexes was parsed into a MySQL database. The following query has been submitted to both Science Direct and ISI Web of Knowledge: '(SOFTWARE EFFORT *or* SOFTWARE COST) *and* (ESTIMATION *or* PREDICTION)'. Table 1.3 provides the number of journal papers and conference publications that were outputted by both indexes on 22/08/2011. Note that Science Direct only contains information on Elsevier journal publications and thus contained no conference publications. Some assumptions were made while parsing the publications and storing them into the database. Each publication is assumed to have a unique title; if a publication with exactly the same title already exists in the database, no new entry was created. Since conference publications are sometimes extended to journal publications which are typically regarded as containing more established results, the loading sequence of the database was adjusted accordingly by first collecting journal publications and adding conference publications in a second step. A second assumption relates to the author names. Some publications were found not to list the complete author names. Instead, the first name was occasionally abbreviated by the first letter. It was assumed that if the second name and the first letter of the first name match, these two authors are the same person. For instance, K. Dejaeger is assumed to be the same person as *Karel* Dejaeger or *Koen* Dejaeger. Note that in the literature, a number of more advanced algorithms for name matching have been proposed, taking context or other meta data into account [124]. However, both indexes do not provide such additional meta data and preliminary analysis indicated double names to occur only very sporadically, relegating this issue. Furthermore, additional measures which will be explained later, have been taken to further mitigate this issue. It should also be noted that occasionally, publications did not list any authors. These publications, typically technical reports or editorials, were not loaded into the database. These aspects explain the difference between the total number of publications in the database and the sum of the number of papers returned by both indexes.

An important question that remains is the extent to which the publications of the query are indeed related to the topic at hand. More specifically, what are the precision (i.e. the fraction of publications that are related to the topic, software effort prediction) and recall (i.e. the fraction of relevant publications that is correctly identified) of the query. The first would require to locate and read all publications present in the data set, which would seem infeasible given the total number of identified publications (cfr Table 1.3). To partially verify the precision, a number of summary statistics were considered, such as publication venue and publication date. These are detailed later throughout the text. On the other hand, checking the recall would require some independent set of publications, preferably systematically hand collected which can then be matched to the publications present in the database. While this also might seem

[7] www.sciencedirect.com
[8] www.apps.webofknowledge.com

Topic	Papers	Conferences	Authors
Software effort prediction	1196	1077	3545
Science Direct	*60*	*N/A*	
ISI Web of Knowledge	*1152*	*1371*	
Software fault prediction	657	936	2582
Science Direct	*40*	*N/A*	
ISI Web of Knowledge	*681*	*1160*	

Table 1.3: Overview of the collected information

infeasible at first, we can make use of the set of publications identified in prior literature reviews. In case of software effort prediction, the systematic literature overview of Jørgensen and Shepperd can be adopted to this purpose [163][9]. In this study, the authors performed a systematic literature review which led to the identification of 304 journal publications (conference publications were not included in the scope of the paper). As the complete list is provided in appendix of their paper, this set of publications can be matched to the database. A total of 177 papers (an effective recall of 58.2%) were identified by the query and successfully loaded into the database. It is important to note that the inclusion criteria used in the study of Jørgensen and Shepperd were slightly different in that 'papers describing research on software development effort of cost estimation' were selected and thus also included e.g. managerial papers on development effort which are less relevant to our particular setting of using data mining techniques to software effort prediction.

Table 1.4 provides an overview of the top 10 journals publishing the most papers in the field of software effort prediction together with their 5-Year Impact Factor. It can be observed that 3 of the most important journals are published by Elsevier with 'Information and Software Technology' containing most publications. Table 1.5 conveys similar information on the conferences in the field of software effort prediction while Fig. 1.3 provides the evolution of the number of publications (both journal publications and conference proceedings) over time. These summary statistics indicate a sufficient precision in the data set as the number of publications in non-computer related venues was found to be limited. It can be concluded that the lack of definitive answer on how to best estimate the development effort of a software project has indeed inspired much research and resulted in an increasing number of publications in the domain of effort prediction over the last years.

Each publication is written by one or more authors who collaborate with each other. Considering this information allows to construct so-called academic social networks [302,303]. These networks can be geographically bound (e.g. only include researchers from the same university or country) but are often found to

[9]Note that the authors of this literature review provide an updated version of the list of relevant papers at www.simula.no/BESTweb. This updated list was however not publicly available at the time of writing.

Label	Journal	Impact Factor
Inf&SfTech	Information and Software Technology	1.433
	Proportion: 6.77 %, cumulative: 6.77 %	
IEEETSE	IEEE Transactions on Software Engineering	3.468
	Proportion: 6.10 %, cumulative: 12.88 %	
JSys&Sf	Journal of Systems and Software	1.282
	Proportion: 5.69 %, cumulative: 18.56 %	
IEEESf	IEEE Software	1.899
	Proportion: 4.60 %, cumulative: 23.16 %	
EmpSfEng	Empirical Software Engineering	1.795
	Proportion: 2.51 %, cumulative: 25.67 %	
SfQualJ	Software Quality Journal	0.813
	Proportion: 2.26 %, cumulative: 27.93 %	
ESWA	Expert Systems With Applications	2.193
	Proportion: 1.17 %, cumulative: 29.10 %	
AmProg	American Programmer	N/A
	Proportion: 1.17 %, cumulative: 30.27 %	
CommACM	Communications of the ACM	2.487
	Proportion: 1.09 %, cumulative: 31.35 %	
SfEngNt	Software Engineering Notes	N/A
	Proportion: 1.09 %, cumulative: 32.44 %	

Table 1.4: Journal overview: software effort prediction

Label	Venue	Count
METRIC	International Symposium on Software Metrics	60
	Proportion: 7.57 %, cumulative: 7.57 %	
ICSE	International Conference on Software Engineering	59
	Proportion: 7.19 %, cumulative: 14.76 %	
ESEM	International Symposium on Empirical Software Engineering and Measurement	20
	Proportion: 2.52 %, cumulative: 17.28 %	

Table 1.5: Conference overview: software effort prediction

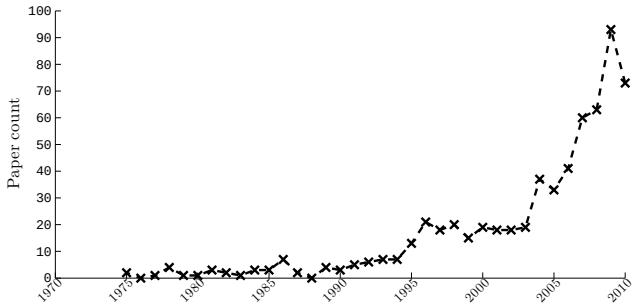


Figure 1.3: Number of publications per year on software effort prediction

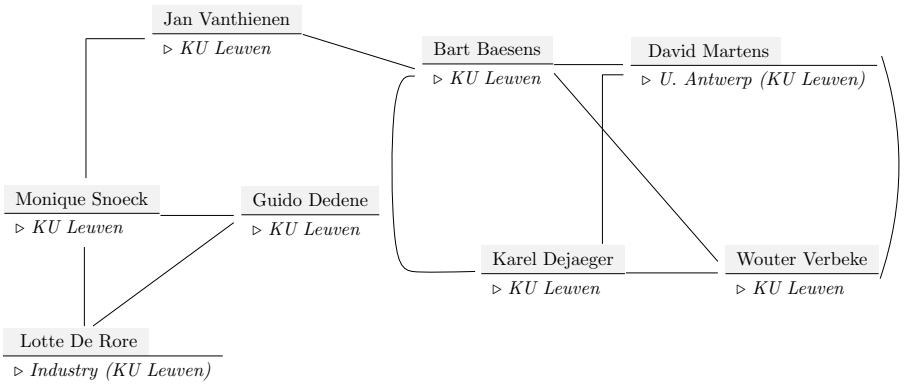


Figure 1.4: The K.U.Leuven academic network in Empirical Software Engineering

be more related to specific research topics. A typical metric of successful collaboration is the number of publications and the impact of these publications (e.g. the number of citations). As the number of citations is also dependent on the publication age and since it was available for only a fraction of all publications, only the total number of publications is used to denote the success of collaboration and thus the strength of the ties in the academic social networks. As an example of such network, the academic network of the Department of Decision Sciences and Information Management at the KU Leuven is provided in Fig. 1.4. The current affiliation of each person is listed in italic script (previous affiliations between brackets).

Fig. 1.5 displays the most important (in terms of total number of publications) academic collaboration network in the domain of software effort prediction. The number of publications of each researcher found in the database is provided between square brackets. It should be noted that only ties with a strength of two and up are taken into account during network construction to make them more robust to minor errors introduced during the parsing or to the inclusion of irrelevant papers that happen to match the query. Note that academic collaboration networks have for instance already been studied by Thang et al. [302] and implemented in the Arnetminer tool[10]. To the best of our knowledge, academic social networks have not been explored in the domain of empirical software engineering (i.e. software effort and software fault prediction), and further investigation herein could be a valuable addition to a literature overview paper. The use of these networks allows to identify the key researchers who made significant contributions to a certain topic, including the number of joint publications which is captured by the weight of the edges in the network. Moreover, there exists a class of specific metrics, called social network metrics, to further quantify the importance of individual nodes (researchers) with respect to the network [328]. One such typical social network measure

[10]www.arnetminer.org

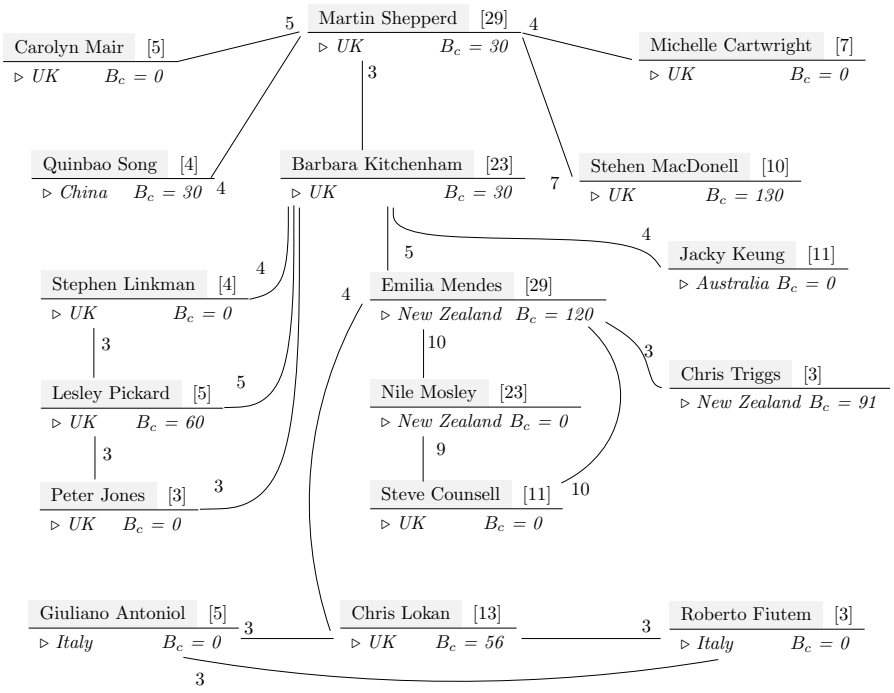


Figure 1.5: Software effort collaboration network

is the betweenness centrality of a node, B_c, which is defined as the number of shortest paths between any two other nodes that pass through a particular node. This metric thus can be used to quantify the degree to which one researcher mediates between others, and thus is a proxy to ones importance in the network, together with the number of publications. The betweenness centrality of each researcher is provided in italic script.

Note that in calculating the betweenness centrality, all edges in a network have been taken into account; this includes edges with a weight of one or two, which are not displayed in both figures[11]. The distance between any two connected authors is given by

$$d_{ij} = \frac{1}{c_{ij}} \tag{1.1}$$

where c_{ij} equals the number of joint publications of author i and j.

1.3.2 Methods for software effort prediction

Software projects are recognized to often exceed initial budgets and/or experience delays or even cancelations. In this respect, the findings of the Chaos

[11]The betweenness centrality has been calculated making use of the BGL toolbox in Matlab, www.mathworks.com/matlabcentral/fileexchange/10922.

Chaos reports				
Year	Successful	Challenged	Failed	*Avg. cost overrun*
1994	16 %	53 %	31 %	*189 %*
1996	27 %	33 %	40 %	*142 %*
1998	26 %	46 %	28 %	*69 %*
2000	28 %	49 %	23 %	*45 %*
2002	34 %	51 %	15 %	*43 %*
2004	29 %	53 %	18 %	*56 %*
2006	35 %	46 %	19 %	*47 %*
2008	32 %	44 %	24 %	*54 %*
Other studies				
Study		Reference	# Projects	*Avg. cost overrun*
Jenkins et al., 1984		[151]	72	*36 %*
Phan et al., 1988		[256]	191	*33 %*
Bergeron et al., 1992		[31]	89	*33 %*
Moløkken-Østvold et al., 2004		[238]	52	*41 %*

Table 1.6: Overview of costs overruns in software development

1994 report are often cited, which claim that 'a staggering 31.1% of projects will be canceled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates' [304]. More recently however, the validity of these results have been questioned since the Standish group, the authors of the 1994 and other bi-annual Chaos reports, refused to provide transparency in how these results were obtained [90,162]. On the other hand, it should be noted that also other studies found significant cost overruns of 30 to 40% on average. An overview of studies including the bi-annual Chaos reports is given in Table 1.6.

In response to the lack of accurate software effort estimations, a range of techniques and models have been proposed, which can be further subdivided into three main categories: expert driven estimation methods, formal model based estimation, and data mining oriented approaches [180]. Note that also other taxonomies have been proposed to structure the corpus on software effort estimation, see e.g. Kocaguneli et al. for a discussion hereon [182].

Expert driven estimation

The oldest and still frequently used approach for software development effort estimation is expert driven estimation, in which one or more experts, based on their experience, will come up with an estimation of the development effort required to complete a specific project. Possible variations hereon include e.g. delphi based prediction methods, in which multiple experts each provide an independent estimate as a first step. Afterwards, all estimates are returned to the experts, based on which they can revise their initial suggestion until an agreement on the total development effort has been reached. The applicability of

expert based judgement (also referred to as *clinical judgement*) versus statistical methods (or *actuarial judgement*) has also been investigated in e.g. medical settings. In such settings, actuarial methods were found to be preferable as they are consistent (i.e. the same set of inputs always returns the same result). It was also found that experts are often inclined to assume worst-case scenarios and that the use of actuarial methods will ensure variables contribute to predictions based on their actual relationship with the criterion of interest [68]. It should be noted that actuarial methods can also often be automated thus possibly saving time and expenses, even if the accuracy is comparable with clinical methods.

The software planning task is however dissimilar to medical problems since e.g. the number of observations to learn from is typically low and there are often unquantifiable factors affecting software development. As a result, software effort prediction is sometimes found to benefit from expert opinion and a number of researchers have taken an interest into this topic [160, 161, 165]. One important researcher is Jørgensen, who mainly focussed on expert driven estimation approaches. As the overlap with other software effort prediction approaches is limited, this researcher was found to be only weakly connected with other researchers (i.e. M. Shepperd and E. Mendes) in the academic network shown in Fig. 1.5 and since network connections of insufficient strength were discarded during network building, this researcher instilled a separate academic network (not shown).

Formal models

A second approach to software effort prediction is the use of formal models. Several such models have been developed, including Cocomo (COnstructive COst MOdel) [35, 39], SLIM (Software LIfecycle Management) [261] and FPA (Function Point Analysis) [5]. These models consider the size of the software project and a number of predetermined parameters in some preset formulaic form to estimate the development effort. Arguably the most commonly used formal model is Cocomo, developed by B. Boehm in 1981. This model considers lines of code (LOC) as a proxy for size (or more correctly: delivered source instructions, which exclude comment lines) and a number of other markup factors to estimate effort, depending on the selected model. The basic Cocomo model relates size directly to effort, disregarding markup factors, while the intermediate and advanced models include these additional factors. The original data set on which the COCOMO81 model was calibrated is publicly available in the Promise repository[12]. More recently, Cocomo II was introduced to account for new trends such as the increased complexity of software projects and the reuse of code. In fact, Cocomo II again constitutes of two models, referred to as the early design model and the post architecture model respectively. The first can be used during software design to explore various alternatives while the latter is more detailed and will be used for the actual effort estimation task. The

[12]The Promise repository is a public data repository founded in 2005 which is aimed at providing publicly available data sets to the research community to facilitate empirical software engineering research. www.promisedata.org

Cocomo II post architecture model is defined as:

$$\text{effort} = a \times \text{size}^e \times \prod_{i=1}^{17} EM_i \tag{1.2}$$

in which a takes on a fixed value and EM_i are the effort multipliers that quantify specific project properties. The exponent e in this equation is determined by 5 additional project characteristics which are called scale factors since these have a non linear impact on the estimated development effort. A definition of each of the factors in the Cocomo II model can be found in [71].

SLIM, developed by L. Putnam in 1978, is an alternative to Cocomo which again takes LOC as a proxy for project size and then modifies this through the use of a Rayleigh curve model. SLIM is a proprietary model which is commercially exploited by QSM (Quantitative Software Management Inc.), a company founded by the inventor of SLIM[13]. Being an active company for many years, this company collected a considerable data set which is however not accessible to researchers.

In the literature, several issues regarding LOC as a size measure have been described. These issues include the lack of an universal definition of LOC, the fact that LOC count is implementation dependent and its dependency on the coding style of a developer [159]. In response to these remarks, more end user oriented size measures were put forward, most notably function points. A function point can be regarded as a quantification of the amount of 'function' that a particular piece of code should perform. The idea of function points originated from the work of A. Albrecht at IBM and his method entails the counting of different elementary functions [5]. A distinction is made between 5 such functions being external inputs, external outputs, external queries, internal logical files and external file interfaces. Referring to the functional user requirements, a count of the different elementary functions is made. Next, each function is classified as simple, average or complex and, depending on this classification, is given a larger weight towards the Unadjusted Function Point (UFP) count. The adjusted (or final) function point count is then defined as

$$\begin{aligned} &FP = UFP \times VAF \\ &\text{with } VAF = 0.65 + 0.01 \times \sum_{i=1}^{14} F_i \ . \end{aligned} \tag{1.3}$$

F_i is the set of project characteristics that influence the function point count and its definition can be found in [5]. This way of counting function points is often referred to as the IFPUG method to the similarly named organization which was founded in 1986. IFPUG (International Function Point User Group) is a not for profit organization which resides under the ISBSG umbrella and promotes the effective management of software development and maintenance

[13] www.qsm.com

activities through the use of Function Point Analysis[14]. The ISBSG (International Software Benchmarking Standards Group) in turn is a not for profit initiative which also hosts the well known ISBSG data set[15]. This data set is often used by companies for benchmarking purposes; note that also researchers are granted access to this data set. The data is collected using standardized questionnaires and contains data from a very diverse set of companies. The ISBSG has 11 participating IT and metric organizations with the (international) IFPUG organization being the most important one. The others are nationally focussed organizations from countries such as The Netherlands (NESMA), Finland (FISMA) and Germany (DASMA). The IFPUG function point counting method, together with other function point counting methods such as the Mark II method proposed by C. Symons, are sometimes referred to as the first generation of function points.

A new, second generation functional size measurement was proposed in 1999 by the COmmon Software Measurement International Consortium (COSMIC) to address several shortcomings in the IFPUG approach. For example, the distinction between simple, average and complex proved to be too rigid and IFPUG function points were found to be not suitable to quantify the size of real time projects. Moreover, the concept of 'logical file' underpinning the IFPUG function point method is outdated. Despite several improvements, COSMIC function points are seldom used in practice. It was noted by C. Symons that probably 'less than 1 % of all IT organizations use any type of functional size measure. Of those who do, I would say that 99 % of them are probably using the IFPUG method' [300]. Note that both the IFPUG and the COSMIC functional size measurement are ISO/IEC standards.

The academic network centered around B. Boehm was found to be one of the most important in terms of number of publications. However, since the focus of this work does not lie on the formal modeling approach, this network is not included in the text. While academia have done interesting research into these models, it was reckoned that their focus has recently shifted towards the use of data mining techniques for software effort prediction [163].

Data mining

A third approach to software effort prediction is the use of data mining techniques to create a mathematical model from a set of historic observations. Each historic observation represents a previously completed project with a set of known project characteristics. Typically, these include the project size (often expressed in LOC, FP or an equivalent measure), information concerning the development environment (i.e. concerning the team or company), project data such as the purpose and type of the project and development related variables expressing managerial and/or technical aspects of the software project. This information is aggregated into a matrix $\mathbf{X} \in \mathbb{R}^{N \times n}$. N is used as the number of

[14] www.ifpug.org
[15] www.isbsg.org

observations throughout this text while n represents the number of characteristics or attributes. The time to complete the project is also recorded and this is aggregated into a data set $\{(\mathbf{X}, e)\}$ where the target variable e denotes the project completion time (in hours or an equivalent measure). Based hereon, a data mining model predicting the effort of an unseen project i can be seen as providing a mapping from the (known) project characteristics to the estimated effort, $f(x_i) : \mathbb{R}^n \mapsto \hat{e}_i$.

Data mining techniques have been investigated in the context of software effort prediction since the early 90's [44, 296] and have gained widespread academic attention during the last few years. A large and diverse set of techniques have been proposed in this context and since development effort can take any positive value, especially regression techniques have received attention. The often recurring use of analogy based learners (also known as case based reasoning or lazy learning) which construct no explicit model deserves a special mention. Instead, these learners select the set of most similar projects from the data and based on their development effort, a prediction is generated. This approach somewhat resembles the way in which experts will draw on their previous experience to form predictions [205, 244, 283]. Note that some work also discussed the use of classification techniques after discretising the effort [275].

Data sets in the domain of software effort prediction seem to share a number of characteristics which pose difficulties to data mining techniques. More specifically, data in this domain is typically difficult to collect resulting in smaller data sets. Collecting sufficient data thus represents a considerable startup cost for companies eager to explore data mining techniques. Moreover, the data are typically found to be skewed since larger projects are often underrepresented in these data sets. Thirdly, as software development remains a human centric endeavor, a large number of intangible factors can come into play which are difficult to capture by data mining techniques (e.g. disagreement amongst developers or volatile end user requirements). Despite the academic interest into these issues, the question which (data mining) technique is most applicable remains an open issue.

Since the focus in this thesis lies on this third approach to software effort prediction, the academic network shown in Fig. 1.5 focusses on researchers interested herein. Although both metrics, the number of publications and the betweenness centrality, not always agree on who are the key researchers in a network, a number of interesting observations can be made. Focussing on this academic network, it can be remarked that researchers from different countries (China, UK, New Zealand) are present, indicating the importance of such networks. Further investigation also indicated the importance of conferences as academic meeting points, as these researchers typically had a considerable number of such publications. The researchers in this network have performed research on various domains; for instance J. Keung looked into analogy based learners, proposing a framework called Analogy-X while B. Kitchenham did research on the metrics which are used to assess the performance of prediction techniques and on software bidding models together with the research group of L. Pickard. On the other hand, E. Mendes leads an initiative called the Tukutuku Bench-

marking Project, which aims at gathering effort data from web developers to build cost models geared towards this type of companies[16]. M. Shepperd, among other topics, has focussed on the impact of the number of observations and how to deal with missing data elements. Overall, it can be concluded that their joint research interest lies in applying data mining to software effort prediction. It should be noted that also other researchers have worked on these and related topics, and more details hereon can be found in Chapter 3.

1.3.3 Research objectives

As indicated earlier, a myriad of techniques has been adopted to the topic of software effort prediction. While data mining techniques have recently become more widely adopted, there is still no definite answer as to which data mining technique would be most suitable [75, 163]. In Chapter 3, a large number of techniques are compared to each other on a selection of software effort data sets in order to provide an answer to this first question. Furthermore, this chapter also looks into the aspect of feature subset selection, speculating that more concise models can be obtained without incurring a performance penalty.

A closely related theme is that of model comprehensibility; depending on the context, data mining models are required to provide a varying degree of insight into the underlying relationships within the data. This seems especially true for software effort estimation, as the outcome of these models serves as input to budgeting and remuneration decisions. While this aspect was contemplated upon in a study relating to regression rule extraction, see [278], we decided against incorporating this part into this dissertation for reasons of brevity.

[16] www.metriq.biz/tukutuku/index.html

1.4 Software fault prediction

The following section further elaborates on the field of software fault prediction and is structured along the lines of the previous section by first providing an academic overview of this domain followed by a more in depth discussion on software fault prediction. The section is concluded by presenting the research objectives for this topic.

1.4.1 Research landscape

Crucial to efficient software development is the introduction of a testing phase to timely detect and correct faults since corrective maintenance costs typically increase exponentially when faults are detected later during the development life cycle [38]. An important observation is that faults tend to cluster; i.e. are contained in a limited set of software modules [286]. Thus, testing efforts can be made more efficient by upfront detection of fault prone code, no longer requiring the whole code base to be tested. This finding motivated research into the characteristics that discriminate between fault prone and non fault prone modules [52, 54]. Data mining techniques have often been employed to this end, although software failure has also been studied from various other viewpoints including the use of stochastic models to estimate post-deployment software reliability [112] and other fault identification approaches such as the mining of fault patterns [67]. Important hereto is the timing aspect: as costs incurred to correct faults tend to increase exponentially over time, fault identification and correction should take place before releasing the software. Other topics such as post-deployment reliability modeling assume project completion. In this dissertation, the focus lies on predicting the location of faults using data mining techniques; to distinguish with other fault identification approaches, this topic is further referred to as software fault prediction.

To get an overview of the domain of software fault prediction, a search query similar to that of software effort prediction has been submitted to both Science Direct and ISI Web of Knowledge indexes and information concerning the publications returned by this query have been entered into a separate database, cfr. Table 1.3. The following query was adopted hereto: '(SOFTWARE FAULT *or* SOFTWARE DEFECT *or* SOFTWARE QUALITY) *and* (ESTIMATION *or* PREDICTION)'.

In order to assess the recall and precision of this query, the approach described earlier in Section 1.3.1 was adopted. In order to judge the first, a recent literature review by Catal and Diri was identified [54]. This study included publications on software fault prediction and software quality prediction, but excluded those that did not provided any experimental results. As such, they identified 27 journal publications and 47 conference publications, a total of 74 academic publications. It was found that 42 out of the 74 publications were included in the database, an effective recall of 57%. Note that this recall rate is comparable to that obtained on the topic of software effort prediction. The precision is again verified indirectly, by calculating a set of summary statistics

Label	Journal	Impact Factor
JSys&Sf	Journal of Systems and Software	1.282
Proportion: 7.31 %, cumulative: 7.31 %		
IEEETSE	IEEE Transactions on Software Engineering	3.468
Proportion: 6.85 %, cumulative: 14.16 %		
Inf&SfTech	Information and Software Technology	1.433
Proportion: 5.02 %, cumulative: 19.18 %		
SfQualJ	Software Quality Journal	0.813
Proportion: 4.26 %, cumulative: 23.44 %		
IEEESf	IEEE Software	1.899
Proportion: 3.81 %, cumulative: 27.25 %		
EmpSfEng	Empirical Software Engineering	1.795
Proportion: 3.50 %, cumulative: 30.75 %		
IEEETRel	IEEE Transactions on Reliability	1.698
Proportion: 2.89 %, cumulative: 33.64 %		
IntJSfEngKEng	International Journal of Software Engineering and Knowledge Engineering	0.313
Proportion: 1.98 %, cumulative: 35.62 %		
ESWA	Expert Systems With Applications	2.193
Proportion: 1.83 %, cumulative: 37.44 %		
ASfEng	Annals of Software Engineering	N/A
Proportion: 1.52 %, cumulative: 38.96 %		

Table 1.7: Journal overview: software fault prediction

similar to those obtained in Section 1.3.1. Table 1.7 provides an overview of the 10 journals containing the most software fault prediction related publications together with their 5-Year Impact Factor. It can be seen that both IEEE and Elsevier publish three journals from this list with the Elsevier journal 'Journal of Systems and Software' ranked first. Table 1.8 conveys similar information on the conferences in the field of software fault prediction while Fig. 1.6 illustrates the evolution of the number of publications (both journal publications and conference proceedings) over time. Conclusions similar to that of Section 1.3.1 regarding the precision can be drawn, as the summary statistics e.g. indicate that the number of publications in non-computer related venues is limited. The evolution in the number of publications is indicative to the increasing interest into this topic. One peculiarity however showing in both Fig. 1.3 (Number of publications per year on software effort prediction) as in Fig. 1.6 is that a sharp decrease can be observed in the year 2010. Since this occurs in both cases, it is believed this can be attributed to the indexes themselves as not all 2010 publications have already been included.

1.4.2 Methods for software fault identification

Unlike in the case of software effort prediction, data collection can be automated and intangible factors are typically less important. To the best of our knowledge, only a single study discussed the application of expert based fault prediction and concluded that 'when it comes to comparing both methods we found that

Label	Journal	Count
ISSRE	International Symposium on Software Reliability Engineering	37
	Proportion: 7.91 %, cumulative: 7.91 %	
ICSM	International Conference on Software Maintenance	9
	Proportion: 3.53 %, cumulative: 11.43 %	
METRIC	International Software Metrics Symposium	7
	Proportion: 2.56 %, cumulative: 14.00 %	

Table 1.8: Conference overview: software fault prediction

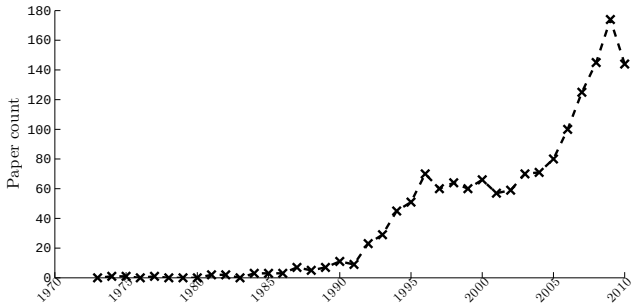


Figure 1.6: Number of publications per year on software fault prediction

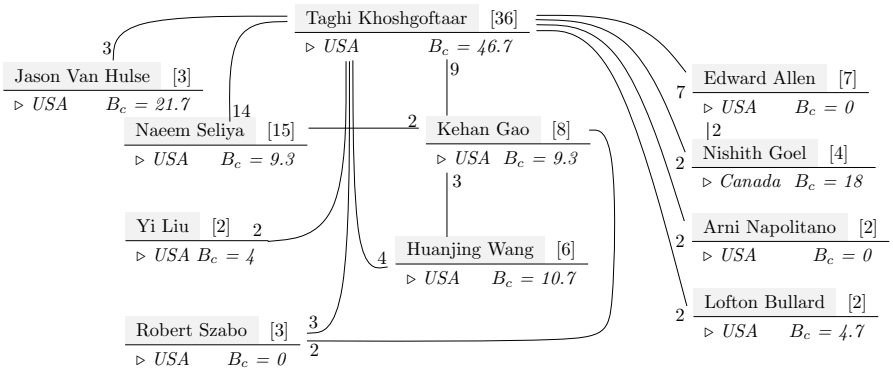


Figure 1.7: Software fault collaboration network, number 1

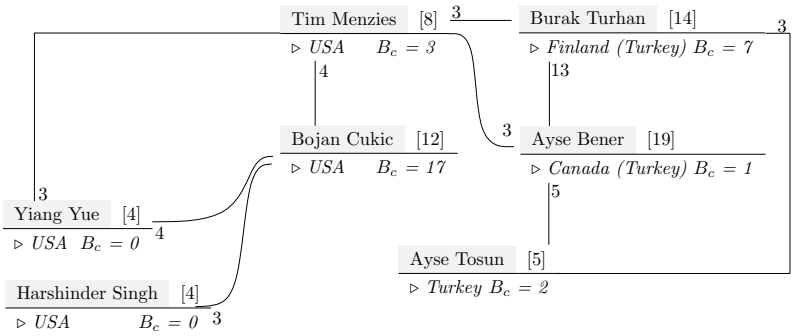


Figure 1.8: Software fault collaboration network, number 2

statistical models outperformed expert estimations' [309]. Other papers have focussed on data mining and alternative approaches that can be automated. In this thesis, the focus lies on the application of data mining (i.e. software fault prediction).

Alternative approaches

Trying to identify fault prone regions in the source code, a number of alternative approaches have been proposed. The following paragraph serves to provide a non exhaustive overview of these approaches. These can be regarded as a alternative or a complement to software fault prediction.

A first such approach, termed 'specification inference', noted that software specifications are often incomplete or missing and aims to derive a more complete set of specifications from source code or program execution. Further analysis of these specifications can reveal faults, which are then linked to regions in the source code. E.g. Yang et al. illustrated such an approach in which they focussed on temporal properties (e.g. acquiring and releasing locks) and induced a set of rules hereon to indicate the presence of bugs [337].

Others mined the source code for implicit programming rules which, if violated, can indicate the presence of an error. Li et al. for example used frequent item set mining to derive sample rules from the source code [206]. An example of such rule could be that if the command 'AskCache' is used, it should be followed by the command 'ReleaseCache'. An alternative approach which compares the Abstract Syntax Tree of a program before and after introducing a fix to derive typical fault patterns is described in Dallmeier et al. [67]. The authors also introduced IBugs, a technique to extract source code snapshots of bugs and their code fixes and using this technique, they constructed a data set from the AspectJ and Rhino source code repository that could serve as a benchmark for similar approaches. This data set is publicly available[17].

[17] www.thomas-zimmermann.com/research/mining-bug-databases

Software fault prediction

Contrary to the alternative approaches discussed earlier, the data mining techniques underpinning software fault prediction research consider characteristics of code segments to predict whether a segment is fault prone or not. Possible characteristics include LOC counts, complexity metrics and code churn metrics and this information can be represented as a matrix $\mathbf{X} \in \mathbb{R}^{N \times n}$ with N the number of code segments and n the number of characteristics. These characteristics or attributes can be extracted by using static code analysis tools such as Prest [183], McCabeIQ[18] or Emerald [144]. Other attributes such as code churn metrics can directly be derived from content management systems. For each code segment it is also noted whether a previous release contained one or more faults. This information is often contained in a separate repository called the bug tracker, as detailed in Section 1.2.1. As entries in the bug tracker can also refer to modification requests or functionality additions, a matching between the actual bugs in the bug tracker and code changes in the source code repository is required [20]. Note that some modern software management systems integrate both into a single platform implying that changes to the source code can only be made if an accompanying entry in the bug tracker exists[19]. Let y_i be a dichotomous target which indicates whether a code segment is faulty ($y_i = 1$) or error free ($y_i = 0$). Using data mining techniques, a mathematical model can then be derived that provides a mapping from the (known) code segment characteristics to an estimate of the fault proneness of that code segment, $f(x_i) : \mathbb{R}^n \mapsto P(\hat{y}_i = 1|x_i)$. Models that provide such mapping to a dichotomous target are often referred to as classification models. Note that some work also regarded the software fault prediction task as a regression problem, predicting the actual number of faults per code segment [251].

Data mining models can be induced on different levels of granularity; i.e. the code segments can be for instance files, classes or methods, depending on data availability. Clearly, a finer granularity would be preferred to guide testing efforts more efficiently but requires increased data collection efforts and thus a trade off should be made. Depending on the granularity, different code characteristics can be used; for instance, LOC counts are independent of the granularity while the Chidamber-Kemerer metric suite can only be used in case of classes. Other well known complexity metrics such as McCabe cyclomatic complexity metrics and Halstead numbers are often defined on the granularity of methods but can also be calculated or otherwise aggregated to other granularity levels.

More recently, it was argued that future research into software fault prediction should change its focus from designing better modeling algorithms towards improving the information content and / or investigation into the model evaluation functions to take the context of software development into account [157, 233]. For instance, Jiang et al. used requirement metrics such as the number of weak phrases and imperatives associated with a source code segment,

[18] www.mccabeiq.com

[19] Examples of such integrated platforms are Fossil and Veracity.

which can e.g. be obtained by text parsing of requirement specification documents [155]. A number of other researchers investigated the use of networks and social network metrics to augment software fault prediction. Zimmermann et al. considered software dependency networks and compared social network metrics derived from these networks with static code metrics. They concluded that the inclusion of software dependency networks improved predictive performance [343]. Also Turhan et al. considered a network approach, using the PageRank algorithm to determine the importance of each software module. This importance was subsequently used to rescale the attribute values of individual software modules, and the result was used as input to a Naive Bayes learner. They found their approach to reduce false alarm rates [316].

It also remains unclear how software fault prediction models should be evaluated. Typical measures include accuracy, recall, precision and F1 measure, which is the harmonic mean of recall and precision. A property shared amongst these measures is that they only take a single operating point of a classifier into account, which can be suboptimal as data sets are often skewed (i.e. a minority of erroneous modules). As a response hereto, a number of other measures have been introduced into the domain of software fault prediction such as the area under receiver operating curve (AUROC) and the recently introduced H-measure [125]. Also evaluation techniques more geared towards the specificities of the fault prediction task have been introduced; examples hereof are the introduction of the notion of effort awareness in model evaluation [226], the use of cost curve analysis [156] and the WHICH meta-learner framework that can be customized to optimize specific goals [231].

Another question relates to the empirical validation setup; often some learner is applied to one part of the data, using the remaining data for performance assessment. These results are subsequently extrapolated towards the future without supporting evidence. Typical procedures here include holdout splitting [200], x-fold cross validation [4, 70] and leave-one-out cross validation. However, it has been argued that when data on multiple releases is available, one could revert to a *cross release* validation setup, accounting for the ordering in releases when defining training and test set [173, 251, 342]. Moreover, the recent introduction of the notion of concept drift in software fault prediction by Ekanayake et al. [87] is underscoring the appropriateness of this approach, as they concluded that the quality of defect prediction approaches varies over time. Other work discussed the feasibility of *cross project* validation, training and testing models on data stemming from different projects [134, 317]; the results of cross project studies paint however a more mixed picture, see e.g. Zimmermann et al. [344].

From the academic networks built for the topic of software fault prediction, the two most important in terms of papers are shown in Fig. 1.7 and Fig. 1.8. The first network is centered around Prof. dr. Taghi Khosghoftaar, affiliated with the Florida Atlantic University, who has done research in several areas including software fault prediction, data imputation techniques and efficient attribute filtering and selection methods. He collaborates with many others active in software fault prediction, including Naeem Seliya and Kehan Gao.

The second academic software fault prediction network include 7 researchers, which can be subdivided into a USA and a Turkish branch. The person with the most publications in this network is Burak Turhan; it should however be noted that a researcher like Tim Menzies arguably has been equably important to this domain as he is one of the founders of the Promise repository initiative mentioned in Section 1.3.2. He also e.g. investigated the NASA MDP repository data sets, which are now in the public domain. Prof. dr. Turhan is collaborating with Prof. dr. Menzies since 2008 and this cooperation has resulted in a number of valuable publications on out of sample validation (i.e. cross-company) and model evaluation.

1.4.3 Research objectives

While a large number of different data mining techniques have been investigated in the context of software fault prediction, considerable attention has been given to one particular class of techniques: Bayesian Network (BN) classifiers. For instance, it was noted by Menzies that the gain of advanced techniques to a simple technique such as Naive Bayes is limited [230]. Furthermore, it should be noted that BN classifiers typically learn comprehensible models [189]. While the Naive Bayes classifier is often selected, other BN learners which allow to construct more flexible networks are less well investigated. In a first chapter relating to software fault prediction, Chapter 4, these BN classifiers are explored; it is hypothesized that these other classifiers are not outperformed by Naive Bayes while allowing for smaller and more comprehensible models. Also an BN inspired feature selection approach is studied in this chapter.

A second chapter on software fault prediction finds its motivation in the recent work of Gray et al., which signaled several data quality issues with the NASA data sets which could potentially invalidate earlier findings [117]. As these data sets have been frequently used for fault prediction, the importance of thoroughly investigating these issues and quantifying their impact cannot be underestimated. A first aspect which is investigated is the many versions of the NASA data sets that seem to be in circulation. Furthermore, Gray et al. outlined a second problematic issue, namely the large number of repeated data points, resulting in the repetition of training observations in test and validation sets. It has been argued that when tuning a learner towards the specific characteristics of a data set using a validation set containing duplicates, learners are prone to over-fitting [187]. Duplicates in the test set on the other hand can result in overly optimistic performance estimates [334]. These issues, together with an update to the well established benchmarking framework of Lessmann et al. allowing for a better statistical discrimination of machine learning algorithms, are presented in Chapter 5.

Current and future research objectives

As a first future research objective, the different evaluation metrics currently in use in this domain will be further studied. As was noted by Menzies, and

by researchers in other domains [201], data mining models tackling real-world business problems should be optimized taking the context of the problem into account. Hereto, an ensemble learning framework proposed by Caruana et al. will be adopted [50]. This approach allows, similar to the WHICH meta-learner framework recently proposed by Menzies, to optimize context specific evaluation metrics.

A second future research topic relates to the use of network learners to augment software fault prediction models. As noted earlier, software dependency networks (and other types of networks) have been found to complement static code based prediction models. This work typically considers a number of static code features of each code segment and also constructs a network, indicating the relationships between all segments. This network is then summarized into a few key characteristics (e.g. the betweenness centrality discussed in Section 1.3.1) for each segment (node) in the network. These characteristics, together with the static code features, are used as an input to traditional data mining techniques, which assume observations to be independent from each other. There exists however a class of learners that no longer impose this assumption: relational learners [213]. These learners try to generalize from the network topology itself and might prove to be a valuable addition as these models are able to incorporate information relating to neighboring code segments, thus improving the information density of the data.

1.5 Overview of publications

The findings presented in this thesis resulted in the following publications; where appropriate, the 5-Year Impact Factor is also indicated.

Journal publications

K. Dejaeger, T. Verbraken and B. Baesens, "Towards comprehensible software fault prediction models using Bayesian network classifiers," *IEEE Transactions on Software Engineering*, Accepted for publication. ***3.468***

H.-T. Moges, K. Dejaeger, W. Lemahieu and B. Baesens, "A total data quality management for credit risk: new insights and challenges," *International Journal of Information Quality*, 3 (1): 1–27, 2012.

H.-T. Moges, K. Dejaeger, W. Lemahieu and B. Baesens, "A Multidimensional Analysis of Data Quality for Credit Risk Management: New Insights and Challenges," *Information and Management*, Accepted for publication. ***3.796***

K. Dejaeger, W. Verbeke, D. Martens and B. Baesens, "Data mining techniques for software effort estimation: a comparative study," *IEEE Transactions on Software Engineering*, 38 (2): 375–397, 2012. ***3.468***

K. Dejaeger, A. Giangreco, L. Mola and B. Baesens, "Gaining insight into student satisfaction using comprehensible data mining techniques," *European Journal of Operational Research*, 218 (2): 548–562, 2012. ***2.512***

W. Verbeke, K. Dejaeger, D. Martens, J. Hur and B. Baesens, "New insights into churn prediction in the telecommunication sector: a profit driven data mining approach," *European Journal of Operational Research*, 218 (1): 211–229, 2012. ***2.512***

J. Huysmans, K. Dejaeger, C. Mues, J. Vanthienen and B. Baesens, "An empirical evaluation of the comprehensibility of decision table, tree and rule based predictive models," *Decision Support Systems*, 51 (1): 141–154, 2010. ***2.568***

Conference publications

M. Baojun, K. Dejaeger, J. Vanthienen and B. Baesens, "Software defect prediction based on association rule classification," *International Conference on Electronic-Business Intelligence*, 2010.

R. Setiono, K. Dejaeger, W. Verbeke, D. Martens and B. Baesens, "Software effort prediction using regression rule extraction from neural networks," *22^{nd} International Conference on Tools with Artificial Intelligence*, pp. 45–52, 2010.

K. Dejaeger, B. Hamers, J. Poelmans, B. Baesens, "A novel approach to the evaluation and improvement of data quality in the financial sector," *15^{th} International Conference on Information Quality*, 2010.

W. Verbeke, K. Dejaeger, D. Martens and B. Baesens, "Customer churn prediction: does technique matter?," *Joint Statistical Meeting*, 2010.

H.-T. Moges, K. Dejaeger, W. Lemahieu and B. Baesens, "Data Quality for Credit Risk Management: New Insights and Challenges," *16^{th} International Conference on Information Quality*, 2011.

Conference posters

M. Mylle, K. Dejaeger and E. Dejaeger, "Welke parameters zijn determinerend voor ontslag bij de oudere CVA patiënt na revalidatie?," *34^{ste} Winter Meeting voor Gerontologie en Geriatrie*, 2011.

F. Wynants, K. Dejaeger, L. De Wit and E. Dejaeger, "Verband tussen cholesterol en survival na CVA bij de oudere patiënt," *34^{ste} Winter Meeting voor Gerontologie en Geriatrie*, 2011.

W. Verbeke, K. Dejaeger, T. Verbraken, D. Martens and B. Baesens, "Mining social networks for customer churn prediction," *Workshop on Information and Decision in Social Networks*, 2011.

International publications

K. Dejaeger, S. vanden Broucke, T. Eerola, R. Wehkamp, L. Goedhuys, M. Riis and B. Baesens, "Beyond the hype: cloud computing in analytics," *Data Insight & Social BI: Executive Update*.

National publications

K. Dejaeger, J. Ruelens, T. Van Gestel, B. Baesens, J. Poelmans and B. Hamers, "Evaluatie en verbetering van de datakwaliteit," *Informatie*, 51 (9): 8–15, 2009.

K. Dejaeger, W. Verbeke, D. Martens and B. Baesens, "De kosten van softwareontwikkeling voorspellen," *Informatie*, 52 (9): 8–13, 2010.

J. Poelmans, K. Dejaeger and G. Dedene, "Software requirements engineering: een iteratieve aanpak," *Informatie*, 53 (5): 8–13, 2011.

K. Dejaeger, S. vanden Broucke, T. Eerola, R. Wehkamp, L. Goedhuys, M. Riis and B. Baesens, "Cloud computing in analytics: de hype ontraadseld," *Informatie*, Accepted for publication.

Conference abstracts

K. Dejaeger, T. Verbraken and B. Baesens, "Assessing Bayesian Network classifiers for software defect prediction," *25^{th} EURO conference*, 2012.

K. Dejaeger, W. Verbeke, J. Huysmans, C. Mues, J. Vanthienen and B. Baesens, "Rule based predictive models, decision table and tree: an empirical evaluation on comprehensibility," *24^{th} EURO conference*, 2010.

W. Verbeke, K. Dejaeger and B. Baesens, "Comparing classification techniques to forecast customer churn," *OR52 Annual Conference*, 2010.

K. Dejaeger, R. Setiono, W. Verbeke, D. Martens and B. Baesens, "Software effort prediction using regression rule extraction from neural networks," *OR52 Annual Conference*, 2010.

Technical reports

M. Baojun, K. Dejaeger, J. Vanthienen and B. Baesens, "Software defect prediction based on association rule classification," *FBE Research Report KBI_1105*, 2011.

K. Dejaeger, Louis P., S. vanden Broucke, T. Eerola, L. Goedhuys and B. Baesens, "Beyond the hype: cloud computing in analytics," *FBE Research Report KBI_1220*, 2012.

Work in progress

K. Dejaeger, S. vanden Broucke, S. Lessmann, M. Shepperd and B. Baesens, "Benchmarking Classification Models for Software Defect Prediction: Revisiting Earlier Results," Submitted to: *IEEE Transactions on Software Engineering*.

M. Baojun, K. Dejaeger, B. Baesens, J. Vanthienen and G. Chen, "Investigating associative classification for software fault prediction: an experimental perspective," Submitted to: *Software Quality Journal*.

Mathematics is written for mathematicians.

Nicolaus Copernicus, 1473–1543

2

Machine Learning: general concepts

This chapter introduces the notation adopted throughout the remainder of this dissertation, and subsequently presents the general supervised machine learning taxonomy which serves as an underpinning to the next chapters. Next, basic concepts relating to model evaluation are detailed, also discussing the dimension of model comprehensibility. Finally, all concepts are linked to each other by making use of the concept of the Knowledge Discovery in Databases (KDD) cycle and an application hereof in the domain of student satisfaction is presented.

Parts of this chapter have been published in

- K. Dejaeger, A. Giangreco, L. Mola and B. Baesens, "Gaining insight into student satisfaction using comprehensible data mining techniques," *European Journal of Operational Research*, 218 (2): 548–562, 2012.
- R. Setiono, K. Dejaeger, W. Verbeke, D. Martens and B. Baesens, "Software effort prediction using regression rule extraction from neural networks," *22nd International Conference on Tools with Artificial Intelligence*, pp. 45–52, 2010.

2.1 General notation

The following notation is adopted throughout the dissertation. A scalar $x \in \mathbb{R}$ is denoted in normal script while a vector $\mathbf{x} \in \mathbb{R}^n$ is in boldface script. A vector is always a column vector unless indicated otherwise. A row vector is indicated as the transposed of the associated column vector, $\mathbf{x}'$. A matrix $\mathbf{X} \in \mathbb{R}^{N \times n}$ is in bold capital notation. i is used to identify observations while j refers to attributes; hence, $x_{i(j)}$ is an element of matrix $\mathbf{X}$ representing the value of the j^{th} variable on the i^{th} observation. N is used as the number of observations in a data set while n represents the number of variables.

When considering a regression task, the target variable is a scalar $e \in \mathbb{R}$; in the context of software effort estimation, the target variable is effort in man-months or an equivalent measure. Consequently, the actual effort of the i^{th} software project is indicated as e_i while the predicted effort is indicated as $\hat{e}_i$. In line with this notation, the task of estimating the software development effort can be defined as follows: let $D_{trn} = \{(\mathbf{x}_i, e_i)\}_{i=1}^{N}$ be a training data set containing N observations, where $\mathbf{x}_i \in \mathbb{R}^n$ represents the characteristics of a

software project and $e_i \in \mathbb{R}$ is the continuous target variable. A software effort estimation model is then formally defined as a function f mapping instances to the continuous target, $f(\mathbf{x}_i) : \mathbb{R}^n \mapsto \hat{e}_i$.

When on the other hand considering a classification task, the target variable, y_i, is a dichotomous attribute indicating the class of an instance. The number of classes is denoted by K. Thus, in case of software fault prediction, the target variable, y_i, is a dichotomous attribute indicating whether a software module contains faults ($y_i = 1$) or not ($y_i = 0$) and $K = 2$. In line with the above notation, the task of learning a software defect model can be defined as follows: let $D_{trn} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ be a training set containing N observations, where $\mathbf{x}_i \in \mathbb{R}^n$ represents the static code features characterizing the i^{th} instance, and $s(\mathbf{x}_i) \in]-\infty, \infty[$ is a score indicating the propensity of belonging to the class of fault prone code segments. A software defect model is then formally defined as a function m mapping instances to real values (scores) on some unspecified scale, $m(\mathbf{x}_i) : \mathbb{R}^n \mapsto s(\mathbf{x}_i) \in \mathbb{R}$. Alternatively, a classifier c maps instances to a crisp label which indicates the presence of faults, $y_i \in \{0, 1\}$, possibly by setting a threshold t on the scores. Formally, $c(\mathbf{x}_i) : \mathbb{R}^n \mapsto y_i \in \{0, 1\}$.

2.2 Machine Learning

Supervised machine learning reasons on the basis of externally supplied instances to learn hidden patterns, allowing to score unseen instances. Depending on the values the target attribute can take, a distinction between classification and regression is made. Machine learning entails however not only classification or regression tasks, but also e.g. reenforcement learning, association rule mining and clustering [301].

Reenforcement learning pitches an agent in a dynamic environment and provides a scalar reenforcement signal indicating the operational performance as input to this agent. The agent is not told which actions to take, but instead must iterate over all possible actions to find out which actions result in the highest reward. An example application is the learning to control a RC airplane by such agent[1].

Association rule mining tries to find relationships between attributes, not requiring per se a target variable. Let $I = \{i_1, \ldots, i_n\}$ be a set of items. An association rule is then an implication of the form $x \rightarrow x'$ where $x \in I$ and $x' \in I$ are called itemsets and $x \cap x' = \emptyset$. Often used for market-basket analysis, association rule mining can also be applied to e.g. a classification setting by inferring rules of the form $x \rightarrow y$ with sufficient support and confidence. An example of the application of association rule based classification within the domain of software fault prediction is presented in Baojun et al. [25].

Clustering is another unsupervised machine learning task in which one tries to group instances in clusters by minimizing the intra-cluster distance while at the same time maximizing the inter-cluster distance. Clustering techniques are

[1]For a demonstration on reenforcement learning, please refer to e.g. the online machine learning classes of Standford university. www.youtube.com/watch?v=UzxYlbK2c7E

ML based algo.	**Perceptron based**	**Evolutionary algo.**	**Swarm intelligence**
# 186 - *59.8%*	# 84 - *27.0%*	# 29 - *9.32%*	*unknown*
Kernel methods	**Staticical methods**	**Ensemble learners**	**Other techniques**
# 89 - *28.6%*	✓Regression	✓Bagging	✓SNA
	# 180 - *57.9%*	# 63 - *20.3%*	# 44 - *14.2%*
	✓Bayesian	✓Boosting	
	# 68 - *21.9%*	# 73 - *23.5%*	

Table 2.1: Usage of ML techniques: KDnuggets survey results of 2012

often used as a data preprocessing or data exploration step [334]. E.g. Self Organizing Maps (SOMs) can also be used for dimensionality reduction or data visualisation [276].

Several supervised machine learning taxonomies have been put forward in the literature, see e.g. Kotsiantis [189], who distinguished logic based algorithms from perceptron-based techniques, support vector machines, statistical learning algorithms, and instance-based learning. Figure 2.1 presents the general supervised machine learning taxonomy adopted in this dissertation; it should be noted that, depending on the task at hand, some types of techniques are only suitable for regression (e.g. linear regression) or classification (e.g. discriminant analysis). These are indicated by an r and c respectively. Evidently, some techniques are more popular than others; this can be due to a variety of reasons, including predictive power, computational efficiency, noise tolerance, the possibility to use graphical artifacts, the ability to deal with redundant or irrelevant attributes, and the ease of using the algorithm. Table 2.1 reports on the popularity vis-à-vis our taxonomy; the numbers of users per type of learner are taken from the yearly KDnuggets[2] survey on this topic. Note that these numbers are also dependent on the research domain; other techniques are more popular in e.g. the domain of software engineering [76] or medical sciences [152].

2.2.1 ML based algorithms

Machine Learning (ML) based algorithms can be further subdivided into decision tree inducing algorithms and those that output a rule set. It can be shown that decision trees and rule sets consisting of mutually exclusive propositional rules (cfr below) are logically equivalent in the sense that one type of representation can be automatically translated into another (albeit in a simpler or more complex form), while preserving the predictive behavior of the original model [145,321]. Due to their expressive possibilities and computational efficiency, this family of techniques have been often adopted in business and research alike.

[2]www.kdnuggets.com

Figure 2.1: Supervised machine learning: proposed taxonomy

Decision tree algorithms

A decision tree consists of a number of internal nodes, specifying conditions to be tested, and a number of leaf nodes with a class label. New observations can be classified by traversing the tree from top to bottom where condition tests in the internal nodes indicate whether the left or right branch must be followed. The tree is constructed in a top-down fashion, considering all training instances in a first step, recursively splitting the set of instances into smaller, more homogenous subsets according to some heuristic to assess the purity of the splits. In case of classification, the splitting heuristic is often derived from Shannon's information theory [279]. More specifically, entropy-based measures are commonly adopted hereto, which can be seen as a proxy to the purity of a data set with respect to the class variable. Let $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ be a set of instances and $P(y_i = 0)$ ($P(y_i = 1)$) be the prior probability of instance i to belong to class 0 (class 1). The entropy is then defined as:

$$\text{Entropy}(D) = -P(y_i = 1)\times\log_2(P(y_i = 1)) - P(y_i = 0)\times\log_2(P(y_i = 0)) \quad (2.1)$$

For example, the ID3 and the C4.5 algorithm (also sometimes known as J48) employ a splitting criterion which quantifies the gain in purity (i.e. the decrease in entropy) during model construction. The CART (Classification And Regression Tree) learner utilizes yet another splitting heuristic, the Gini diversity index:

$$\text{Gini}(D) = 1 - \sum_{k=1}^{K} P(y_i = k)^2 \quad (2.2)$$

Alternatives

Many decision tree learners can be adjusted to a regression setting, by changing the splitting criterion at the internal nodes. One option is e.g. to split the data in such a way that the variance is minimal at the child nodes. Note that also further extensions are possible, by e.g. fitting a regression function to the instances at each leaf node [263]. In the next chapter, both CART for regression and M5′ are further discussed in the context of software effort estimation.

Also other alternatives to the ubiquitous C4.5 decision tree learner exist, which e.g. consider splits based on multiple attributes, resulting in non-axis parallel decision boundaries such as the Oblique Classifier 1 (OC1) explained later in this chapter. Other tree learners employ statistical tests such as the CHi-squared Automatic Interaction Detection (CHAID) algorithm [170].

Pruning

Decision tree learners are known to be sensitive to overfitting the training data; without stopping criterion, a tree would try to fit the idiosyncracies in the training data, leading to a tree which will poorly generalize to unseen instances. Thus, many decision tree learners employ a pruning procedure and/or use a stopping criterion to prevent the tree from overfitting the training data. For instance, CART and OC1 employ a cost-complexity pruning procedure, regulated by a complexity parameter α, which can be seen as a parameter to balance additional predictive performance against a more complex tree. Briefly, this

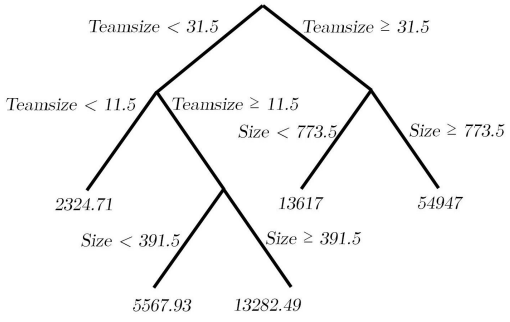
procedure works as follows:
Construct trees of decreasing size from the original, complete tree, iteratively collapsing the child nodes until only a decision stump remains. Classify the validation set using the pruned trees and measure each time the classification performance. Finally, the smallest tree is selected which is within α standard errors of the tree with the best classification performance.

Rule induction

The most common type of rules is without any doubt propositional if-then rules. The condition part of a propositional rule consists of a combination of conditions on the input variables. While the condition part can contain conjunctions, disjunctions, and negations, most algorithms will return rules that only contain conjunctions. As an example, the rule proposed in Visual Studio to indicate a candidate module for refactoring is given below.

IF cyclomatic complexity > 25 THEN fault prone
DEFAULT:not fault prone

Most algorithms will ensure that the condition parts of each rule demarcate separate areas in the input space: i.e., the rules are mutually exclusive. Therefore, only one rule is satisfied when a new observation is presented and that rule will be the only one used for making the classification decision. Other algorithms allow multiple rules to fire for the same observation. This requires an additional mechanism to combine the predictions of individual rules, such as assigning a confidence factor to each rule or sorting the rules and allowing only the first firing rule to decide. Finally, it should be noted that a tree is logically equivalent to a set of IF-THEN rules [145]; Fig. 2.2 illustrates this on the ISBSG R11 software effort estimation data set.



Equivalent rule set
IF Teamsize >= 31.5 AND Size < 773.5 THEN effort = 13617
IF Teamsize >= 31.5 AND Size >= 773.5 THEN effort = 54947
IF Teamsize < 11.5 THEN effort 2324.71
IF Teamsize >= 11.5 AND Teamsize < 31.5 AND
Size < 391.5 THEN effort = 5567.93
ELSE effort = 13282.49

Figure 2.2: Pruned CART tree induced on the ISBSG R11 data set

Several popular rule induction learners exist, including Repeated Incremental Pruning to Produce Error Reduction (RIPPER), which produces ordered rule sets and the OneR learner of Holte, which learns exactly one rule, with a single rule antecedent [142]. Menzies et al. compared the OneR learner to J48 and Naive Bayes with a log transformation of the attribute space. They reported that 'predictors that use simple threshold comparisons (OneR) perform worse than predictors built from more elaborate decision trees (J48)' [230]. However, Holte showed that the gain of very simple models (i.e. OneR rule sets) to more complex models (i.e. C4.5 trees) is often limited, concluding that 'simple-rule learning systems are often a viable alternative to systems that learn more complex rules' [142]. Arisholm et al. compared PART, a more recent alternative to RIPPER with C4.5, neural networks, and other classification algorithms in the context of software fault prediction and concluded that the performance of PART varied widely while e.g. C4.5 yielded more consistent results [12]. Note that e.g. in a telco setting, Verbeke et al. found rule sets induced by RIPPER to be small and comprehensible, while retaining a high sensitivity, comparing this technique to a.o. C4.5, Antminer+ and the ALBA rule extraction technique.

2.2.2 Perceptron based models

Neural networks (NNs) are mathematical representations inspired by the functioning of the human brain [34], and have previously been applied in various contexts, including software effort estimation and software fault prediction, as they enjoy some beneficial properties.

- NNs have previously been applied with success on data sets with complex relationships between inputs and output, and where the input data is characterized by high noise levels.
- NNs with one hidden layer have been proven to be universal approximators which can approximate any continuous function or mapping to a discrete target to a desired degree of accuracy.

The most recurring type of NN are multilayer perceptron (MLP) networks. A MLP network is typically a three-layer feedforward (i.e. not containing recursive loops) network with each layer consisting of several neurons. Assuming a hidden layer consists of H hidden neurons, the output of the h^{th} hidden neuron is computed as:

$$h = f_{hidden}(w_h + \sum_{j=1}^{n} \mathbf{W}_{jh} \times x_{(j)}) \tag{2.3}$$

Herein represents w_h the bias of the h^{th} hidden neuron and $\mathbf{W}$ the weight matrix, whereby $\mathbf{W}_{jh}$ is the weight of the edge connecting input node j with hidden node h. Function $f_{hidden}(.)$ is a so-called transfer function, which can e.g. be a linear, a threshold or a sigmoid function. The output of the MLP

network is finally given by:

$$\hat{y} = f_{output}(v_0 + \sum_{h=1}^{H} \mathbf{V}_h \times h) \tag{2.4}$$

with $\mathbf{V}_h$ the weight matrix on the edges from the hidden layer to the output node and v_0 the bias of the output node. Again, f_{output} represents some transfer function. The randomly initiated biases and weights are iteratively updated as the learning algorithm is presented with new training instances to generalize from.

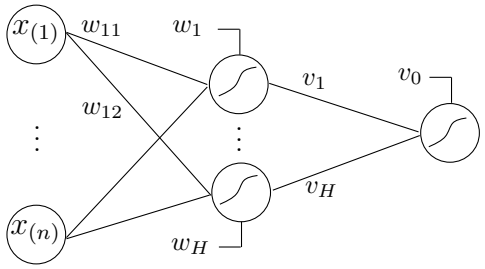


Figure 2.3: Three layered MLP topology

Back propagation is arguably the most well known training algorithm and uses in fact a gradient descend approach in which the network weights are iteratively updated along the most negative direction of the gradient of the performance indices, i.e. the direction in which the performance function decreases the most rapidly, assuming e.g. the sum of squared errors as performance function. One iteration of this algorithm can be written as:

$$[\mathbf{W}\ \mathbf{V}]_{t+1} = [\mathbf{W}\ \mathbf{V}]_t + \alpha_t \times g_t \tag{2.5}$$

with α_t the learning rate at iteration t and g_t the gradient. The learning rate is a parameter which can be set to enable faster convergence of the algorithm. Note that also further improvements to the algorithm are possible, such as using a momentum parameter in order to avoid getting stuck in a local minimum or conjugate gradient descend algorithms which will also perform a line search to determine the optimale distance along the current gradient.

There also exist learning methods based on the Hessian of the performance indices, referred to as Newton algorithms, which boosts faster convergence. However, as the Hessian is expensive to calculate in the case of MLPs, a number of approximative algorithms have been proposed, including the Levenberg-Marquardt algorithm, which use the first order derivatives to approximate the Hessian [120].

Finally, note the possibility to introduce a weight penalization term during training, allowing the removal of irrelevant and redundant input units and hidden nodes. An example of the application of such algorithm in the context of software effort estimation can e.g. be found in Setiono et al. [278].

RBF networks

Radial Basis Function (RBF) networks are a special case of neural networks, rooted in the idea of biological receptive fields [240]. A RBF network is a three-layer feedforward network consisting of an input layer, a hidden layer typically containing multiple neurons, and a linear output layer. Each of the neurons is positioned in the input space, e.g. by applying a clustering procedure in the input space. The output of each hidden unit, when confronted with an unseen case, is inversely proportional to the distance between this instance and the center of this neuron. In calculating the output of the hidden units, a radial symmetric gaussian transfer function is typically used:

$$\text{radbas}(x_i) = e^{-||c_k - x_i|| \times b^2} \tag{2.6}$$

where c_k is the k^{th} cluster centroid, $||.||$ the Euclidian distance between two points, and b a bias term. Hence, each neuron has its own receptive field in the input domain: a region centered on c_k with size proportional to the bias term, b [240].

2.2.3 Evolutionary algorithms

Evolutionary algorithms are developed for tackling general optimization problems and are based on the Darwinian theory of natural selection. This theory states in essence that genetic operations between individuals eventually generate fitter individuals which are better suited for survival. In case of Genetic Algorithms (GA), an individual is represented by a fixed length binary string. Genetic Programming (GP) is an extension to GA in which an individual is some algebraic expression of arbitrary length [190]. This expression is evaluated to assess the fitness of the individual. Learning a GP model in general constitutes of three phases.

1. Generate the initial population of solutions
2. Create a new population based on the previous population by applying genetic operations to a selection of individuals of the current population
3. Iterate step (2) until some stopping criterion is met or a specified number of generations has been reached

Application of GP requires a number of preparation steps. First, a suitable set of operators needs to be selected; this set of operators should reflect the relations expected in the data set. Commonly used operators include $\{+, -, \times, /, \log\}$. Secondly, a suitable representation format has to be decided upon. A typical representation format used in GP are trees. Finally, a number of parameters needs to be specified, including how the initial population is constructed, which genetic operators are applied (e.g. mutation and crossover) and the evaluation criterium used to score the individuals at each iteration. Also which individuals make it to the next round is a parameter that can be specified. Evolutionary algorithms have been used in various domains, including software fault and effort estimation [49, 209].

2.2.4 Swarm intelligence

Swarm intelligence studies systems which are composed of many individuals who interact with each other and their environment. This family of techniques found its origin in the success of biological swarm systems such as ant hives and the flocking behavior of birds. While individuals exhibit limited cognitive abilities, the swarm behaves more intelligent by finding efficient solutions for complex problems such as finding the shortest path to a food source or predator evasion. The typical swarm intelligence system exhibit the following properties [82].

- it is composed of many individuals
- all individuals are similar to one other
- the interactions amongst individuals are mandated by simple behavioral rules, using information derived from the local environment
- the system coordinates itself, without the presence of an external controller

Swarm intelligence for supervised machine learning can be further subdivided into Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO), which are the two main meta-heuristics in this field of research. Swarm intelligence has been accredited for learning robust and scalable models and have been the focus of many recent research [219].

Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic based on the foraging behavior of ants, and underlying to ACO is the observation that shorter paths will be attributed with higher pheromone levels. Evaporation causes the pheromone levels of all trails to diminish, and trails that fail to be chosen gradually lose pheromone and will be selected by subsequent ants with a lower probability.

ACO initiates by constructing a solution space in which the values of discrete attributes constitute the nodes (a vertex group) and links between any two values of two subsequent attributes the edges, assuming some random ordering on the attribute space. A dummy vertex signifying a random value for that attribute is added to each vertex group. Ants are created and traverse the network from source to sink, each path corresponding with a classification rule. The path followed by each ant depends on the pheromone levels of that path, and, depending on how pheromone is added to the paths, a further distinction can be made between different ACO approaches. E.g. the Elitist Ant System will deposit pheromone on all traversed paths as well as the global best path on every iteration while the $max - min$ Ant System [298] will add pheromone only to the best path. Also, to prevent early stagnation, the pheromone levels will be limited to a range $[\tau_{min}, \tau_{max}]$. The well known Antminer+ learner is an example of a $max - min$ Ant System and has been applied in various contexts, including to the domain of software fault prediction [17, 320]. It was found

to perform on par with ML based algorithms such as C4.5 and Ripper while resulting in smaller rule sets.

Note that ACO is arguably the most popular swarm intelligence meta-heuristic and its application area is limited to classification problems. It was recently recognized that one of the main elements holding back the wide spread use of ACO systems is the limited availability of software implementations [219]. Noteworthy in this regard is the availability of e.g. the AntMiner+ algorithm in the Matlab environment[3].

Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a second swarm intelligence meta-heuristic which was proposed in 1995 by Kennedy and Eberhart [172]. In PSO, a number of simple entities, the particles, are positioned in the search space of some continuous optimization problem. Each particle represents a possible solution, and will be moved through the search space by combining some aspects of the local history, i.e. information regarding its own current location and best previous location, with information of one or more members of the swarm [257]. Each particle a constitutes three vectors. These are the current position, $\overrightarrow{x}_a$, the previous best location, $\overrightarrow{p}_a$, and the velocity, $\overrightarrow{v}_a$. At each iteration of the algorithm, the current position of each particle is updated by the following formula.

$$\begin{cases} \overrightarrow{v}_a \leftarrow \overrightarrow{v}_a + \overrightarrow{U}(0,\phi_1) \otimes (\overrightarrow{p}_a - \overrightarrow{x}_a) + \overrightarrow{U}(0,\phi_1) \otimes (\overrightarrow{p}_g - \overrightarrow{x}_a), \\ \overrightarrow{x}_a \leftarrow \overrightarrow{x}_a + \overrightarrow{v}_a \end{cases} \quad (2.7)$$

Herein represents $\overrightarrow{U}(0,\phi_k)$ a vector of random numbers that are uniformly distributed in $[0,\phi_i]$ and $\otimes$ represents a component-wise multiplication. $\overrightarrow{p}_g$ is the best location of the particles in the neighborhood of particle a. Initially, the particles are scattered randomly through the search space; as the algorithm converges, the particles will stochastically be moved to their own best solution together with the neighborhoods' best solution. This idea is illustrated in Fig. 2.4. The number of particles and the perturbation vectors ϕ_1 and ϕ_2 (also referred to as the acceleration coefficients) are parameters of the algorithm. Note that especially the acceleration coefficients are of crucial importance to the convergence of the algorithm as too large values will result in uncontrolled particle acceleration while too low values results in unresponsive systems. The impact of the acceleration coefficients is indicated by the shaded areas in the figure, and are re-initiated at each iteration. A typical value for ϕ_i is 2, while other work also defined a maximum bound on the speed of the particles, $\overrightarrow{v}_{max}$. Note that PSO has been applied with some success in a software engineering setting [70].

2.2.5 Kernel methods

In the last few years, much attention has been given to kernel methods. Arguably, the main protagonist in this domain is the Support Vector Machines

[3] www.antminerplus.com

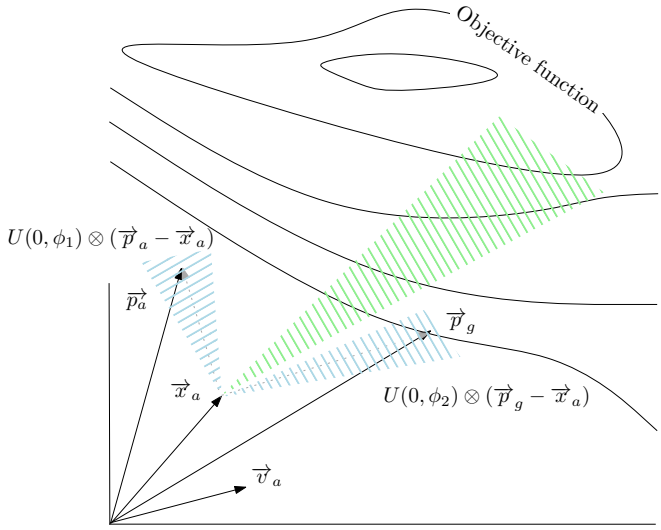


Figure 2.4: The update of a particle in the original PSO algorithm

(SVM) learner, but e.g. Gaussian processes have also gained ground in certain application fields [265]. However, to the best of our knowledge, there has been no work discussing the use of Gaussian processes in software effort nor fault prediction and thus our focuss will lie on this main protagonist: Support Vector Machines (SVM).

Support Vector Machines

SVM is a (non) linear modeling technique based on recent advances in statistical learning theory [322] and constructs a maximum margin hyperplane separating the instances of two classes. In essence, the following objective function is minimized.

$$\begin{aligned} \min \quad & \frac{||\mathbf{w}||^2}{2} + C \sum_{i=1}^{N} \varepsilon_i \\ \text{s.t.} \quad & y_i[\mathbf{w}\phi(\mathbf{x}_i) + b] \geq 1 - \varepsilon_i, \ i = 1, \ldots, N \\ & \varepsilon_i \geq 0, \ i = 1, \ldots, N \end{aligned} \tag{2.8}$$

Herein, $\mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$ represent the parameters of the SVM model that are learned during training, and ε_i is a slack variable to allow to construct a maximum margin hyperplane while misclassifying some training instances. C is a hyperparameter representing the penalty of misclassifying training instances and can be set by the user. Finally, $\phi(.)$ is some unspecified function which transforms the data to a higher, possibly infinite, dimensional space. This convex optimization problem can be solved using the method of Lagrange multipliers

and hereto, the Lagrangian to the constrainted optimization problem is defined:

$$\min \frac{||\mathbf{w}||^2}{2} + C\sum_{i=1}^{N}\varepsilon_i - \sum_{i=1}^{N}\lambda_i\{y_i[\mathbf{w}\phi(\mathbf{x}_i)+b]-1+\varepsilon_i\} - \sum_{i=1}^{N}\mu_i\varepsilon_i \qquad (2.9)$$

where λ_i and μ_i are the Lagrange multipliers. By reformulating this expression in function of the Lagrange multipliers, and then solving by numerical techniques such as quadratic programming, the following classifier is obtained:

$$y(\mathbf{x}) = \text{sign}\left[\sum_{i=1}^{N}\lambda_i y_i K(\mathbf{x}_i,\mathbf{x}) + b\right] \qquad (2.10)$$

Herein, $K(\mathbf{x}_i,\mathbf{x}) = \phi(\mathbf{x}_i)\cdot\phi(\mathbf{x})$ is a positive definite kernel satisfying the Mercer theorem. The advantage of using a kernel function lies in the fact that the mapping function $\phi(.)$ does not need to be explicitly known. The Mercer theorem in fact guarantees that a kernel function $K(.,.)$ can always be expressed as a dot product between the constituting input vectors. Typical examples of such kernel functions are:

$$\begin{array}{ll} K(\mathbf{x}_i,\mathbf{x}) = (\mathbf{x}_i\cdot\mathbf{x}+1)^p & \text{(Polynomial kernel)} \\ K(\mathbf{x}_i,\mathbf{x}) = e^{-||\mathbf{x}_i-\mathbf{x}||^2/2\sigma^2} & \text{(RBF kernel)} \end{array} \qquad (2.11)$$

The Polynomial kernel has one user defined parameter, the exponent p. If $p = 1$, it simplifies to a so-called linear kernel, resulting in a linear decision boundary in the input space. The Radial Basis Function (RBF) kernel has also one parameter, σ or the kernel bandwidth, which determines the sensitivity to perturbations in the training data. The different kernel functions are illustrated in Fig. 2.5 on a subset of the Pima Indians Diabetes data set, hosted on the UCI repository[4].

Kernel methods have been subject of much recent work in the field of empirical software engineering; see e.g. [88]. An extensive literature overview will be presented in the next chapters.

2.2.6 Statistical methods

Several methods used in supervised ML research find their origin in the statistical literature. These include (non) linear regression, logistic regression, discriminant analysis and Bayesian networks. Note that also other statistical techniques exist, such as survival analysis and penalized estimation models which further assume time-related data [211]. Bayesian network learners are discussed in detail in Chapter 4.

(non) Linear regression

Arguably, one of the oldest and most widely applied regression techniques is Ordinary Least Squares (OLS) regression. This well documented technique fits

[4] http://archive.ics.uci.edu/ml/index.html

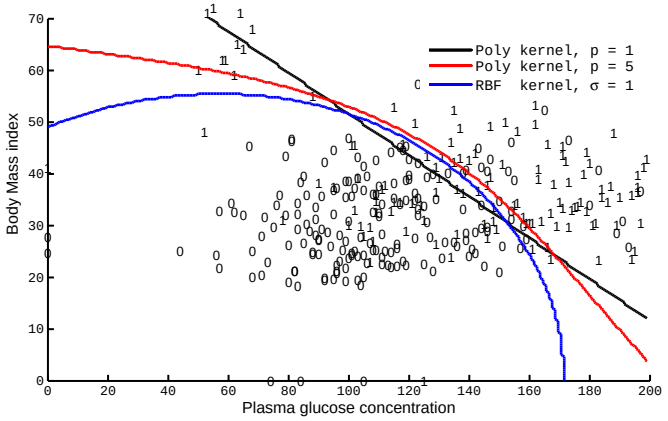


Figure 2.5: Comparison of kernels

a linear regression function to a data set containing a dependent, e_i, and multiple independent variables, $x_{i(j)}$; this type of regression is also commonly referred to as multiple regression. OLS regression assumes the following linear model of the data:

$$e_i = \mathbf{x}_i'\boldsymbol{\beta} + b_0 + \epsilon_i \tag{2.12}$$

where $\mathbf{x}_i'$ represents the row vector containing the values of the i^{th} observation. $\boldsymbol{\beta}$ is the column vector containing the slope parameters that are estimated by the regression, and b_0 is the intercept scalar. ϵ_i is the error associated with each observation and is used to estimate the regression parameters, $\boldsymbol{\beta}$, by minimizing the following objective function:

$$\min \sum_{i=1}^{N} \epsilon_i^2 \tag{2.13}$$

Several (non linear) alternatives have been put forward such as Robust regression and MARS (Multivariate Adaptive Regression Splines), as well as various forms of data transformations to capture non linear relationships in the data. These are further discussed in Chapter 3.

Logistic regression

Logistic regression is another well known statistical technique that fits the data to the following expression:

$$P(y_i = 0|x_i) = \frac{1}{1 + e^{-\mathbf{x}_i'\boldsymbol{\beta}}} \tag{2.14}$$

where $\boldsymbol{\beta}$ is a vector of unknown parameters. Logistic regression typically uses an iterative maximum likelihood parameter estimation procedure and outputs a

value between 0 and 1 which can be interpreted as a posterior class probability [336].

2.2.7 Ensemble learners

While the other ML approaches build a single model (with the exception of case based reasoning techniques, see also Section 2.2.8), it is possible to pool the outcome of several models to obtain a more robust final prediction. Hereto, several approaches exist, including bagging (bootstrapping the data, building each time a separate model), boosting (iteratively weighting instances according to the misclassification errors) and stacking (combining different classifiers based on their performance on a hold-out set). In the domain of empirical software engineering, specific attention has been given to Random Forest due to its good performance [118, 200].

Random forest

Random forest can be regarded as a classifier which consists of a collection of independently induced base classifiers which are then combined using a voting procedure. As originally proposed by Breiman, CART decision trees are adopted as base classifier [42]. Key in this approach is the dissimilarity amongst the base classifiers, which is obtained by adopting a bagging procedure to select the training samples of individual base classifiers and the selection of a random subset of attributes at each node, and the strength of the individual base models. More specifically, let $\{c(D_{tr}, \Theta_k)\}_{k=1}^{K}$ be a collection of K CART decision tree classifiers and Θ_k a random vector indicating the data selected for the k^{th} tree. It can then be shown that an upper bound to the misclassification error is given by the following expression.

$$Error^* \leq \bar{\rho}(1 - s^2)/s^2 \tag{2.15}$$

$\bar{\rho}$ represents the average correlation of the votes cast by the base classifiers and s represents the strength of a set of base classifiers. Due to the randomness of the bagging procedure, random forest will yield base classifiers which are only moderately correlated with one another.

More recently, an alternative to random forest was proposed: rotation forest. This ensemble technique takes the idea of random forest one step further by selecting a subset of features and instances, and subsequently applying Principal Component Analysis (PCA) hereon as a diversifying heuristic. Typically however, rotation forest results in base classifiers that are not as diverse as e.g. random forest and boosting, but on the counterbalance, the PCA procedure renders the base classifiers more accurate.

2.2.8 Other approaches

Various alternatives to the ones mentioned in the previous sections exist; Artificial Immune Systems for instance is a learning paradigm based on lymphocyte

cloning and mutation while Social Network Analysis considers the relationships between instances as an additional source of information, relaxing the assumption of independently and identically distributed data. One technique often used in empirical software engineering is instance based learning, sometimes also referred to as estimation by analogy or Case Based Reasoning (CBR).

Instance based learners

Contrary to other methods, instance based learning will not construct an explicit mapping from input to output space, but will instead provide an output based on the most similar cases in the training set. The distance function used to calculate the similarity between an unseen instance $\{\mathbf{x}_i, y_i\}$ and the instances from the training set D_{trn} is often the Euclidian distance. Irrelevant attributes should be discarded in this calculation and attributes should be rescaled, to avoid one attribute dominating all others. E.g. statistical tests can be adopted hereto [75]. Other aspects to consider include the number of most similar instances that should be selected and how the target value of these cases is transformed into a prediction of the unseen instance.

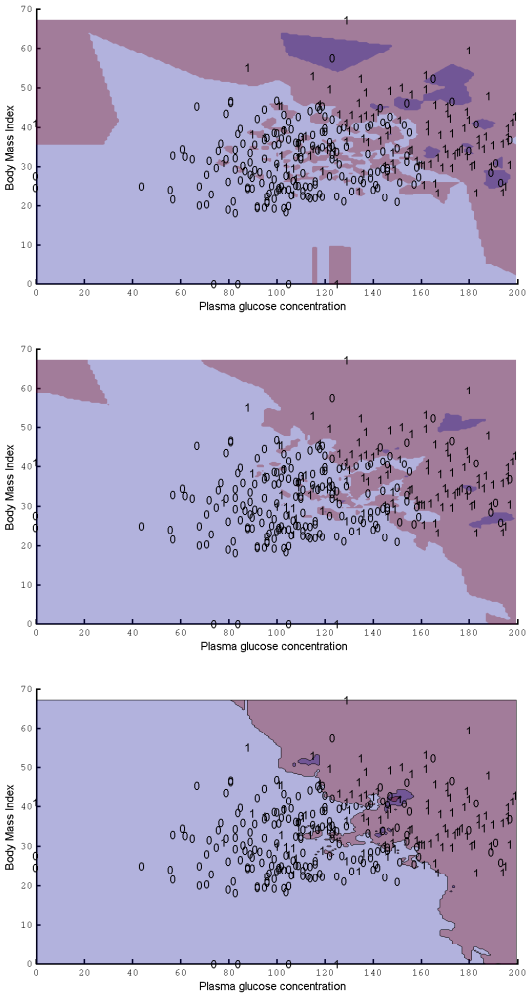


Figure 2.6: Varying the number of analogies, k=1, 5 and 15

Since this approach resembles the way in which experts form an opinion on e.g. the required time to complete a software project, it has been particularly popular in the software effort estimation literature. Note that this approach can result in arbitrary decision boundaries, and depending on the exact setup, can be made

more robust to idiosyncracies in the training data. Fig. 2.6 visualizes the decision boundaries derived by instance based learning on the Pima Indians Diabetes data set, while varying the number of most similar cases.

2.3 Model evaluation

2.3.1 Within data set

Typically, predictive ML involves the use of a training set, D_{trn}, from which patterns are extracted by learning the parameters of a ML model, and a set of unseen instances or test set, D_{ts}, on which the learned model is validated. Furthermore, note that several learners exhibit user-adjustable parameters, also termed hyperparameters, which allow learners to better model data set characteristics; examples include the C and the kernel bandwidth σ in case of SVM with a RBF kernel or the number of hidden neurons in a MLP model. Then, it is custom to set a part of the training set aside as independent validation set on which the impact of different hyperparameter values is assessed.

The data can be split according to various procedures including hold-out splitting, x fold cross validation, or leave-one-out cross validation. In case of a hold-out splitting procedure, the data is divided into training and test set, typically a rato of 2/3 and 1/3 respectively. However, a learner might benefit from a lucky split, and therefor, this random splitting procedure should be repeated multiple times, as is e.g. done by Dejaeger et al. [75]. Alternatively, a cross validation procedure can be employed, splitting the data in x bins of equal size, using all minus one bins for training and the last bin as a separate test set. The leave-one-out cross validation procedure is similar in the sense that there are as many bins as there are samples in the data set. Aspects like computational effort and the number of instances will dictate the preferred procedure [175]. As a rule of thumb, we suggest to use the leave-one-out cross validation procedure if N is smaller than 100. Finally, note the possibility of using a different procedure for training and tuning of a learner, as is illustrated in Fig. 2.7.

Furthermore, it should be pointed out that, preferably, the test set contains no samples also present in the training set [117]. For instance, the often used NASA MDP data sets exhibit large quantities of repeated instances, which positively impacts the predictive performance of supervised ML methods. However, as indicated by Dejaeger et al., the impact of these repeated vectors is limited [74]. This is also further discussed in Chapter 5.

Regression

Previously, a regression model was defined as a function f mapping instances to the continuous target, $f(\mathbf{x}_i) : \mathbb{R}^n \mapsto \hat{e}_i$, and a variety of performance measures has been proposed to gauge the strength of such regression model. Table 2.2 provides an overview of some typical regression metrics. Note that within the

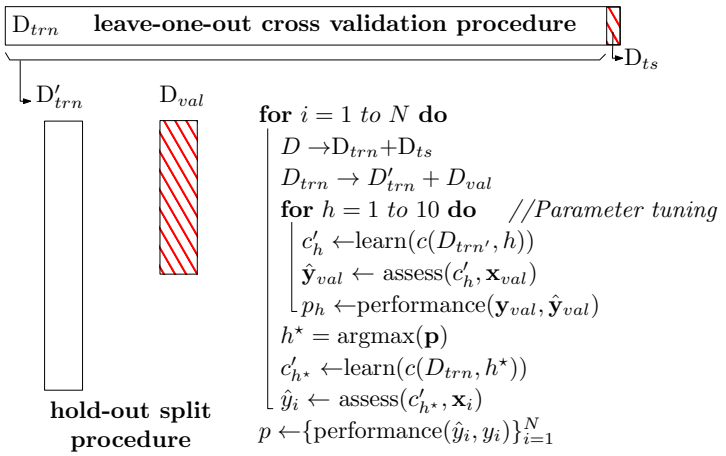


Figure 2.7: Validation procedures exemplified

Coefficient of determination	$R^2 = 1\text{-}\frac{\sum_i (e_i - \hat{e}_i)^2}{\sum_i (e_i - \bar{e})^2}$
Spearman's rank correlation	$r_s = 1 - \frac{6 \sum_i d_i^2}{N(N^2-1)}$
	$d_i = \triangle$ ordinal ranks
Root mean squared error	$\text{RMSE} = \sqrt{\frac{1}{N} \sum_i (e_i - \hat{e}_i)^2}$
Mean absolute error	$\text{MAE} = \frac{1}{N} \sum_i \lvert e_i - \hat{e}_i \rvert$

Table 2.2: Overview of regression performance metrics

field of software effort estimation, which in essence corresponds to a regression problem, a number of alternative metrics are regularly considered, including those based on the magnitude of relative errors. This can be attributed to the fact that some metrics suffer from a flaw or limitation and should be used with caution. E.g. Foss et al. note that both the standard deviation as well as the logarithmic standard deviation make certain assumptions to whether the data is homo- or heteroscedastic [101]. R^2 and the mean absolute error (MAE) are on the other hand known to be outlier sensitive [244]. As the debate on which metric should be preferred is still ongoing, it is common practice to report on a selection of metrics. Note that the rank reversal problem (a better performing model mistakenly found to be less accurate) cannot be ruled out, but by investigating a broad selection of metrics, and considering a robust validation procedure, this issue can be minimized. A literature review on software effort estimation, detailing empirical setup and evaluation metrics, is given in Chapter 3.

Classification

A classification model can be regarded as a function m mapping instances to real values (scores) on some unspecified scale, $m(\mathbf{x}_i) : \mathbb{R}^n \mapsto s(\mathbf{x}_i) \in \mathbb{R}$, and by setting a threshold t on these scores, a crisp classification can be obtained. Note that e.g. decision tree learners apply a majority vote at each leaf node to directly obtain such crisp classification. Based on this dichotomy, a confusion matrix can be obtained, comparing the number of correctly classified cases to those misclassified. This confusion matrix serves as the cornerstone to many of the metrics in use in the domain of software fault prediction [154]. Often, these 'single threshold' metrics neglect the issue of class imbalance and fail to discriminate between misclassification types. Fig. 2.8 provides an overview of these single threshold metrics.

Another commonly used tool in the performance measurement of classifiers is receiver operating characteristic (ROC) analysis [92]. A ROC curve shows the fraction of the identified faulty instances (the sensitivity) versus one minus the fraction of the identified fault free instances (one minus the specificity), for a varying threshold. The classifier corresponding to the dominating ROC curve is to be preferred; ROC curves can however also intersect one another. Although ROC curves are a powerful tool for comparing classifiers, practitioners prefer a simple numeric measure indicating the performance over the visual comparison of ROC curves. Therefore, single point metrics such as the area under the ROC curve (AUC) were proposed. ROC analysis is further discussed in the case study at the end of this chapter; a more recent alternative proposed by Hand, the H-measure, is introduced in Chapter 4. Also lift charts (known in software engineering as Alberg diagrams) and cost curves are sometimes considered.

An overview of classification metrics aimed at software fault prediction in five community flagship journals is presented in Table 2.3. From this overview can be concluded that the majority of articles are endorsed by single threshold metrics, although there is a tendency towards the use of alternative metrics such as Alberg diagrams and the AUC.

Finally, note that many of the above metrics can easily be extended towards a multiclass setting; this is also illustrated in the KDD case study presented in Section 2.4.

IEEE Trans. on Software Engineering	Year	Metrics
Quantitative analysis of faults and failures in a complex software system	2000	Alberg diagram
Assessing the applicability of fault-proneness models across object-oriented software projects	2002	correctness completeness
Empirical validation of object-oriented metrics on open source software for fault prediction	2005	correctness precision
Empirical analysis of object-oriented design metrics for predicting high and low severity faults	2006	correctness completeness precision

A replicated quantitative analysis of fault distributions in complex software systems	2007	Alberg diagram
Data mining static code attributes to learn defect predictors	2007	pd-pf balance
Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes	2007	accuracy
Empirical analysis of software fault content and fault proneness using bayesian methods	2007	Alberg diagram precision sens.-spec. fpr-fnr
Using the conceptual cohesion of classes for fault prediction in object-oriented systems	2008	correctness completeness precision
Benchmarking classification models for software defect prediction: A proposed framework and novel findings	2008	AUC
Evolutionary optimization of software quality modeling with multiple repositories	2010	fpr-fnr
A general software defect-proneness prediction framework	2011	balance AUC
Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts	2011	accuracy precision-recall F-measure
Towards comprehensible software fault prediction models using Bayesian network classifiers	2012	AUC H-measure
Software Quality journal		
Empirical validation of object-oriented metrics for predicting fault proneness models	2010	recall sens.-spec. F-measure
An industrial case study of classifier ensembles for locating software defects	2012	precision pd-pf balance
Predicting high-risk program modules by selecting the right software measurements	2012	AUC
Journal of Systems and Software		
The prediction of faulty classes using object-oriented design metrics	2001	accuracy J-coefficient AUC
Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics	2003	Alberg diagram

Predicting defect-prone software modules using support vector machines	2008	accuracy precision-recall F-measure
Applying machine learning to software fault-proneness prediction	2008	accuracy
Mining software repositories for comprehensible software fault prediction models	2008	accuracy sens.-spec.
The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process	2008	AUC
Increasing diversity: Natural language measures for software fault prediction	2009	Alberg diagram
A systematic and comprehensive investigation of methods to build and evaluate fault prediction models	2010	accuracy precision-recall fpr-fnr AUC cost effectiveness
A symbolic fault-prediction model based on multiobjective particle swarm optimization	2010	accuracy precision-recall F-measure AUC
On the ability of complexity metrics to predict fault-prone classes in object-oriented systems	2010	AUC
Comparing case based reasoning classifiers for predicting high risk software components	2010	J-coefficient
Empirical Software Engineering		
Uncertain classification of fault-prone software modules	2002	accuracy
Comparative assessment of software quality classification techniques: An empirical case study	2004	fpr-fnr
Assessment of a new three-group software quality classification technique: An empirical case study	2005	fpr-fnr
Techniques for evaluating fault prediction models	2008	Alberg diagram accuracy precision sens.-spec. F-measure J-coefficient geometric mean balance AUC cost curves
On the relative value of cross-company and within-company data for defect prediction	2009	pd-pf balance

Calculation and optimization of thresholds for sets of software metrics	2011	F-measure Matthews corr.
On the use of calling structure information to improve fault prediction	2012	Alberg diagram
Information and Software Technology		
Object-oriented software fault prediction using neural networks	2007	accuracy correctness completeness
Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry	2010	pd-pf balance
An ant colony optimization algorithm to improve software quality prediction models: Case of class stability	2011	accuracy
Transfer learning for cross-company software defect prediction	2012	precision-recall AUC
Fault prediction and the discriminative powers of connectivity-based object-oriented class cohesion metrics	2012	precision-recall AUC

Table 2.3: Overview of classification metrics in software fault prediction

Comprehensibility

As opposed to predictive performance, there exists no single point metric to quantify the understandability of a model. Typically, comprehensibility is regarded as the extent to which there exists a mental fit with the classification model and is thus somewhat subjective by nature [15]. It is argued that the main drivers of this mental fit are representation type and the specific task requirements. Previous research further indicated that individual differences such as prior experience and education also have an impact on the perceived comprehensibility of the decision model as well as model size, contributing to its subjectiveness [30]. Interesting in this regard is the work of Huysmans et al. who considered a modified task-representation fit model, see Fig. 2.9 [145]. It was found that, independent on the representation format (decision tables, univariate/multivariate decision trees and propositional IF-THEN rules were considered), the proportion of correct answers to both classification and logical questions dropped sharply for more elaborate models, questioning whether such degree of interpretability is acceptable in practical applications where these models must be validated or where explanatory power is deemed important.

In the second part of this chapter, a case study elucidating how valuable insights were obtained from data in the education domain is presented. Comprehensibility was a key requirement imposed by the management of the participating school, and therefore, especially ML based models were investigated. The reason being that one should be able to understand how a model reaches a spe-

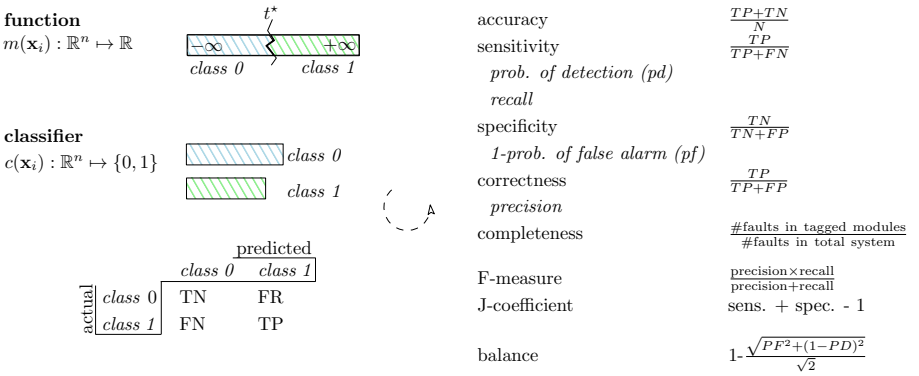


Figure 2.8: Overview of classification metrics

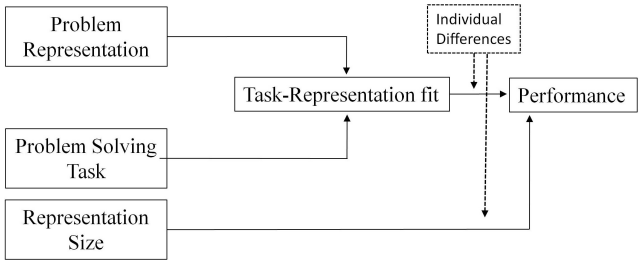


Figure 2.9: The modified task-representation fit model

cific decision and check whether the model is in line with previous assumptions on what drives perceived education quality.

2.3.2 Cross data set

It has been reckoned that the search for a universal predictor to software effort or faults is unlikely to be fruitful, as conclusion instability is an often recurring thema in empirical software engineering research [228]. On the other hand, data in this domain is often hard and time consuming to collect, if not impossible, necessitating the appliance of data from other projects or releases. Cross company or cross project studies consider data stemming from other teams, possibly developing software in unrelated domains while cross release validation questions the stability of conclusions across releases of the same software project. The concepts of cross company/project and cross release validation has attracted some interest, reporting very mixed results.

As explained in Chapter 1, organizations such as ISBSG and QSM have collected rich data relating to the software development effort in various economic sectors and countries. Kichenham et al. juxtaposed 10 cross company effort estimation studies, listing 4 studies that found no difference between cross company and within company data, while 4 others reported opposite results. 2 studies were inconclusive in this regard. As such, it was concluded that 'It is clear that some organizations would be ill-served by cross-company models whereas others would benefit' [178]. More recently, Kocaguneli et al. applied a relevance filtering prior to model training on the NASA, Cocomo81 and Desharnais data sets (see also Chapter 3), obtaining more promising results. More specifically, it was conjectured that new projects follow similar practices as historical projects and should require a similar development effort [184]. Echoing the findings of Koceguneli et al., Menzies et al., addressing the question whether locality is important during learning effort and defect models, stated that 'it was found that in 18 out of 20 *local* treatments, the treatments were completely different to the treatments learned from a *global* analysis of all the data' [228]. Both fault (Xalan and Lucene) and effort (China and NASA) data sets were scrutinized in this study.

Recently, the notion of concept drift for software fault prediction was introduced by Ekanayake et al. [87], underscoring the importance of out-of-time or out-of-universe validation to this domain. Zimmermann et al. for instance, investigating 12 real world applications including OSS like Eclipse and Firefox and CSS such as Internet Explorer, found that cross project fault prediction failed in 96.6% of all cases. Determinants of successful cross project validation were data (e.g. number of observations) and process characteristics (e.g. code churn or number of deleted lines). Jureczko et al. continued along the same line, applying a clustering on software project characteristics to identify possible sources of cross company data [214]. Turhan et al. analyzed NASA MDP data, in combination with data stemming from a Turkish white good manufacturer. Using a relevance filter, as was also done by Koceguneli et al. [184], a two step approach was proposed. When no company data is available, public repository data can be used after passing it through a relevance filter; however, as proprietary data becomes available, the use of cross company data should be phased-out, relying on own data [317]. More recently, He et al. revisited this topic, looking only at OSS data, and found evidence assenting to the work of Turhan et al., stating that in a best case scenario, even better performing fault prediction models can be built on carefully selected data from other projects than those induced on in-project data [134]. Cross release defect prediction has also been investigated in several studies, as this practice coincides with the natural life cycle of a software package. E.g. Ostrand et al. discussed a software fault prediction model in AT&T labs that ran for 4 consecutive years, spanning 17 releases [251]. Also Arisholm et al. [12] analyzed a legacy system of a large telco provider across 17 releases while Koshgoftaar et al. [173], also relying on data stemming from a telco provider, considered 4 consecutive releases. They typically found that cross release prediction is feasible; e.g. Koshgoftaar et al. calculated the return on investment of using such a model's predictions to be

as high as 4,266%.

2.3.3 Statistical inference

Upon obtaining empirical results, a statistical framework should be instated to allow for inferencing on the relative performance of different treatments. As empirical results are seldom normally distributed, the nonparametric testing procedure described in Demšar [77] is often applied in such setting. The first step of this procedure consists of the Friedman test [105] which is a nonparametric equivalent of the well known ANOVA test (ANalysis Of Variance). The null hypothesis of the Friedman test states that all treatments are equivalent. The test statistic is defined as:

$$\chi_F^2 = \frac{12P}{k(k+1)} \left[\sum_m AR_m^2 - \frac{k(k+1)^2}{4} \right] \tag{2.16}$$

with AR_m the average rank of treatment $m = 1, \ldots, k$ over P test attempts. Under the null hypothesis, the Friedman test statistic is distributed according to χ_F^2 with $k-1$ degrees of freedom, at least when P and k are big enough ($P > 10$ and $k > 5$). Otherwise, exact critical values should be used based on an adjusted Fisher z-distribution.

Subsequent to the rejection of the Friedman test, an appropriate post-hoc test can be performed. One possibility is the Bonferroni-Dunn test which compares a single treatment to all others, while a Nemenyi test makes a pairwise assessment across treatments; the peculiarities of each of these specific post-hoc tests are presented in Chapters 3 and 4 respectively. Chapter 5 additionally presents an alternative statistical testing procedure which is able to better discriminate between treatments.

2.4 The KDD process by example

In this short case study, we provide an example of the Knowledge Discovery in Databases (KDD) process, which can be defined as the process entailing the collection of raw data and further refining it, resulting in a data set which can be made subject to analysis by ML learning techniques. The outcome can subsequently form the basis for managerial decisions of operational or strategic nature. The KDD process thus involves more than the mere application of some learners on some data set, but also includes aspects such as data preprocessing, and outcome interpretation, be it as a set of statistical analyses or expert user validation.

2.4.1 Context

Customer orientation is essential in many businesses. This case study investigates how data mining techniques can be used to enable a more customer oriented management in the education industry. Students' satisfaction with

training is important for educational institutions in order to attract new students and to retain the current student population, especially during periods of crisis when tuition fees are increased as a last option to ensure the sustainability of educational systems. Moreover, student mobility is steadily increasing as the consequence of a search for high value education, which is also facilitated by a number of educational system reforms such as the Bologna Treaty which ensures the inter-country compatibility of academic degrees in Europe. As a result, the number of students going abroad for their university education in the last three decades has more than quadrupled, from 0.8 million in 1975 to 3.3 million in 2008 [247]. In addition, students tend to take many more countries into consideration. While the USA still attracted about 24% of all international students in 2000, this figure decreased to 18.7% by 2008. Countries such as France, Australia, Italy and Spain on the other hand have experienced a clear increase in the number of international students [247]. The students' heightened mobility and focus on education quality have opened a new globalized competitive arena of opportunities and threats for educational institutions.

Responding to this challenge, educational institutions are collecting increasing amounts of data concerning their education quality by evaluating their programs, courses and training. The collected data can be used to build a mathematical model to identify crucial factors determining the perceived education quality by students and as such, can assist educational institutions in better responding to the student needs and attracting more (international) students. Such a model can be constructed using the various techniques discussed in Section 2.2, including tree/rule based learners which typically construct a piecewise linear model or nonlinear techniques such as techniques based on the principle of structural risk minimization or neural networks [301].

2.4.2 Evaluating class satisfaction by the KDD process

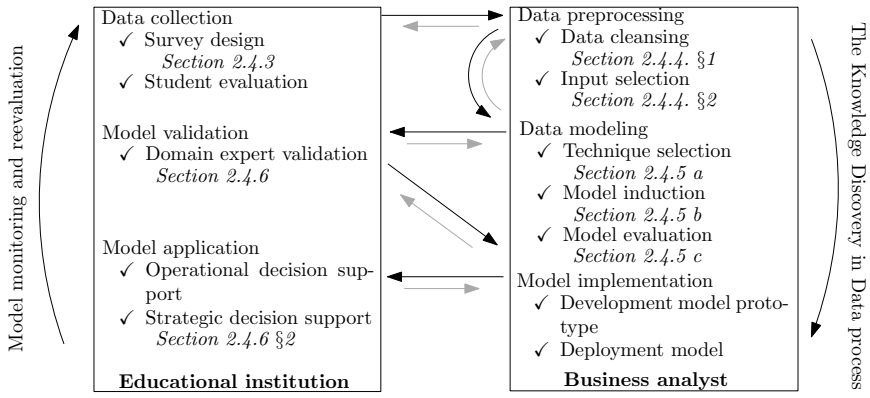


Figure 2.10: Illustration of the KDD process and links to the sections in this dissertation

Knowledge discovery from databases (KDD) is typically a time consuming process which consists of several steps, moving back and forward between the business expert (the management of the educational institution) and the business analyst responsible for the induction of the mathematical models, see Fig. 2.10. The first step in this process consists of gathering relevant data. Here, it is decided what will be measured and at which moment during the academic year. Typically, data is collected by surveys based on the student reactions forms evaluation method, questioning students about their course satisfaction. These evaluation forms are sometimes also known as 'happy sheets', 'smile sheets' or 'reactionnaires' [196], see also Section 2.4.3. As a second step, the resulting data set is preprocessed by first cleansing the data, addressing issues such as missing values, discretisation and recoding of categorical attributes. Following the data cleansing, an input selection procedure can be applied, reducing the number of attributes to learn from, to get an unbiased and relevant set of attributes [34]. The precise data preprocessing procedure is detailed in Section 2.4.4. The third step consists of the actual data modeling. A selection of possible modeling techniques has to be made based on the characteristics of the data set and requirements of the model, e.g. comprehensibility and computational efficiency (Section 2.4.5). Furthermore, the model is evaluated using one or more different metrics. In this case study, both classification accuracy via notch difference diagram and multi class ROC analysis are used, see Section 2.4.5. In a fourth step, the induced model is returned to the management of the cooperating educational institutions to validate whether the model is in line with their expectations and previous knowledge. This requires a basic understandability of the model. Next, a prototype of the model could be developed and deployed into the ICT architecture of the educational institution. However, as the model serves to aid managerial decisions at a strategic level, the implementation of a model for frequent operational use is considered less important in this case study. Finally, the model should be reevaluated on a regular basis to account for changes in the way students perceive the quality of the courses [21]. Note that if the results in one of the steps of the process are not satisfactory, one returns to a previous step in the process.

2.4.3 Data collection

Note that in the paper that serves as the basis of this text, two business educational institutions (IESEG School of Management (Lille) in France and University of Verona in Italy) are analyzed. For brevity however, the focus in this text lies with the results from the IESEG School of Management. Please refer to [73] for additional details.

Institution background

The IESEG School of Management has a 3+2 educational program in place in line with the Bologna treaty framework. Table 2.4 provides an overview of its main characteristics. The school is a private French teaching and research

	IESEG
Educational program	
Length educational program	5 years (3+2)
Type of enrolment	written and oral selection
Internships in program	minimum 10 months
Student population	
Number first year students	500 out of 6000 applicants
Total number of students	2300
International orientation	
International oriented professors	± 75%
Courses taught in English	± 80%

Table 2.4: Overview of IESEG School of Management

institution ('Grand Ecole') with relatively high tuition fees which is part of the Lille Catholic University. Being characterized by a strong international orientation, applicants are subjected to an oral and written selection procedure.

Data set details

At IESEG, students received anonymous paper questionnaires in the time period between the end of a course and the final exam. Although filling out the questionnaire was not compulsory, generally more than 90% of students enrolled for a class returned the questionnaire. This way, about 15.000 class evaluation forms were gathered during the academic year 2007-2008. The questionnaire follows the traditional student reactions form format [196] and aims at measuring Kirkpatrick's first level of students' overall training satisfaction (OTS). In total, the data concerns 199 courses of which 48% belonged to the Bachelor cycle (the first 3 years) and 52% to the M.Sc cycle (the last two years). Hence, the courses' contents covered a wide spectrum of topics. Students had to answer questions by giving a score from 1 (strongly disagree) to 5 (strongly agree). For illustrative purposes, these questions are shown in Table 2.5. Besides these questions, a number of control questions were included in the questionnaire.

Following prior research in the field, we made four aggregated constructs which are believed to influence the overall training satisfaction (OTS): the perceived trainer performance (PTP), perceived training efficiency (PTE), perceived ease of learning (PEL) and perceived usefulness of training (PUT) [110, 210, 273]. The mapping of the individual questions to the 4 constructs is indicated on the right of Table 2.5, while the overall training satisfaction (OTS) was measured using Q17. Furthermore, the IESEG management provided a number of master-data variables about the professor and about the course. As such, a total of 41 attributes was taken into account (37 univariate attributes and 4 aggregated constructs).

Code	Description	
Q0	Were you interested in this topic before the start of the class?	perceived usefulness of training
Q1	Were objectives well explained at the beginning of the class?	
Q2	Was the importance of the class in the educational program clear?	
Q3	Was the knowledge you had before the class appropriate to be able to follow?	
Q4	Was the pace of the class appropriate?	perceived ease of learning
Q7	Was the workload of the class in line with its number of ECTS credits?	
Q8	Was the class understandable?	
Q18	Do you feel like you master the material that was taught?	
Q5	Was the course well structured?	perceived training efficiency
Q6	Was the number of contact hours with the professor sufficient?	
Q9	Was the supporting material helpful to understand the subject?	
Q10	Was the manual well adapted to the class?	
Q12	Was the course atmosphere pleasant?	perceived trainer performance
Q13	Was the course presented clearly?	
Q14	Was the class presented in a convincing and in a dynamic way?	
Q15	Did the teacher follow up on how good you understood the class?	
Q11	Are the evaluation modalities appropriate?	
Q16	Was the teacher sufficiently available outside class hours?	
Q17	Overall, are you satisfied with the class?	
Q19	Would you say you have worked very hard for this class?	
French or international student		Binary
Number of sessions attended		4 categories

Table 2.5: Student reactions form attributes IESEG data set

2.4.4 Data preprocessing

As indicated in Fig. 2.10, an important step before the actual development of an education evaluation decision model, is data preprocessing. To fairly assess the different machine learning techniques, the following procedure is used for both data sets.

Data cleansing

Data collected through questionnaires is rarely directly usable as input for data mining techniques. Hence, prior to data analysis, a number of data cleansing activities need to be performed.

As we intend to evaluate courses instead of the preferences of individual students, the collected data is aggregated per course by taking the average of all students following a specific course. The target (i.e. the overall training satisfaction with a course) is first discretised into four levels, one being the lowest and four the highest. As some techniques are unable to cope with missing values (e.g. logistic regression), these values are replaced by the median of that attribute in case of continuous attributes. Otherwise, in case of categorical attributes, mode imputation is used. If more than 10% of the values are missing, the instances associated with the missing values are removed from the data set. Instances with missing values for the target attribute are also discarded. Categorical attributes with a specific ordering in the categories are encoded using thermometer encoding, see Table 2.6; otherwise, dummy encoding is used.

Original input	Categorical input	Thermometer outputs			
		I_1	I_2	I_3	I_4
age < 30	1	0	0	0	0
$30 \leq$ age < 40	2	0	0	0	1
$40 \leq$ age < 50	3	0	0	1	1
$50 \leq$ age < 60	4	0	1	1	1
age ≥ 60	5	1	1	1	1

Table 2.6: Thermometer encoding

Input selection

An often recurring finding in data mining literature is the difficulty to learn from high dimensional data [75, 230, 325]. Given a number of observations, the search space increases exponentially with the number of available features and several types of solutions have been proposed hereto. One possible approach is to apply a filter prior to data analysis, selecting a subset of most promising features using a heuristic. Such approach has the advantage of being computationally inexpensive and in addition, unlike for instance factor analysis or principal component analysis, the selected features remain unaltered. The lat-

ter is especially important as models built on newly created features consisting of a (linear) combination of other features can be hard to interpret.

In this study, the minimum Redundancy, Maximum Relevance (mRMR) filter, introduced by Peng et al., is adopted [255]. This filter is based on concepts from Shannon's information theory [279] to express the information concerning the target contained in a set of independent attributes using the concept of mutual information. Let $S_m \subset X$ be a feature subset containing m attributes, $x_{(j)}, j = 1, \ldots, m$. The mRMR filter combines both a maximum relevance and a minimum redundancy criterion. The relevance of a subset S is defined as:

$$D = \frac{1}{|S|} \sum_{x_{(j)} \in S} I(x_{(j)}; y) \tag{2.17}$$

while the redundancy between features within the subset S is defined as:

$$R = \frac{1}{|S|^2} \sum_{x_{(j)}, x_{(j')} \in S} I(x_{(j)}; x_{(j')}) \ . \tag{2.18}$$

$I(x_{(j)};x_{(j')})$ represents the mutual information between two random variables $x_{(j)}$ and $x_{(j')}$ and is defined as follows:

$$I(x_{(j)}; x_{(j')}) = \sum_{j,j'} p(x_{(j)}, x_{(j')}) log_2 \frac{p(x_{(j)}, x_{(j')})}{p(x_{(j)}) p(x_{(j')})} \ . \tag{2.19}$$

The mRMR filter finally combines both relevance and redundancy into a single expression:

$$\max \Phi(D, R) = D - R \ . \tag{2.20}$$

Using this filter approach, a subset of the ten best features is selected each time for model construction.

2.4.5 Data modeling

Technique selection

A key requirement to use data mining models in supporting school's management decisions is the aspect of comprehensibility; i.e. one must be able to understand how a model reaches a specific decision. Previous studies from other domains indicated that nonlinear techniques often outperform linear models such as tree/rule based models [23, 200]. This can be attributed to the fact that these techniques allow to construct arbitrary decision boundaries in the output space. However, the resulting models are often difficult to understand; indeed, they are even called 'black box' models [10]. Likewise, in order to better assist the school's management in taking correct decisions, an easy to understand model that gives insight into the factors that are most important to the students is preferred. Furthermore, a comprehensible model would allow to check whether the model is in line with previous assumptions on what drives perceived

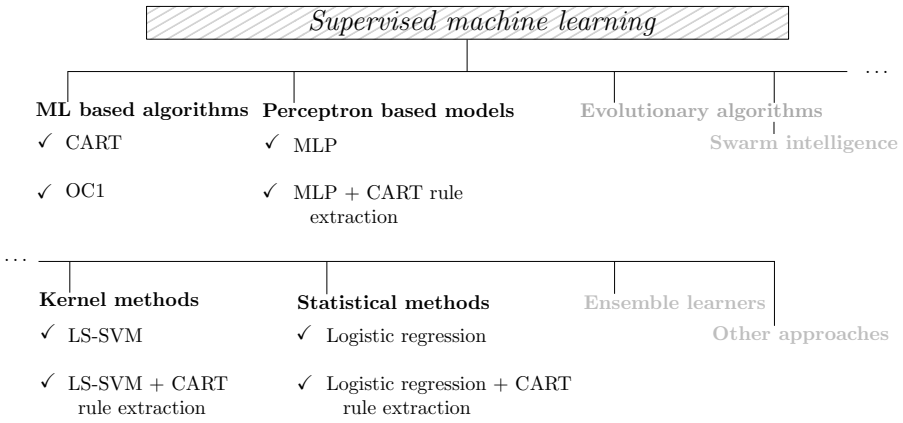


Figure 2.11: Overview applied techniques

education quality [291]. For instance, it was assumed at IESEG that professor availability is a very important driver of student satisfaction. However, if this attribute would not be withheld in our analyses, management could decide to change the policy towards encouraging academic staff to always work in the office, giving them more flexibility. Balancing out both aspects, a selection of ML algorithms in combination with three nonlinear techniques (logistic regression, MLP and SVM) were investigated. Fig. 2.11 shows the positioning of these techniques in the general taxonomy of Section 2.2. Remark that most classifiers can directly be applied to a situation with more than two classes (i.e. a situation with $y_i \in \{1,\ldots, K\}$). However, some techniques such as support vector machines are less suited to cope with more than two classes and alternative procedures to use these techniques should be considered [8].

The following paragraphs relate to the extension of dichotomous learners towards a multi class environment. Also the use of a simple pedagogical rule extraction approach to perform knowledge elicitation from nonlinear models is detailed. We commence however with a discussion on the OC1 learner, which is a multivariate decision tree learner able to induce flexible decision boundaries[5].

Multivariate trees: OC1

In contrast to univariate tree learners such as CART and C4.5, Oblique Classifier

[5]CART, LSSVM and logistic regression are implemented in the Matlab^{TM} environment, www.mathworks.com. In case of LSSVM, an additional open source toolbox is used (LS-SVMlab, http://www.esat.kuleuven.be/sista/lssvmlab). OC1 is implemented using a publicly available toolbox developed for UNIX systems (http://www.cbcb.umd.edu/~salzberg/announce-oc1.html).

1 (OC1) tries to find a hyperplane of the following form:

$$\sum_{j=1}^{n} a_j x_{(j)} + a_{n+1} = 0 \tag{2.21}$$

with $a_j \in \mathbb{R}$ [243]. By allowing additional flexibility in constructing the decision boundary, the resulting trees can potentially be both smaller and more accurate at the expense of decreased comprehensibility [145]. The first algorithm allowing oblique splits was CART with Linear Combinations [43], and OC1 is an algorithm which boosts several improvements over this algorithm. At the heart of OC1 lies the 'randomized perturbation' algorithm. First, OC1 tries to find the best axis parallel split, and will then perturb the resulting hyperplane by considering a single attribute at a time, fixing all others. Once the optimal value for this attribute has been established, the algorithm continues by taking the next attribute until either a maximum number of iterations has been reached or the perturbation fails to result in a decrease of impurity. OC1 also incorporates several mechanisms to avoid getting stuck in a local minimum; e.g. by adding a vector with random values to the coefficients of the hyperplane or taking different initial values. In this case study, both OC1 and CART use the Gini diversity index during tree construction, see Eq. 2.2. This algorithm has previously been applied in several domains [103, 239, 274].

Multi class SVM

By nature, some algorithms (e.g. SVM [322] or AdaBoost [102]) can only discriminate between two classes; however, by adopting a different learning schema, these approaches can still be used in case of more than two classes. Hereto, the multiple classification problem will be decomposed into several binary classification problems. There are many approaches to this decomposition; the most straightforward method is to consider each class separately and compare it to all other classes. This is referred to as a one versus all approach. Hastie and Tibshirani [132] suggested a different approach in which all pairs of classes are compared to each other (one versus one). A third approach suggested in [79] makes use of error correcting output codes in an attempt to make the final classification more robust to binary classification errors.

In this case study, we adopted the use of a one versus one learning schema with majority voting to determine the final classification. A total of $\frac{K!}{2!(K-2)!} = \frac{(K-1)K}{2}$ separate models will thus be estimated, considering each time only the instances of two particular classes. These models can be combined to produce class probability estimates by means of a voting procedure:

$$p_k = 2 \sum_{k,r=1 \ (k \neq r)}^{K} \frac{I(S_k > S_r)}{K(K-1)} \tag{2.22}$$

where p_k is the probability of belonging to class k and $I(x) = 1$ if x is true and 0 otherwise. S_k is the score of belonging to class k predicted by the SVM models.

Note that the least squares support vector machines (LS-SVM) in this study differs to the the SVM discussed in Section 2.2.5 in the sense that it directly solve a set of linear equations [299]. Hereto, equation 2.8 is reformulated as follows.

$$\begin{aligned} \min \quad & \frac{||\mathbf{w}||}{2}^2 + C \sum_{i=1}^{N} \varepsilon_i \\ \text{s.t.} \quad & y_i[\mathbf{w}\phi(\mathbf{x}_i) + b] = 1 - \varepsilon_i, \ i = 1, \ldots, N \end{aligned} \tag{2.23}$$

Rule extraction

It is argued by e.g. Fung et al. that adding even limited explanatory power can positively influence the acceptance of decision models [107]. This is especially important in the presented situation since the induced models are to be validated by the management of the educational institutions, cfr. Section 2.4. It is difficult, if not impossible, to understand the inner workings of nonlinear models such as SVM and neural networks. In an attempt to combine the good comprehensibility of tree/rule based techniques and the classification performance of nonlinear models, a number of rule extraction algorithms have been proposed [22, 26]. These rule extraction techniques are often categorized into pedagogical and decompositional rule extraction techniques [10][6]. Decompositional techniques extract rules at the level of individual components of the black box technique (e.g. at the level of support vectors in a SVM or activation values in a NN) while pedagogical techniques treat the underlying classifier as a black box. The latter approach can thus be used independently of the nonlinear technique.

A simple pedagogical approach is adopted, using the CART algorithm of Section 2.2.1. First, the nonlinear model is trained. Then, we proceed by changing the class labels of the original data set to the value predicted by the nonlinear model. This updated data set is finally presented to the CART algorithm, obtaining a (more comprehensible) CART tree. Due to the equivalence between trees and rule sets [145], this approach can be considered as a pedagogical rule extraction procedure. This approach has previously been used in [27, 220].

Model induction

The data set is randomly partitioned into two disjoint sets, i.e. a training and a test set consisting of respectively 2/3 and 1/3 of the observations. The model is induced on the training set while the independent test set is used for model evaluation. To account for a potential bias introduced by the holdout split, see also Section 2.3.1, this procedure is repeated twenty times as it is argued by Kirsopp et al. that 'ideally more than 20 sets should be deployed' [175].

Furthermore, some techniques have adjustable parameters, also referred to as hyperparameters, which enable a model to be adapted to a specific problem. When appropriate, default values are used based on previous empirical studies

[6] Sometimes also a third category of rule extraction techniques termed 'eclectic' is defined, incorporating elements from both pedagogical and decompositional rule extraction techniques [306].

and evaluations reported in the literature. If no generally accepted default parameter values exist, these parameters are tuned using a grid-search procedure. A set of candidate parameter values is defined and all possible combinations are evaluated by means of a split-sample setup. The models are induced on 2/3 of the training data and the remainder is used as a validation set. The performance of the models for a range of parameter values is assessed using this validation set. The parameter values resulting in the best validation performance are selected and a final model is trained on the full training set. The performance metric used for hyperparameter tuning is the overall classification accuracy.

Model evaluation

The induced education evaluation models are compared to each other both in terms of performance and comprehensibility, which is an often overlooked criterion during model selection.

Classifier performance

Possibly the most straightforward method of measuring classifier performance is the use of an overall accuracy measure such as, in case of a dichotomous target, the percentage of correctly classified (PCC) instances. The PCC can be obtained by taking the sum of all diagonal elements of the confusion matrix and dividing it by the total number of instances. This concept can easily be extended to a multi class context, in which each cell (k, r) in the confusion matrix represents the number of instances belonging to class k which are labeled as class r instances. In this calculation, every instance is assigned to the class associated with the highest class membership value across *all* K classes.

Related to the PCC concept is the use of notch difference diagrams, which originates from the field of credit scoring [9]. The PCC tacitly makes the important assumption of equal misclassification costs for the various different kinds of misclassification, while it is observed that such assumption is in fact only rarely suitable [127]. Assuming a natural ordering in the values taken by the target, classifying a class 1 instance as belonging to class 4 is indeed a more grave error than classifying the same instance to belong to class 2, which results only in a 1 notch difference between the actual and predicted value. Hence, a notch difference diagram which explicitly reports on the different levels of misclassification provides more information than the percentage of correctly classified. Both concepts are illustrated with the SVM classifier on the publicly available Teacher Assistant Evaluation data set, see Fig. 2.13. As this data set was not collected using student reaction forms, it was deemed inappropriate to validate our findings, and is only included to illustrate concepts.

A second tacit assumption made if using the PCC metric to compare different classifiers, is the fact that the class distribution (the class priors) would be constant over time and relatively evenly balanced [260]. While it is theoretically possible to calculate both the correct misclassification costs and the correct class distribution, this is only seldom done [41,56]. If a classifier discriminates between two classes (e.g. between 0 and 1) and produces a continuous

		Predicted class		
		1	**2**	**3**
Actual class	**1**	6	2	1
	2	1	10	2
	3	2	3	10

Table 2.7: Illustration of metrics: confusion matrix SVM model

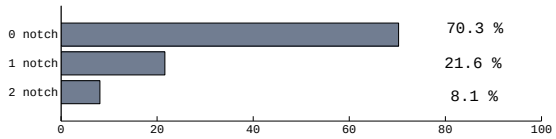


Figure 2.12: Illustration of metrics: notch difference diagram

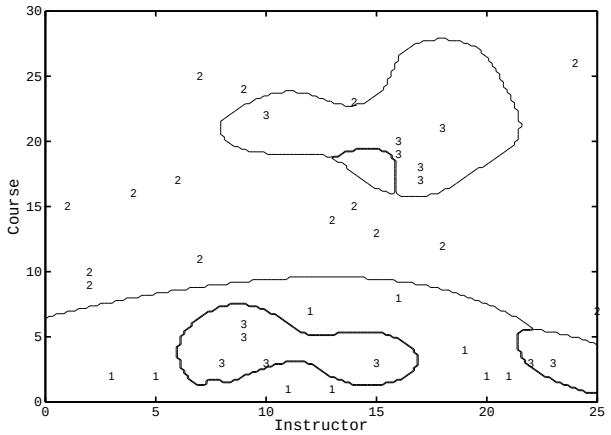


Figure 2.13: Illustration of metrics: classification of SVM classifier

output (e.g. a score indicating the posterior class probability of an instance), another evaluation framework can be used: receiver operating characteristics (ROC) analysis [86]. The use of ROC analysis effectively decouples classifier performance from assumptions concerning classification costs and class distribution [260]. The ROC space is a two dimensional plot of the true positive rate (also referred to as the 'hit rate') on the Y axis against the false positive rate (the 'false alarm rate') on the X axis. Every discrete classification corresponds to a specific true positive and false positive rate, and by varying the threshold on the continuous output of the classifier to predict the class membership, different operating points can be obtained. Plotting these points in ROC space gives rise to the ROC curve which depicts the trade off between benefits (true positives) and costs (false positives). The optimal classifier passes through the point (0,1) while random classification would result in a diagonal line, $x = y$ [92]. In order to better compare different ROC curves, a single scalar value is often used, i.e. the area under the curve (AUC or AUROC). The AUC corresponds to the portion of the area of the unit square that lies underneath the ROC curve, hence the AUC lies always between 0 and 1. As random guessing results in a diagonal line in ROC space with a corresponding AUC of 0.5, any realistic classifier should have an AUC higher than 0.5. The AUC has an interesting statistical property; that is, the AUC is equivalent to the probability that a randomly chosen instance of class 1 would have a higher estimated probability of belonging to class 1 than a randomly chosen instance of class 0. It can be shown that the AUC is equivalent to the Wilcoxon ranked sum test [129]. This allows to calculate the AUC directly from the scores outputted by the classifier, without the need of first plotting the ROC curve. Let $\{\mathrm{f}_1,\ldots,\mathrm{f}_{n_0}\}$ and $\{\mathrm{g}_1,\ldots,\mathrm{g}_{n_1}\}$ be the scores predicted for the n_0 negative and the n_1 positive instances in the test set respectively. Assume a high score indicates a high probability of belonging to the positive class. As the AUC is equivalent to the Wilcoxon ranked sum test, it can be estimated by simply counting the number of pairs of positive and negative instances such that the former has a higher score than the latter:

$$\widehat{AUC} = \hat{A} = \frac{1}{n_0 \times n_1} \sum_{i=1}^{n_0} \sum_{j=1}^{n_1} I_{ij} \tag{2.24}$$

where I_{ij} equals 1 if $\mathrm{g}_j > \mathrm{f}_i$. Alternatively, by combining the scores for the positive and negative instances, and ranking them in ascending order, the estimate of the AUC can be rewritten as:

$$\hat{A} = \frac{S_1 - n_1(n_1+1)/2}{n_0 \times n_1} \tag{2.25}$$

where S_1 is the sum of ranks of all positive instances.

While the AUC was originally only defined in case of binary classification tasks, a number of extensions towards a multi class context have been proposed in the literature. One of the earliest extensions in a three class context was done by Mossman [241], who plotted the true positive rates for each of the three classes while varying the classification threshold. This way, a ROC surface can

be obtained and by taking the volume under the surface, a three class alternative to the AUC can be calculated. Ferri et al. later extended this approach to obtain ROC surfaces for problems with more than 3 classes [97]. A different approach has been taken by Hand and Till, suggesting the calculation of a multi class AUC by combining several binary AUC values, called the 'M index'[127]. This straightforward extension of the binary case is selected in this study as it inherits the good properties of the binary AUC such as independence of class distribution and misclassification costs.

The M index takes the one versus one principle explained in Section 2.4.5, estimating the AUC for each possible pairwise combination of classes. Let k and $r \in \{1,\ldots, K\}$ be two class labels, $k \neq r$ and consider only instances from the test set belonging to either class k or class r. By considering the estimated probabilities of belonging to class k, an estimation of the discriminative potential between class k and class r can be obtained:

$$\hat{A}(k \mid r) = \frac{S_k - n_k(n_k+1)/2}{n_k \times n_r} \tag{2.26}$$

where S_k is the sum of the ranks of all class k test instances and n_k and n_r the number of k class test instances and r class instances respectively. $\hat{A}(k \mid r)$ can be interpreted as the probability that a random instance of class k will have a higher probability of belonging to class k than a random instance of class r. Similarly, $\hat{A}(r \mid k)$ can be defined in terms of probability of belonging to class r. Note that, in the case of more than two classes, $\hat{A}(k \mid r) \neq \hat{A}(k \mid r)$. This problem can be circumvented by taking the average of both values as a measure of separability between both classes:

$$\hat{A}(k,r) = [\hat{A}(k \mid r) + \hat{A}(r \mid k)]/2 \ .$$

The M index which gives the performance of the classifier in separating the K classes is then finally defined as:

$$M = \frac{2}{K(K-1)} \sum_{k<r} \hat{A}(k,r) \ .$$

Classifier comprehensibility

As opposed to classifier performance, there exists no single point metric to quantify the understandability of a model [145]. As indicated in Section 2.3.1, comprehensibility is often regarded as the extent to which there exists a mental fit with the classification model and is thus somewhat subjective by nature [15]. It is argued that the main drivers of this mental fit are representation type and size of the model.

In this study, a number of different representation formats are under investigation (tree/rule based models, linear and nonlinear models). It is believed that symbolic representations (i.e. tree or rule based formats) are more comprehensible as they allow to visualize the resulting models [234]. A second aspect influencing comprehensibility is size of the model; it is argued by e.g. Domingos

et al. that smaller, less complex models are to be preferred [80]. However, the exact relationship between comprehensibility and representation size has, to the best of our knowledge, not been thoroughly investigated so far. We report for each of the tree based models the number of leaf nodes, which is equivalent to the number of rules minus one (a default rule) in the equivalent rule set.

2.4.6 Results

In the first paragraph, the empirical results of applying the different techniques on the IESEG data set are shown. Equally important to the observed performance is the aspect of comprehensibility. As such, the decision tree selected by the management of IESEG is dissected afterwards.

Empirical results

In Table 2.8, the results of the IESEG data set are shown. The best performances are displayed in bold face script. Results found not to be significantly different from the best one at the 1% level with respect to a paired t-test are indicated in italic script while the others are in normal script. The standard deviation over the 20 hold out splits is provided between brackets. It can be observed that logistic regression consistently performs best, while, depending on the data set, nonlinear techniques do not perform significantly worse. Statistical techniques such as logistic regression (logit) result however in mathematical models which may be not fully comprehensible to the institution's management [69]. Comprehensible, univariate decision trees (i.e. CART) on the other hand are less able to capture the underlying relationships in the data sets, indicating the existence of nonlinear relations. By applying rule extraction as described in Section 2.4.5, an increase of the CART decision tree performance can be observed while at the same time reducing the complexity of the resulting model (i.e. the number of rules).

The final univariate decision trees presented to the institution's management are obtained by taking the complete data set as training data [334]. To account for potential overfitting, the results of the final decision trees have been compared to the results obtained on the 20 independent hold out splits. It can be observed that the results of the final decision trees are in line with the out of sample performance reported in Table 2.8, indicating the absence of overfitting. It should also be noted that built-in pruning procedures were used to render the trees less susceptible to overfitting.

Comprehensibility

In total, four univariate decision trees were presented, being the CART tree and the trees obtained as the result of the rule extraction procedure. It was argued by the management that decision models that jointly consider multiple variables (i.e. OC1, neural networks, logit and LS-SVM), while better performing than univariate decision trees, were less feasible as managerial aids. Hence, these decision models were not taken into account directly by the management.

	CART	OC1	Logit	Rulex logit
Classifier performance				
Percentage Correctly Classified (PCC)				
PCC	0.65 (±0.050)	0.63 (±0.050)	**0.75** (±0.044)	0.70 (±0.050)
Notch difference				
1 notch	0.29 (±0.047)	0.28 (±0.042)	0.23 (±0.044)	0.26 (±0.050)
2 notches	0.06 (±0.020)	0.08 (±0.033)	0.02 (±0.015)	0.04 (±0.025)
3 notches	0.003 (±0.008)	0.018 (±0.025)	0 (±0)	0 (±0)
Multi class AUC measure				
M index	0.806 (±0.033)	0.753 (±0.049)	**0.908** (±0.023)	0.820 (±0.022)
Classifier comprehensibility				
Number leaves	11 (±1.34)	5.7 (±4.43)	N/A	8.25 (±1.07)

	MLP	Rulex MLP	SVM	Rulex SVM
Classifier performance				
Percentage Correctly Classified (PCC)				
PCC	*0.71* (±0.044)	0.67 (±0.043)	*0.73* (±0.039)	0.68 (±0.058)
Notch difference				
1 notch	0.26 (±0.046)	0.27 (±0.043)	0.23 (±0.034)	0.28 (±0.062)
2 notches	0.03 (±0.023)	0.06 (±0.024)	0.03 (±0.022)	0.04 (±0.022)
3 notches	0 (±0)	0.005 (±0.01)	0 (±0)	0.001 (±0.003)
Multi class AUC measure				
M index	0.889 (±0.023)	0.784 (±0.039)	0.854 (±0.023)	0.812 (±0.039)
Classifier comprehensibility				
Number leaves	N/A	8.25 (±1.71)	N/A	8.35 (±1.18)

Table 2.8: Comparative results on the IESEG data set with the best result indicated in bold face script and techniques not significantly outperformed at the 1% confidence level indicated in italic script; the other techniques are indicated in normal script

It was observed that the decision tree obtained without prior rule extration (i.e. CART) was the most complex, consisting of 9 leafs while the decision trees obtained after rule extraction were considerably smaller. This trend toward simpler trees by rule extraction can also be observed in Table 2.8. Finally, the management selected the decision tree obtained after rule extraction from support vector machines (i.e. Rulex SVM) as this decision tree was said to be the simplest, still offering an appropriate PCC and fitting with intuition. This is the decision tree discussed in the following paragraphs for IESEG School of Management.

The other decision trees obtained after rule extraction provided little additional insights in the sense that typically similar variables were found to be important. This strengthens the insights gained by the univariate decision trees presented in the following paragraphs.

Interpretation and discussion

The Rulex SVM decision tree was evaluated by the IESEG's management as being very simple, comprehensible and valuable, see Fig. 2.14. Out of the 41 available attributes, only three are retained in the selected decision tree while still maintaining a satisfactory overall performance. Furthermore, the management believed the performance to be of subordinate importance to the knowledge gained concerning which attributes are generally most important. Managers were looking in the first place for what attributes need to be prioritized. Hence, the tree was considered interesting in terms of the attributes selected as well as those left out of the decision tree. Still, the tree at hand was based on data gathered from only one group of stakeholders (the students) and in order to take tactical and strategic decisions, information from other sources, representing other viewpoints, would also be required.

The tree reflects the IESEG student population's attitude towards the perceived education quality. The tree shows that priority should be given to perceived ease of learning (PEL) over other variables. With a low score on PEL, a very good perceived training performance is still not likely to lead to a very good class satisfaction. Concerning the importance of 'perceived ease of learning' in the IESEG tree, several professors said: 'We are becoming more and more aware of the fact that students ask for a smoother learning process. To acquire knowledge, they need guidelines to orient themselves in a world of over-information. We are no longer (only) subject experts for them; they find all kinds of information online; we have to be learning-facilitators'. We note that this finding should not be generalized to just any educational institution. More specifically, for countries where Internet penetration is low, students may particularly see the professor as a source of knowledge. This may for example be the case in India, where the Internet penetration rate is only 8.4% (whereas it is 69.5% in France and 49.2% in Italy). Also, the importance of the aspect of 'perceived trainer performance' (PTP) was acknowledged by academic staff, stating 'I often read on my evaluation form that they like my class because it is dynamic, although they don't necessarily like the topic of my class'. IESEG's management agreed

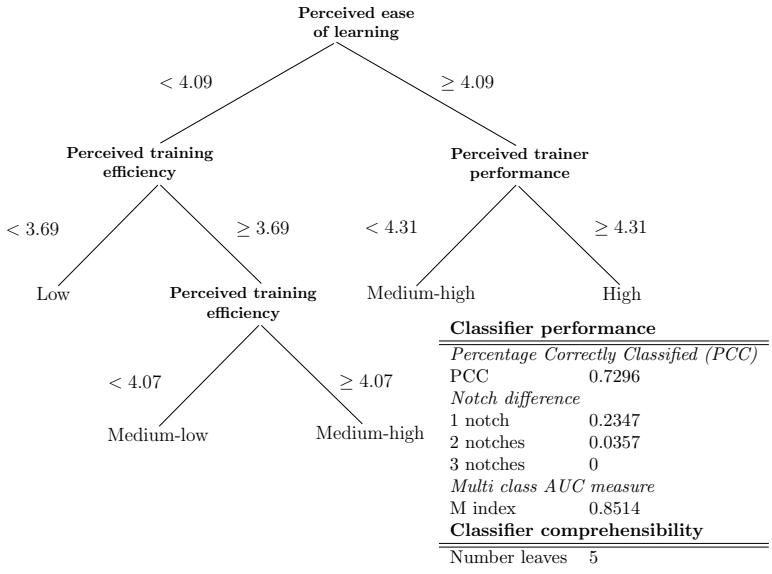


Figure 2.14: Univariate decision tree obtained by rule extraction from the SVM model

that 'our students have many contact hours and then it is understandable they need to be animated a bit more'. While it may seem obvious that this variable is included in the tree of an educational institution, it is not so obvious in light of the fact that several other variables were not included in the tree (see below). The tree thus shows that this PTP deserves priority over other variables. However, it is equally important to notice that PTP only deserves this priority in case the PEL is considered high (i.e., well above the average of 3.95). In many cases, the PEL will get a lower score than 4.09 (the threshold), in which case 'perceived training efficiency' (PTE) is more important. The inclusion of the perceived training efficiency also was understandable to the management: 'We sometimes get complaints from students that they have to buy expensive books, which may be very good references in general, but which are hardly used in class'. This issue might be especially applicable to IESEG, as this school offers many short classes (of 18 contact hours) in which professors try to provide students with a decent, comprehensive manual that usually cannot be treated entirely within the time frame of the class. Consequently, professors were questioned about their manual use and motivated to reconsider the manual, in light of the importance of this variable for the student's satisfaction with their class.

The management's conclusion was that for classes with complex subjects (which are likely to lead to a lower PEL), it is very important for the professor to ensure a good PTE (e.g. by improving teaching materials) while for less complex subjects a dynamic, empathic behavior of a professor and a good atmosphere are more important.

While management agrees on the tree's structure, they admit it was upfront hard to put everything in perspective. In order to prioritize PEL, management now for example follows up on budgeted workloads, which should be in line with the number of ECTS credits while the workload was in the past primarily left to the discretion of individual professors. Moreover, given the importance of PTE, management now tries to give newly hired professors a lower teaching load so they have more time to develop/choose appropriate manuals and supporting material. Finally, in the hiring process more attention is paid to the teaching style of teachers. While the school used to concentrate on the content that the professor was mastering (related to his/her research), the school introduced additional requirements with respect to teaching style to the extent the school 'would rather not hire a new, fixed professor this year than hiring a professor which is little dynamic'.

Several attributes were not retained in the model. Some of those particularly drew management's attention. In the past the management of the school heavily encouraged professors to work at the office rather than at home. The main reason for doing this was to guarantee a high level of service to students given that tuition fees are relatively high at this school. Moreover, the distance between professors and students at IESEG is significantly smaller than at many other French institutions, thanks to the international atmosphere at the school and the 'open door policy' as portrayed by the dean. A common belief was therefore, that students would find professor availability important. However, the attribute reflecting this was not retained in the tree (and actually in none of the trees that were derived in earlier versions). Management saw an opportunity to act upon a comment that was often ventilated by professors: in order to enhance their research activities, more flexibility was given to academic staff by formally authorizing them to work at home one day per week. Another element that was considered interesting is the fact that nationality plays no role. At IESEG, more than 75% of professors are non-French because the majority of courses are delivered in English. This aspect seems irrelevant for students and there is thus no need for (positive or negative) discrimination in terms of nationality from a teaching viewpoint. Overall, the assessment of the decision tree led to a win-win situation at the school. First, students can get better service because professors know what aspects really matter and deserve additional attention. Secondly, professors get a better 'life quality' and probably better research output thanks to the introduced flexibility.

The board noticed that the fact that some attributes were not retained does not mean they are not important. The viewpoint of other stakeholders is important too. For example, if research output is important for the institution, aspects such as having a PhD (which was not retained in the tree) may be important from that point of view.

Threats to validity

Upon performing empirical analyses, it is important to identify the potential threats to both internal and external validity of the obtained results.

Internal validity in this context refers to the extent to which insights gained from the student population are valid and correct. Since the data collected in this study stems from student reactions forms, one potential threat to the internal validity is the unknown effect of sampling bias which may be present in our data set. However, great care was put into the collection of these forms in both educational institutions considered in this study, cfr Section 2.4.3. As such, a large number of reactions forms were collected at both institutions, leading to a high response rate and hence a minimal threat of sampling bias. A second possible threat to the internal validity lies in the way the questionnaires were conceived; the questionnaires have been designed with the specific educational literature in mind. It has been illustrated that data collected using such questionnaires are reliable and valid [14, 109, 171]. Moreover, all questions have invariantly been formulated in a positive manner and scored on a similar scale (i.e. from 1 to 5). A third threat to internal validity is the use of aggregated constructs, see Section 2.4.4. For each of these constructs, the Cronbach's alpha was calculated and found to be appropriate and thus these constructs were used during the analyses.

External validity refers to the extent to which the results of this study can be generalized to other student populations and/or other research settings. Given the differences existing between educational institutions, we do not consider our trees to have external validity. This is illustrated by the fact that the derived trees differed to those of other educational institutions [73]. It is therefore believed that the managerial insights outlined earlier may not be generalized toward other educational institutions as the student population might be characterized by other priorities. Especially the fact that only European educational institutions have been investigated is noteworthy in this respect, and the degree to which the results generalize to other (European) educational institutions is left as a topic for future research.

2.4.7 Case study conclusions

In this study, we investigated the construction of comprehensible data mining models to support strategic decisions taken by the management of educational institutions and showed the valued results for two educational institutions. The selection of techniques encompassed two different decision tree learners, cumulative logistic regression as well as two state of the art nonlinear data mining techniques. While it was found that cumulative logistic regression performed best, the management of the educational institutions cooperating in this study preferred a symbolic representation format such as univariate decision trees due to their excellent comprehensibility. The selected decision tree allowed to identify the main drivers of overall class satisfaction, enabling the management to provide a set of clear guidelines towards academic staff. It is noted that the organizations differ significantly in terms of mission and governance and therefore, rather than taking the trees mentioned in this study as a given, educational institutions are advised to use our approach to derive a tree fitting to their own student society, faculty and culture.

2.5 Chapter summary

This chapter presented a general ML taxonomy, supplemented with a discussion on the related aspect of model validation. Next, a case study relating to student satisfaction was detailed, which served as an example of the concepts treated in the first part of the chapter. This study also encompassed the different steps of the Knowledge Discovery in Databases (KDD) process which implicitly underpins the next chapters of this dissertation.

I have had my results for a long time: but I do not yet know how I am to arrive at them.

Carl Friedrich Gauss, 1777 – 1855

3

Quantifying software development effort

In the third chapter of this dissertation, we deal with the topic of software effort estimation. Definitely not a trivial one, the question of how to come up with accurate estimates has troubled many researcher. During the last 3 decades and more, a library of articles has been written to address this question, without offering a univocal recommendation of which technique is most suited. This chapter addresses this issue by reporting on the results of a large scale benchmarking study, comparing a multitude of different techniques. Furthermore, the aspect of feature subset selection by using a generic backward input selection wrapper is investigated. Subjecting the results to rigorous statistical testing, our findings suggest ordinary least squares regression in combination with a logarithmic transformation to be most appropriate. Another key insight is that by selecting a subset of highly predictive attributes such as project size, development, and environment related attributes, typically a significant increase in estimation accuracy can be obtained.

This chapter is based on the following paper

- K. Dejaeger, W. Verbeke, D. Martens and B. Baesens, "Data mining techniques for software effort estimation: a comparative study," *IEEE Transactions on Software Engineering*, 38 (2): 375–397, 2012.

3.1 Introduction

Resource planning is considered a key issue in a production environment. In the context of a software developing company, the different resources are, amongst others, computing power and personnel. In recent years, computing power has become a subordinate resource for software developing companies as it doubles approximately every 18 months, hereby costing only a fraction compared to the late 60's. Personnel costs are however still an important expense in the budget of software developing companies. In light of this observation, proper planning of personnel effort is a key aspect for these companies. Due to the intangible nature of the product 'software', software developing companies are often faced with problems estimating the effort needed to complete a software project [305]. As was already indicated in Section 1.3.2, there has been strong academic in-

terest in this topic, assisting the software developing companies to tackle the difficulties experienced when estimating the software development effort. In this field of research, the required effort to develop a new project is estimated based on historical data from previous projects. This information can be used by the management to improve the planning of personnel, to make more accurate tendering bids, and to evaluate risk factors [165]. Recently, a number of studies evaluating different techniques have been published, without however offering a univocal conclusion. Menzies et al. disseminated in an editorial the origin of this conclusion instability, listing amongst others differences in preprocessing and data sources, and the preference of researchers towards specific techniques as sources of conclusion variance and conclusion bias respectively [232]. In this chapter, an overview of the existing literature is presented, complementing the overview of Chapter 1. Furthermore, thirteen techniques, able to induce a heterogenous set of models, are investigated. This selection includes tree/rule based models (M5 and CART), linear models (ordinary least squares regression with and without various transformations, ridge regression, and robust regression), non-linear models (MARS, least squares support vector machines, multi layered perceptron neural networks, radial basis function networks), and a lazy learning based approach which does not explicitly construct a prediction model, but instead tries to find the most similar past project. Each technique is applied to nine data sets within the domain of software effort estimation. From a comprehensibility point of view, a more concise model (i.e. a model with less inputs) is preferred. Therefore, the impact of a generic backward input selection approach is assessed.

The remainder of this chapter is structured as follows.
Section 3.2 presents an overview of the literature concerning software effort estimation.
Section 3.3 discusses the applied techniques.
Section 3.4 reflects upon the data sets, evaluation criteria and the statistical validation.
Section 3.5 elaborate on our findings regarding techniques and generic backward input selection procedure.
Section 3.6 finally wraps up the chapter by providing a general conclusion.

3.2 Related research

In brief, software effort estimation draws upon the information of past projects (e.g. size, programming language, and experience of development team together with the associated development effort), to learn a mathematical model which relates the characteristics of a new software project to an estimate of the development time. In Section 1.3.2, we already discussed the three main approaches to software effort estimation (i.e. expert driven estimation, formal models and data mining). As this chapter revolves around the question which data mining technique is most suited to software effort prediction, the focus in this research

EM_i	Description	Impact
acap	Analysts capability	**Positive impact**
pcap	Programmers capability	*Increase results in a decrease of effort*
aexp	Application experience	
modp	Modern programming practices	
tool	Use of software tools	
vexp	Virtual machine experience	
lexp	Language experience	
sced	Schedule constraint	*Convex relation*
stor	Main memory constraint	**Negative impact**
data	Data base size	*Increase results in an increase of effort*
time	Time constraint for cpu	
turn	Turnaround time	
virt	Machine volatility	
cplx	Process complexity	
rely	Required software reliability	

Table 3.1: Overview of the Cocomo I multipliers

overview will be on the latter approach. It is however worth pointing out that many practitioners rely on expert judgement or formal models. Both these approaches have there own strengths and drawbacks; e.g. Jørgensen noticed that 'There are situations where expert estimates are more likely to be more accurate... Similarly, there are situations where the use of models may reduce large situational or human biases'.

Following the comparison of data mining techniques, the applicability of the (formal) Cocomo model is further discussed in Section 3.5.2. In Eq. 1.2, the Cocomo II post architecture model was presented; however, as only data pertaining to the Cocomo I model was obtained, this older variant is assumed in this chapter. The Cocomo I model takes the following form:

$$\text{Effort} = a \times \text{Size}^b \prod_{i=1}^{15} \text{EM}_i \tag{3.1}$$

where a and b are two factors that can be set depending on the details of the developing company and EM_i is a set of effort multipliers, see Table 3.1. As data sets typically contain insufficient projects to calibrate all parameters, only a and b are adapted to reflect the development environment. Data for this model is collected making use of specific questionnaires which are filled in by the project manager and as such requires a considerable effort from the business. The Cocomo II update concerns new software development trends such as outsourcing and multiplatform development [39].

More recently, formal models are being superseded by a number of data intensive techniques originating from the data mining literature [163]. These

include various regression techniques which result in a linear model [98, 275], non-linear approaches like neural networks [98], tree/rule based models such as CART [45, 46], and lazy learning strategies (also referred to as 'case based reasoning' or 'analogy') [180, 181, 204]. Data mining techniques typically result in objective and analyzable formulae and are not limited to a specific set of attributes as is the case with formal models such as Cocomo I. Several studies assessed the applicability of data mining techniques to software effort estimation. However, most of these studies evaluate only a limited number of modeling techniques on a particular, sometimes proprietary, data set which naturally constrains the generalizability of the observed results. Some of these studies also lack a proper statistical testing of the obtained results or evaluate models on the same data as used to build the models [177].

A non-exhaustive overview of the literature concerning the use of various machine learning approaches for software effort estimation is presented in Table 3.2. This table summarizes the applied modeling techniques, the data sets that are used, and the empirical setup for a number of studies and shows that a large number of modeling techniques have been applied in search for the most suitable technique for software effort estimation, both in terms of accuracy and comprehensibility.

For example, Finnie et al. [98] compared Artificial Neural Networks (ANN) and Case Based Reasoning (CBR) to Ordinary Least Squares regression (OLS regression). It was found that both artificial intelligence models (ANN and CBR) outperformed OLS regression and thus can be adequately used for software effort estimation. However, these results were not statistically tested.

Briand et al. [45], while performing a comparison between OLS regression, stepwise ANOVA, CART, and CBR, found that case based learning achieved the worst results while CART performed best; however, the difference was not found to be statistically significant. In a follow up study using the same techniques on a different data set, different results were obtained, i.e. stepwise ANOVA and OLS regression performed best [46].

Shepperd et al. [283] reported that CBR outperforms regression, yet in a study by Myrtveit and Stensrud [244], these results were not confirmed. However, Shepperd et al. used a different regression approach (without a log transformation) to the latter study and opted not to split the data set in a training and test set.

It should be noted that the results of studies are often difficult to compare due to different empirical setup and data preprocessing, possibly leading to contradictory results [114]. Hence the issue of which modeling technique to use for software effort estimation remains an open research question. Another issue in software engineering is the fact that estimation techniques are typically applied to small data sets and/or data sets which are not publicly available, rendering these studies irreproducible [208]. Additionally, inferences are often made on only one or two data sets and as is noted by Kitchenham [177] 'One of the main problems with evaluating techniques using one or two data sets is that no-one can be sure that the specific data sets were not selected because they are the ones that favor the new technique'.

Remark that, although a large variety of techniques are available, see e.g. Section 2.2, expert driven estimation methods are still frequently applied in a business setting. Evidence from other domains suggests that both data mining and formal models could provide more accurate estimates than expert driven estimation methods. Often cited strong points of an analytical approach are consistency (provided with the same input, a model will always reach the same conclusion) and the fact that such models posses the ability to correctly assess the impact of different inputs [68]. This conjecture was however not confirmed by studies in the domain of software effort prediction [160]. Jørgensen stated that 'The use of models, either alone or in combination with expert judgement may be particulary useful when i) there are situational biases that are believed to lead to a strong bias towards overoptimism; ii) the amount of contextual information possessed by the experts is low; and iii) the models are calibrated to the organization using them.' [161]. Other research confirmed that whether expert driven methods perform significantly better or worse than an analytical oriented approach remains a point of debate [137, 164, 242].

Table 3.2: Literature overview of the application of data mining approaches for software effort estimation

Authors	**Title & *Journal***	**Year**	**What?**	**Techniques**	**Data set & # *proj. - # attr.***	**Metrics - Empirical setup**
K. Srinivasan, D. Fisher	Machine Learning Approaches to Estimating Software Development Effort *IEEE Transactions on Software Engineering*	1995	Two alternative data mining techniques are compared to formal models	Artificial neural networks, CART	Kemerer *15 proj. - 6 attr.* Cocomo81 *63 proj. - 16 attr.*	MMRE, R^2 - Holdout
M. Shepperd, C. Schofield	Estimating Software Project Effort Using Analogies *IEEE Transactions on Software Engineering*	1997	Investigation of data mining techniques as an alternative to formal models	OLS regression, case based reasoning	DPS database *24 proj. - 5 attr.* Desharnais *77 proj. - 9 attr.* Finnish *38 proj. - 29 attr.* Kemerer *15 proj. - 2 attr.*	MMRE, $Pred_{25}$ - Cross validation Mermaid *28 proj. - 17 attr.* Telecom data sets *18-33 proj. - 1-13 attr.*
G. Wittig, G. Finnie	Estimating software development effort with connectionist models *Information and Software Technology*	1997	Analysis of backpropagation neural networks for software effort prediction	Artificial neural networks	Simulated data *1000 proj. - 3 attr.* Desharnais *81 proj. - 9 attr.*	MMRE, $Pred_{25}$ - Holdout
G. R. Finnie, G. E. Wittig, J.-M. Desharnais	A Comparison of Software Effort Estimation Techniques: Using Function Points with Neural Networks, Case Based Reasoning and Regression Models *The Journal of Systems and Software*	1997	Comparison of three different estimation techniques using function points as a size estimate	OLS regression, artificial neural networks, case based reasoning	ASMA data set *299 proj. - 1 attr.*	MMRE, $Pred_{25}$ - Holdout
L. Briand, K. El Emam, D. Surmann, I. Wieczorek, K. Maxwell	An Assessment and Comparison of Common Software Cost Estimation Modeling techniques 21^{st} *International Conference on Software Engineering (ICSE '99)*	1999	Investigation of the performance of various software effort estimation techniques both for company-specific and multi-organizational data	OLS regression, ANOVA, CART, case based reasoning	Experience *206 proj. - 19 attr.*	MMRE, MdMRE, $Pred_{25}$ - Cross validation and matched pairs Wilcoxon signed rank test

I. Myrtveit, E. Stensrud	A Controlled Experiment to Assess the Benefits of Estimation with Analogy and Regression Models *IEEE Transactions on Software Engineering*	1999	Experiment of software effort estimation by experts using case based reasoning and regression techniques as estimation tool	OLS regression, case based reasoning	COTS data set *48 proj. - 10 attr.*	MMRE, MdMRE - Paired t-test and Wilcoxon rank sum test
L. Briand, T. Langley, I. Wieczorek	A replicated assessment and Comparison of Common Software Cost Modeling Techniques 22^{nd} *International Conference on Software Engineering (ICSE '00)*	2000	In line with their previous study, a similar set of techniques are applied to a different data set	OLS regression, ANOVA, CART, case base reasoning	ESA *166 proj. - 10 attr.*	MdMRE, $Pred_{25}$- Cross validation and matched pairs Wilcoxon signed rank test
C. Burgess, M. Lefley	Can genetic programming improve software effort estimation? A comparative evaluation *Information and Software Technology*	2001	Evaluation of genetic programming algorithms for software effort prediction	Artificial neural networks, case based reasoning, genetic programming	Desharnais *81 proj. - 9 attr.*	MMRE, BMMRE, $Pred_{25}$, correlation, AMSE - Holdout
K. Strike, K. El Emam, N. Madhavji	Software Cost Estimation with Incomplete Data *IEEE Transactions on Software Engineering*	2001	Study on the effects of missing data in the field of software effort prediction	OLS regression	Experience *206 proj. - 16 attr.*	MdMRE, Pred, R^2_{adj} - Holdout
B. Kitchenham, S. L. Pfleeger, B. McColl, S. Eagan	An empirical study of maintenance and development accuracy *The Journal of Systems and Software*	2002	Comparison of regression techniques using a real-life data set from a single company	OLS regression, median regression	CSC data set *144 proj. - 10 attr.*	MMRE, Pred - Holdout and paired t-test
A. Heiat	Comparison of artificial neural network and regression models for estimating software development effort *Information and Software Technology*	2002	Comparison of the performance of neural networks versus regression approaches	OLS regression, artificial neural networks, RBF networks	DPS database *24 proj. - 5 attr.* Kemerer *15 proj. - 6 attr.* Hallmark *28 proj. - ? attr.*	MMRE, R^2 - Holdout and paired t-test
P. Sentas, L. Angelis, I. Stamelos, G. Bleris	Software productivity and effort prediction with ordinal regression *Information and Software Technology*	2005	Analysis of a novel regression technique for software effort prediction	OLS regression, ordinal regression	Cocomo81 *63 proj. - 22 attr.* Maxwell *62 proj. - 14 attr.* ISBSG R7 *52 proj. - 10 attr.*	MMRE, $Pred_{25}$ - Holdout

T. Menzies, Z. Chen, J. Hihn, K. Lum	Selecting Best Practices for Effort Estimation *IEEE Transactions on Software Engineering*	2006	Investigation of deviations exhibited by different techniques for software estimation	OLS regression	Cocomo81 *63 proj. - 16 attr.* NASA *93 proj. - 16 attr.* COCOMOII *161 proj. - 18 attr.*	MMRE, $Pred_{30}$, Correlation - Holdout
M. Auer, A. Trendowicz, B. Graser, E. Hounschmid, S. Biffl	Optimal Project Feature Weights in Analogy-Based Cost Estimation: Improvement and Limitations *IEEE Transactions on Software Engineering*	2006	Investigation on the impact of different weighting schemata for case based reasoning	Case based reasoning	ESA *44 proj. - 11 attr.* Experience *43 proj. - 18 attr.* Desharnais *77 proj. - 9 attr.* DPS database *24 proj. - 6 attr.* Kemerer *15 proj. - 3 attr.*	MMRE, $Pred_{25}$, Variance - Cross validation
N.-H. Chiu, S.-J. Huang	The adjusted analogy-based software effort estimation based on similarity distances *The Journal of Systems and Software*	2007	The application of a genetic algorithm to derive an estimation based on the retrieved cases	OLS regression, artificial neural networks, CART, case based reasoning	DPS database *23 proj. - 5 attr.* Abran *21 proj. - 6 attr.*	MMRE, MdMRE, $Pred_{25}$ - Cross validation
J. Li, G. Ruhe, A. Al-Emran, M. Richter	A flexible method for software effort estimation by analogy *Empirical Software Engineering*	2007	Introduction of AQUA, a case based reasoning algorithm and subsequent comparison with other techniques	Case based reasoning	USP05 *197 proj. - 15 attr.* ISBSG R4 *2024 proj. - 18 attr.* Kemerer *15 proj. - 6 attr.* Leung02 *24 proj. - 16 attr.*	MMRE, $Pred_{25}$, Strength, Support - Cross validation Mends03 *34 proj. - 8 attr.*
S.-J. Huang, N.-H. Chiu, L.-W. Chen	Integration of the grey relational analysis with genetic algorithm for software effort estimation *European Journal of Operational Research*	2008	Building software effort estimation models using grey relational analysis to counteract incomplete observations	Artificial neural networks, CART, case based reasoning, Grey relational analysis	Cocomo81 *63 proj. - 16 attr.* DPS database *23 proj. - 5 attr.*	MMRE, $Pred_{25}$ - Holdout

K. Kumar, V. Ravi, M. Carr, N. Kiran	Software development cost estimation using wavelet neural networks *The Journal of Systems and Software*	2008	Investigation of wavelet neural networks in the context of software effort estimation	Artificial neural networks, RBF networks, wavelet neural networks, support vector machines	DPS database *23 proj. - 4 attr.* Abran *21 proj. - 6 attr.*	MMRE, MdMRE, $Pred_{25}$ - Cross validation
H. Park, S. Baek	An empirical validation of a neural network model for software effort estimation *Expert Systems with Applications*	2008	The evaluation of a neural network model for software effort estimation for function point based data sets	OLS regression, artificial neural networks	Korean IT data set *148 proj. - 18 attr.*	MMRE - Holdout
Y. F. Li, M. Xie, T. N. Goh	A study of mutual information based feature selection for case based reasoning in software cost estimation *Expert Systems with Applications*	2009	Application of a mutual information based filter approach to select attributes for case based reasoning	Case based reasoning	Desharnais *77 proj. - 8 attr.* Maxwell *62 proj. - 25 attr.*	MMRE, MdMRE, $Pred_{25}$ - Cross validation
S. Koch, J. Mitlöhner	Software project effort estimation with voting rules *Decision Support Systems*	2009	Application of social choice models to the domain of software effort estimation using a genetic algorithm to determine weights	Social choice models	DPS database *24 proj. - 5 attr.* ERP data set *31 proj. - 7 attr.* Cocomo81 *63 proj. - 15 attr.*	MMRE, MdMRE, $Pred_{25}$ - Cross validation

3.3 Techniques

As this study aims to identify the best data mining approach to estimate software effort, a broad selection of learners is under consideration, see Fig. 3.1[1]. These techniques were selected on the premise of achieved results in this or other (regression) settings. Also the computational cost was factored in during technique selection, eliminating learners characterized by high computational loads, in order to keep the scale of the benchmarking experiment more manageable.

Chapter 2 already introduced most learners, and where appropriate, the following paragraphs provide additional details. A selection of statistical methods and data mining techniques (MARS, CART and MLP; cfr infra) is illustrated on the Desharnais data set in Fig. 3.2 (a) to (f) by plotting project size (PointsNonAdjust) against effort in man hours.

3.3.1 Statistical methods

OLS and transformations

Ordinary Linear regression (OLS) forms the cornerstone of many alternative analysis techniques, some of with are discussed below. Detailed in Section 2.2.6, the dependent and independent attributes can additionally be subjected to various transformations to improve the fit of the linear function learned by this technique. More specifically, both a log transformation and a Boc Cox transformation are considered in this study.

Log + OLS
Typically, both dependent and independent attributes in the field of software effort prediction can be heavily skewed; e.g. skewness$(e)_{Desharnais} = 1.97$ and skewness$(e)_{ESA} = 4.89$. Skewness, γ_s, is defined as:

$$\gamma_s = \frac{\mu_3}{\sigma^3} \tag{3.2}$$

where μ_3 is the third moment of the mean, and σ the standard deviation. A normal distribution has a skewness of zero while a positive (negative) skewness indicates a larger number of smaller (bigger) projects. By applying a log transformation to the data, the residuals of the regression model become more homoscedastic, and follow more closely a normal distribution [133]. This transformation is also used in previous studies [45, 46]. Both OLS and Log + OLS are illustrated in Fig. 3.2 (a).

BC + OLS
The Box Cox (BC) transformation is a power transformation which corrects for

[1]The techniques are implemented in Matlab, www.mathworks.com, and Weka, www.cs.waikato.ac.nz/ml/weka. Additionally, open source toolboxes were used in case of least squares support vector machines (LS-SVMlab, www.esat.kuleuven.be/sista/lssvmlab) and MARS (ARESLab, www.cs.rtu.lv/jekabsons/regression.html).

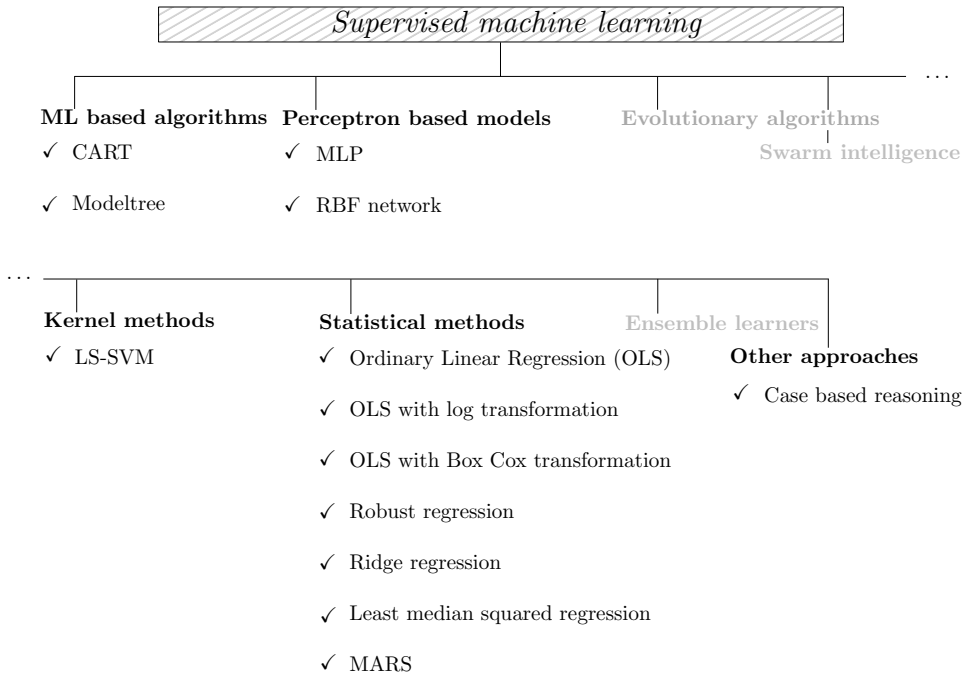


Figure 3.1: Overview applied techniques

discontinuities, e.g. when the transformation parameter, γ, is zero [40]. The BC transformation is defined as:

$$e_i^* = \begin{cases} \frac{(e_i^\gamma - 1)}{\gamma} & \gamma \neq 0 \\ \log e_i & \gamma = 0 \end{cases} \quad (3.3)$$

The transformation parameter is obtained by a maximum likelihood optimization. The BC transformation is an alternative to the log transformation serving similar goals. A BC transformation will resolve also problems related to non-normality and heterogeneity of the error terms [272].

To the best of our knowledge, the BC transformation has not been previously applied in the context of software effort estimation.

Robust regression

Robust regression (RoR) is an alternative to OLS regression with the advantage of being less vulnerable to the existence of outliers in the data [141]. RoR is an application of Iteratively Reweighted Least Squares (IRLS) regression in which the weights $\omega_i^{(t)}$ are iteratively set by taking the error terms of the previous iteration, $\epsilon_i^{(t-1)}$, into account. In each iteration, RoR will minimize the following objective function:

$$\min \sum_{i=1}^{N} \omega_i^2 \epsilon_i^2 \quad (3.4)$$

From this equation, it can be easily seen that OLS regression can be in fact considered as a special case of RoR [237].

Multiple possible weighting functions exist, the most commonly applied being Huber's weighting function and Tukey's bisquare weighting function. In this study, the Tukey's bisquare weighting function is used:

$$\omega_{bisquare} = \begin{cases} (1 - \zeta^2)^2 & \text{if } |\zeta| < 1 \\ 0 & \text{otherwise} \end{cases} \quad (3.5)$$

ζ is the normalized error term and is computed as a function of $\hat{\sigma}$, the estimated standard deviation of the error terms, and a tuning constant, τ, which penalizes for the distance to the regression function. The first iteration consists of an OLS regression since the weights depend on the previous iteration.

RoR is a technique that has been previously applied in the field of software effort estimation [150].

Ridge regression

Ridge regression (RiR) is an alternative regression technique that tries to address a potential problem with OLS in case of highly correlated attributes. OLS regression is known to be BLUE (Best Linear Unbiased Estimator) if a number of conditions are fulfilled, e.g. the fact that $\mathbf{X'X}$ should be non singular. In reality however, different variables are often highly correlated, resulting in a

near singular $\mathbf{X}'\mathbf{X}$ matrix. This will result in unstable estimates in which a small variation in $\mathbf{e}$, the dependent variable, can have a large impact on $\hat{\boldsymbol{\beta}}$.

RiR addresses this potential problem by introducing a so-called ridge parameter, δ [140]. The introduction of the ridge parameter will yield the following estimator of $\boldsymbol{\beta}$:

$$\hat{\boldsymbol{\beta}}_\delta = (\mathbf{X}'\mathbf{X} + \delta I_n)^{-1}(\mathbf{X}'\mathbf{e}) \tag{3.6}$$

where I_n represents the identity matrix of rank n.

To the best of our knowledge, this technique has not been applied before within the domain of software effort estimation.

Least Median of Squares regression

Least Median of Squares regression (LMS) is an alternative to robust regression with a breakdown point $\kappa^* = 50\%$ [269]. The breakdown point κ^* is the smallest percentage of incorrect data that can cause an estimator to take on aberrant values [123]. This breakdown point is 0% for all the other regression techniques considered in this study, indicating that extreme outliers could have a detrimental influence for these techniques. The LMS will optimize the following objective function:

$$\min median(\epsilon_i^2) \tag{3.7}$$

where ϵ_i is the error associated with the ith observation. Although LMS regression is known to be inefficient in some situations [123], this technique has been applied in different domains. However, to the best of our knowledge, the LMS regression has not been applied to the estimation of software effort.

Multivariate Adaptive Regression Splines

Multivariate Adaptive Regression Splines (MARS) is a novel technique introduced by Friedman [104]. MARS is a nonlinear and non-parametric regression technique exhibiting some interesting properties like ease of interpretability, capability of modeling complex nonlinear relationships, and fast model construction. It also excels at capturing interactions between variables and therefore is a promising technique to be applied in the domain of effort prediction. MARS has previously been successfully applied in other domains including credit scoring [197] and biology [89].

MARS fits the data to the following model:

$$e_i = b_0 + \sum_{k=1}^{K} b_k \prod_{l=1}^{L} h_l(x_{i(j)}) \tag{3.8}$$

where b_0 and b_k are the intercept and the slope parameter respectively. $h_l(x_{i(j)})$ are called hinge functions and are of the form $max(0, x_{i(j)} - b)$ in which b is called a knot. It is possible to model interaction effects by taking the product of multiple hinge functions. Hence, this model allows for a piecewise linear function by adding multiple hinge functions.

The model is constructed in two stages. In a first stage, called the forward pass, MARS starts from an empty model and constructs a large model by adding hinge functions to overfit the data set. In a second stage, the algorithm removes the hinge functions associated with the smallest increase in terms of the Generalized Cross Validation (GCV) criterion.

$$GCV_K = \frac{\sum_{i=1}^{N}(e_i - \hat{e}_{iK})^2}{\left(1 - \frac{C(K)}{N}\right)^2} \tag{3.9}$$

Here, $C(K)$ represents a model complexity penalty which is dependent on the number of hinge functions in the model while the numerator measures the lack of fit of a model with K hinge functions, $\hat{e}_{iK}$. Both LMS and MARS are illustrated on the Desharnais data set in Fig. 3.2 (b).

3.3.2 ML based algorithms

CART

CART (Classification And Regression Trees) is an algorithm that takes the well known idea of decision trees for classification [264], and adopts it to continuous targets. The splitting criterion used in this study is the least squared deviation:

$$\min \sum_{i \in L} (e_i - \overline{\mathbf{e}}_L)^2 + \sum_{i \in R} (e_i - \overline{\mathbf{e}}_R)^2 \tag{3.10}$$

The data set is split in a left node (L) and a right node (R) in a way that the sum of the squared differences between the observed and the average value is minimal. A minimum of ten observations at each terminal node is set to halt further tree construction. In retrospect, the fully grown tree is pruned to avoid overfitting on the training set. Fig. 3.2 (c) and (d) respectively present the estimation function and the accompanying binary regression tree for the Desharnais data set.

The good comprehensibility of regression trees can be considered a strong point of this technique. To determine the effort needed for a new project, it is sufficient to select the appropriate branches based on the characteristics of the new project. It is possible to construct an equivalent rule set based on the obtained regression tree.

This technique has previously been applied within a software effort prediction context where it consistently was found to be one of the better performing techniques [45, 46, 150, 176, 227].

M5

Introduced by Quinlan [263], the model tree technique (M5) can be considered as an extension to CART. A model tree will fit a linear regression to the observations at each leaf instead of assigning a single value like CART.

	MLP	RBFN
Training algorithm	Levenberg Marquart	Generalized Regression Neural Networks [294]
Topology	*Hidden layer*: log sigmoid *Output layer*: linear	
Previous applications	[49, 98, 198]	[138, 147]
Example	*Regression function*: Fig. 3.2 (e) *Trained network*: Fig. 3.2 (f)	

Table 3.3: Details perceptron based models

The model tree algorithm used in this study is the M5′ algorithm which is a variant of the original M5 algorithm [327]. A binary decision tree is induced by recursively applying the following splitting criterion, similar to CART.

$$\min\left(\frac{\mathbf{e}_L}{\mathbf{e}_L+\mathbf{e}_R}\times stdev(\mathbf{e}_L)+\frac{\mathbf{e}_R}{\mathbf{e}_L+\mathbf{e}_R}\times stdev(\mathbf{e}_R)\right) \tag{3.11}$$

Instead of taking the absolute deviations into account as is the case with CART, the M5′ algorithm applies a splitting criterion based on standard deviation. After growing and pruning the decision tree, a linear regression is fitted to the observations at each leaf. This regression only considers attributes selected by the different attribute conditions on the nodes, thus resulting in a tree based piecewise linear model. Finally, a smoothing process is applied to compensate for possible discontinuities that may occur between adjacent linear models at the different leaves.

The use of a model tree algorithm should allow for a more concise representation and higher accuracy compared to CART [263].

3.3.3 Perceptron based models

This class of learners was already discussed in Section 2.2.2 of this dissertation. Both Multi Layered Perceptrons (MLPs) and Radial Basis Function networks (RBFN) are considered and Table 3.3 provides additional details on these learners in the context of this chapter.

3.3.4 Kernel methods: LS-SVM

Least Squares SVM (LS-SVM) for regression is a variant of SVM, cfr Section 2.2.5, in which the goal is to find a linear function $f(\mathbf{x}_i)$ in a higher dimensional feature space minimizing the squared error r_i^2 [299]. Hereto, the following SVM

formulation is considered.

$$\begin{aligned} \min \quad & \frac{||\mathbf{w}||}{2}^2 + \frac{C}{2}\sum_{i=1}^{N} r_i^2 \\ \text{s.t.} \quad & \mathbf{w}\boldsymbol{\phi}(\mathbf{x}_i) + b + r_i = \varepsilon_i, \ i = 1 \ldots N \end{aligned} \tag{3.12}$$

The optimum of this convex function can be found by considering its Lagrangian and subsequently reformulating it into its dual form. This will give rise to a dot product in the higher dimensional feature space and a kernel function $K(\mathbf{x}_i, \mathbf{x}) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x})$ will be considered to facilitate computation of this product. A Radial Basis Function (RBF) kernel was used in this study as it was previously found to be a good choice in case of LS-SVMs [319], see Eq. 2.11.

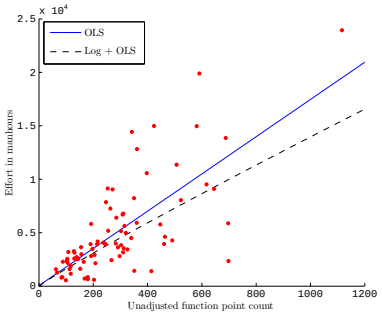
SVMs are a popular technique which has been applied in various domains. Since this is a rather recent machine learning technique, its suitability in the domain of software effort estimation has only been studied to a limited extent [192].

3.3.5 Other approaches: Case based reasoning

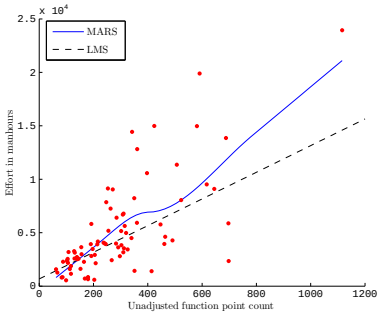
Case Based Reasoning (CBR) is a technique that works similar to the way in which an expert typically estimates software effort; it searches for the most similar cases and the effort is derived based on these retrieved cases. This technique is commonly used in software effort estimation, e.g. [49, 98, 204, 227, 283]. Typically, the Euclidian distance with rescaled attributes is used in retrieving the most similar case.

$$\text{Distance}(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{n} (x_{i(k)} - x_{j(k)})^2} \tag{3.13}$$

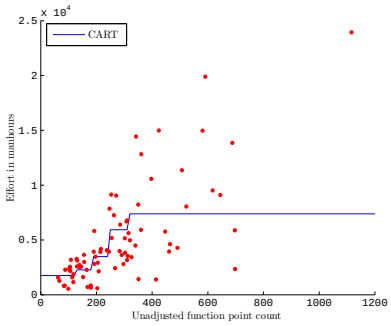
The rescaling is done by subtracting the minimum and dividing by the difference between maximum and minimum of an attribute. Only relevant attributes should be taken into account when calculating the Euclidian distance; in line with [45, 98], only attributes characterized by significant differences in effort are selected as found by applying a t-test in case of binary attributes and an ANOVA test otherwise. In both cases, only attributes with significant differences in effort at $\alpha = 95\%$ are retained during effort estimation. A final issue is the number of analogies to consider. In some studies, it is argued that no significant differences are found when retrieving more than one case, while other studies report a decrease in accuracy if more cases are retrieved [62]. Therefore, multiple alternatives are considered (k=1, k=2, k=3, and k=5). The final effort is determined by taking the average effort of the retrieved cases.



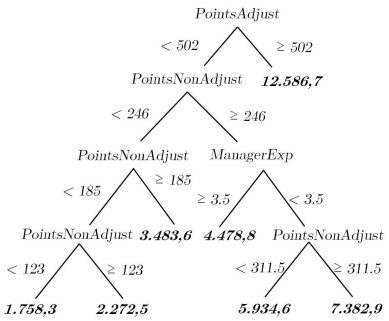
(a) OLS regression with and without log transformation



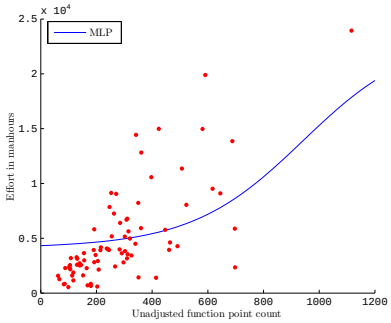
(b) MARS and LMS regression



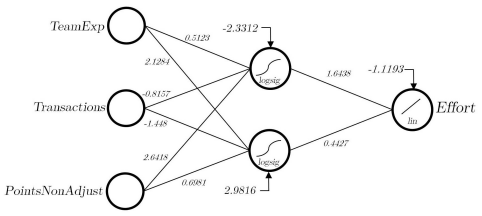
(c) CART estimation function



(d) CART tree



(e) MLP estimation function



(f) The MLP feed forward network

Figure 3.2: Comparison of machine learning techniques on the Desharnais data set. (a) presents OLS regression with and without log transformation, (b) represents MARS and LMS, (c) and (e) show respectively the CART and MLP regression line while (d) and (f) show the accompanying CART tree and neural network

3.4 Empirical setup

3.4.1 Data sets

Nine data sets from companies of different industrial sectors are used to assess the techniques discussed in Section 3.3. While other software effort estimation data sets exist in the public domain (e.g. a study of Mair et al. identified 31 such data sets [216]), the majority of these data sets are rather small. The overview of Mair et al. identified only three published data sets pertaining to over 50 projects; the Coc81, CSC and Desharnais data sets. The CSC data set however focusses on differences between estimation approaches instead of project characteristics and is therefore not included in this study. Investigating recent literature, three other data sets in the public domain were identified; i.e. the Cocnasa, Maxwell and USP05 data sets. Furthermore, researchers having access to data sets pertaining to over 150 projects were contacted as well as several companies involved in effort estimation. As such, access to four additional software effort estimation data sets was obtained (the Experience, ESA, ISBSG and Euroclear data sets).

The data sets typically contain a unique set of attributes that can be categorized as follows:

- *Size attributes* are attributes that contain information concerning the size of the software project. This information can be provided as Lines Of Code (LOC), Function Points, or some other measure. Size related variables are often considered to be important attributes to estimate effort [35].
- *Environment information* contains background information regarding the development team, the company, the project itself (e.g. the number of developers involved and their experience), and the sector of the developing company.
- *Project data* consist of attributes that relate to the specific purpose of the project and the project type. Also attributes concerning specific project requirements are placed in this category.
- *Development related variables* contain information about managerial aspects and/or technical aspects of the developed software projects, such as the programming language or type of database system that was used during development.

Table 3.4 provides an overview of the data sets, including number of attributes, observations, and previous use. The skewness and kurtosis as well as the minimum, mean and maximum of effort and size in Klocs or FP is given. On the right hand side, the partitioning of the different attributes across the four attribute types is shown for each data set. From this overview, the inherent difficulties to construct software effort estimation models become apparent. Data sets typically are strongly positively skewed indicating many 'small' projects and a limited number of 'large' outliers. Also, data sets within this domain are

Data set	Single/multi company	Application domains	Size measure	Range years
ESA	M	Space/military	Kloc	1983-1996
Experience	M	Manufacturing, banking, retail,...	FP[2]	1987-2010
ISBSG	M	Accounting, banking, trade,...	FP	1989-2009
USP05	S	Student projects	FP	2005
Coc81	S	Engineering, science, finance,...	Kloc	1970-1981
Cocnasa	M	Space/military	Kloc	1981-1999
Euroclear	S	Finance	-	2006-2008
Desharnais	S	Unknown	FP	1981-1988
Maxwell	S	Finance	FP	1985-1993

Table 3.5: Characteristics of software effort estimation data sets

typically small as compared to other domains. Most data mining techniques benefit from having more observations to learn from. Table 3.5 further details a number of basic characteristics of each data set including whether the data was collected from a single or multiple companies, the application domain of the software, the size measure used, and the years during which the information was collected.

Development effort

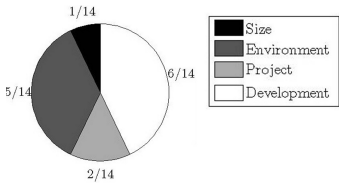
Ideally, development effort of projects is recorded using e.g. daily time sheets such that this value only reflects time spent on the actual development, and not relate to e.g. educational activities. As the data used in this study was collected in various ways, the correctness of the target attribute is hard to assess. E.g. the Maxwell and Desharnais data sets relate to projects done by subcontractors which used an unknown recording approach, while the Cocomo data sets (Cocnasa and Coc81) both include development and management hours. The ISBSG makes use of standardized questionaires which differ according to the functional sizing method employed at the beginning of the project. The questionnaires allow to specify different effort counting methods, with the 'staff hours (recorded)' option the most prevalent. This corresponds to the usage of daily records of project related tasks of all employees working on the project. As the measuring approach is unclear in most cases, this was not taken into consideration in the remainder of this study, in line with other researchers [45, 98, 177, 283].

[2]A five point scale is used to measure complexity weights instead of a three point scale.

Table 3.4: Overview software effort prediction data sets

ESA

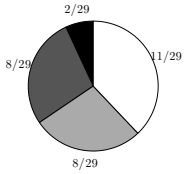
Skewness: 4.89
Kurtosis: 30.13
Minimum effort: 3 *Minimum Kloc: 1.5*
Mean effort: 264 *Mean Kloc: 56*
Max effort: 4,361 *Max Kloc: 413*



Number of observations: 131
Number of attributes: 14
Previously used in other studies, e.g. [16, 46, 224]
Reference: The ESA initiative for Software Productivity Benchmarking and Effort Estimation
www.esa.int/esapub/bulletin/bullet87/greves87.htm

Experience

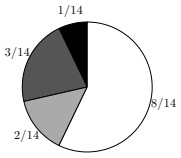
Skewness: 4.45
Kurtosis: 32.30
Minimum effort: 55 *Minimum FP: 7.14*
Mean effort: 4,248 *Mean FP: 728*
Max effort: 67,576 *Max FP: 14,092*



Number of observations: 627
Number of attributes: 29
Previously used in other studies, e.g. [16, 45, 198, 225, 297]
Reference: www.fisma.fi

ISBSG

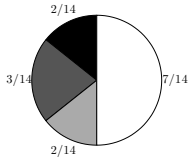
Skewness: 4.40
Kurtosis: 30.50
Minimum effort: 16 *Minimum FP: 4*
Mean effort: 4,226 *Mean FP: 374*
Max effort: 60,826 *Max FP: 7,400*



Number of observations: 1,160
Number of attributes: 14
Previously used in other studies, e.g. [149, 204, 275]
Reference: www.isbsg.org

USP05

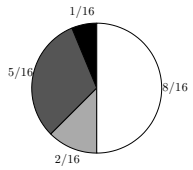
Skewness: 2.10
Kurtosis: 7.14
Minimum effort: 0.5 *Minimum FP: 0*
Mean effort: 7.25 *Mean FP: 25*
Max effort: 50 *Max FP: 321*



Number of observations: 193
Number of attributes: 14
Previously used in other studies, e.g. [203, 204]
Reference: [203, 204]
www.promisedata.org/?p=48

Coc81

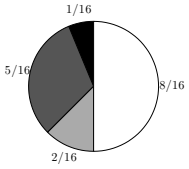
Skewness: 4.37
Kurtosis: 23.08
Minimum effort: 5.9 *Minimum Kloc: 1.98*
Mean effort: 683 *Mean Kloc: 77*
Max effort: 11,400 *Max Kloc: 1,150*



Number of observations: 63
Number of attributes: 16
Previously used in other studies, e.g. [57, 146, 147, 229, 275]
Reference: [35]
www.promisedata.org/?p=6

Cocnasa

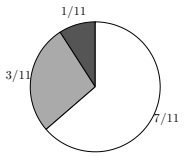
Skewness: 4.19
Kurtosis: 24.81
Minimum effort: 8.4 *Minimum Kloc: 0.9*
Mean effort: 624 *Mean Kloc: 94*
Max effort: 8,211 *Max Kloc: 980*



Number of observations: 93
Number of attributes: 16
Previously used in other studies, e.g. [57, 229]
Reference: www.promisedata.org/?p=35

Euroclear

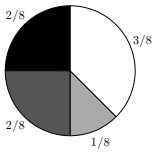
Skewness: 2.25
Kurtosis: 8.49
Minimum effort: 24
Mean effort: 1,402
Max effort: 7,758



Number of observations: 90
Number of attributes: 11
Data set has not been used in previous studies
Reference: www.euroclear.com

Desharnais

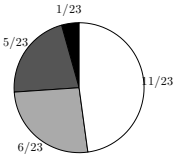
Skewness: 1.97
Kurtosis: 7.36
Minimum effort: 546 *Minimum FP: 62*
Mean effort: 5,046 *Mean FP: 287*
Max effort: 23,940 *Max FP: 1,116*



Number of observations: 81
Number of attributes: 9
Previously used in other studies, e.g. [16, 49, 205, 283, 284, 312]
Reference: [78]
www.promisedata.org/?p=9

Maxwell

Skewness: 3.27
Kurtosis: 15.52
Minimum effort: 583 *Minimum FP: 48*
Mean effort: 8,223 *Mean FP: 673*
Max effort: 63,694 *Max FP: 3,643*



Number of observations: 62
Number of attributes: 23
Previously used in other studies, e.g. [205, 275]
Reference: [223]
www.promisedata.org/?p=108

3.4.2 Data preprocessing

A first important step in each data mining exercise is data preprocessing. In order to correctly assess the techniques discussed in Section 3.3, the same data preprocessing steps are applied to each of the nine data sets.

First, starting from the raw data set, the data used to learn and validate the models is selected; only attributes that are known at the moment when the effort is estimated are taken into account (e.g. duration or cost are not known and therefore not included in the data set). An implicit assumption made in most software effort estimation studies is that size related attributes are taken for granted. However, in reality such attributes are often the result of an estimation process on their own. This remark is also echoed e.g. by Jørgensen, stating that 'a program module's size and degree of complexity ... are typically based on expert judgement' [161]. However, this assumption is made (but rarely mentioned) not only in this study, but in almost all studies in the domain of software effort estimation. Furthermore, some of the data sets include an indication about the reliability of observations. Taking this information into account, the observations with a higher possibility of being incorrect are discarded. In Table 3.4, an overview of the number of retained attributes and observations is provided.

Second, since some of the techniques are unable to cope with missing data (e.g. OLS regression), an attribute is removed if more than 25% of the attribute values are missing. Otherwise, for continuous attributes, median imputation is applied in line with [318]. In case of categorical attributes, a missing value flag is created if more than 15% of the values are missing; else, the observations associated with the missing value are removed from the data set. Since missing values often occur in the same observations, the number of discarded projects turned out to be low. In appendix A, the data preprocessing is illustrated for the ISBSG data set as this is the largest data set, both in number of attributes and number of projects.

Finally, coarse classification with k-means clustering is applied in case of categorical attributes with more than eight different categories (excluding the missing value flag). Afterwards, the categorical variables are transformed into binary variables using dummy encoding. No other preprocessing steps are performed on the data.

Data mining techniques typically perform better if a larger training set is available. On the other hand, a part of the data needs to be put aside as an independent test set in order to provide a realistic assessment of the performance. As can be seen from Table 3.4, the smallest data set contains 62 observations, while the largest contains up to 1,160 observations. In case of data sets containing more than 100 observations, repeated random hold-out splitting is applied. In such situation, Kirsopp et al. [175] claimed that 'ideally more than 20 sets should be deployed', a finding we agree upon, see also Fig. 3.3. When faced with smaller data sets, leave-one-out cross validation (LOOCV) is used, see also Section 2.3.1.

The LOOCV approach is computationally more expensive since as many

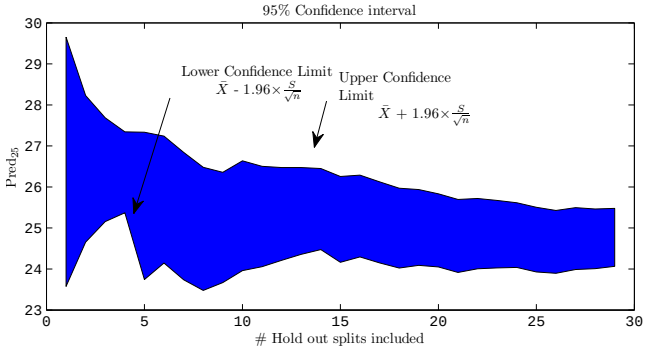


Figure 3.3: Example of 95% confidence interval

models need to be estimated as there are observations in the data set, but guarantees that as much as possible observations are used to learn from [186]. This approach has previously been adopted in the field of software effort prediction, see e.g. [16, 245, 283]. Note that there still is a discussion on whether k-fold cross validation or LOOCV is best, however Myrtveit et al. pointed out that LOOCV is more in line with real world situations [245].

3.4.3 Technique setup

Several of the estimation techniques discussed in Section 3.3 have adjustable parameters, also referred to as hyperparameters, which enable a model to be adapted to a specific problem. When appropriate, default values are used based on previous empirical studies and evaluations reported in the literature. If no generally accepted default parameter values exist, then these parameters are tuned using a grid-search procedure. In other words, a set of candidate parameter values is defined and all possible combinations are evaluated by means of a split-sample setup. The models are induced on 2/3 of the training data and the remainder is used as a validation set. The performance of the models for a range of parameter values is assessed using this validation set. The parameter values resulting in the best performance are selected and a final model is trained on the full training set. The MdMRE performance measure guided this tuning procedure, see Section 3.4.5.

3.4.4 Input selection

A second factor impacting the performance of software effort prediction models is input selection. Typically, similar, or occasionally better performance can be obtained by inducing models from a data set containing less, but highly relevant attributes, yielding a more concise and comprehensible model [222,323]. Therefore, a generic input selection procedure is applied in which a subset of highly predictive attributes is selected, discarding irrelevant variables.

Hereto, a generic backward input selection approach is proposed, in which in each step as many models are induced as there are variables left. Each of these models include all the remaining variables except one. The performance of the estimated models is compared, and the best performing attribute subset is selected. This procedure is repeated until only one attribute remains in the data set. The performance of the sequentially best models with a decreasing number of attributes is plotted, see Fig. 3.5. In the beginning, the performance typically remains stable or even increases while discarding attributes [57]. When the size of the attribute set drops below a certain number of attributes, the performance of the model drops sharply. The model at the elbow point is considered to incorporate the optimal trade off between maximizing the performance and minimizing the number of attributes. The performance at this elbow point is reported in Section 3.5.

Algorithm 1 provides a formal description of the followed procedure in case of data sets containing more than 100 observations. Otherwise, a cross validation based alternative is adopted.

Algorithm 1: Pseudo code of backward input selection

Let $D^n_{tr,l}$ and $D^n_{te,l}$ be the l^{th} $(l = 1 \dots 20)$ random holdout split of a data set with n attributes and N observations
for $j = n$ *to* 1 **do**
  for $k = 1$ *to* j **do**
    Exclude attribute k from data sets $D^j_{tr,l}$ and $D^j_{te,l}$
    for $l = 1$ *to* 20 **do**
      Induce model from $D^j_{tr,l}$
      Calculate model performance $P^j_{k,l}$ on $D^j_{te,l}$
    Calculate mean performance over all holdout splits: $P^j_k = \frac{1}{20} \sum_{l=1}^{20} P^j_{k,l}$
  Remove attribute $\mathbf{x}'_{(m)}$ from D^j where $P^j_m = \max_k(P^j_k)$ resulting in D^{j-1}
Plot(j,P^j_m) with $j = 1, \dots, n$
Select elbow point with optimal trade-off between performance and number of variables

3.4.5 Evaluation criteria

A key question to any estimation method is whether the predictions are accurate; the difference between the actual effort, e_i, and the predicted effort, $\hat{e}_i$, should be as small as possible. While several metrics have been proposed, including those detailed in Section 2.3.1 and Table 2.2, the literature on software effort estimation models is largely supported by Magnitude of Relative Error (MRE)

based metrics [64]. The MRE is calculated per instance as:

$$MRE_i = \frac{|e_i - \hat{e}_i|}{e_i} \tag{3.14}$$

Based on the MRE criterion, a number of accuracy measures are defined. The MRE value of individual predictions can be averaged, resulting in the Mean MRE (MMRE):

$$MMRE = \frac{100}{N} \sum_{i=1}^{N} \frac{|e_i - \hat{e}_i|}{e_i} \tag{3.15}$$

Although being a commonly used measure (see also Table 3.2), the MMRE can be highly affected by outliers [258]. To address this shortcoming, the MdMRE metric has been proposed which is the median of all MREs. This metric can be considered more robust to outliers, and is therefore preferred over the MMRE.

$$MdMRE = 100 \times median(MRE) \tag{3.16}$$

A complementary accuracy measure is Pred_L, the fraction of observations for which the predicted effort, $\hat{e}_i$, falls within L% of the actual effort, e_i:

$$Pred_L = \frac{100}{N} \sum_{i=1}^{N} \begin{cases} 1 & \text{if } MRE_i \leq \frac{L}{100} \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

Typically the Pred_{25} measure is considered, looking at the percentage of predictions that are within 25% of the actual values.

The Pred_{25} can take a value between 0 and 100% while the MdMRE can take any positive value. It is often difficult to compare results across different studies due to differences in empirical setup and data preprocessing, but a typical Pred_{25} lies in the range of 10% to 60%, while the MdMRE typically attains values between 30% and 100%.

Besides Pred_{25} and MdMRE, we also compared the techniques using a correlation metric. As the data is not normally distributed (see also Table 3.4), a rank correlation measure is adopted, which is a measure of the monotonic relationship between e_i and $\hat{e}_i$. More specifically, the Spearman's rank correlation coefficient, r_s, is used since this non-parametric correlation coefficient does not assume a normal distribution of the underlying data [133]. The Spearman's rank correlation takes a value between -1 and +1 with +1 (-1) indicating a perfect positive (negative) monotonic relationship between the actual values and the predicted values, and was already introduced in Table 2.2.

3.4.6 Statistical tests

The testing procedure outlined in Section 2.3.3 is followed to statistically verify the results of this experiment. This procedure consists of a Friedman test, see Eq. 2.16, followed by an appropriate post-hoc test. As the goal is to identify the best treatment (learner and the appliance of input selection), the post-hoc

Bonferroni-Dunn test [84] is selected. Let k be the number of treatments and P be the number of test attempts. The difference between the best treatment and other treatments is significant if the corresponding average ranks differ by at least the Critical Distance (CD), defined as:

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6P}} \tag{3.18}$$

with the critical value q_α based on the Studentized range statistic divided by $\sqrt{2}$, and an additional Bonferroni correction by dividing the confidence level α by the number of comparisons, $(k-1)$, to control for family wise testing. This results in a lower confidence level and thus in higher power.

The previous tests are performed in order to compare the results across the different data sets. Additionally, to compare the performance of two models on a single data set, the non-parametric Wilcoxon Matched Pairs test [331] (in case of the MdMRE) and the parametric t-test [133] (in case of Pred_{25} and correlation) are performed. The Wilcoxon Matched Pairs test compares the ranks for the positive and negative differences in performance of two models, and is defined as:

$$\min\left(\sum_{d_i>0} R(d_i) + \frac{1}{2}\sum_{d_i=0} R(d_i), \sum_{d_i<0} R(d_i) + \frac{1}{2}\sum_{d_i=0} R(d_i)\right) \tag{3.19}$$

with $R(d_i)$ the rank of the difference in performance between two models, ignoring signs. This test statistic follows approximately a standard normal distribution. The t-test is a general statistical test which is typically used to assess the difference between two responses. Under the null hypothesis, this test statistic follows a Student t-distribution.

3.5 Results

This section reports on the results of the techniques discussed in Section 3.3. The results both with and without application of the backward input selection procedure, as explained in Section 3.4.4, are provided in Tables 3.6, 3.7, and 3.8, respectively for the MdMRE, Pred_{25}, and Spearman's rank correlation. The top panels show the results without backward input selection and the bottom panels with backward input selection. The last column of each table displays the Average Ranks (AR) for the different techniques.

The best performing technique is reported in bold and underlined. Results that are not significantly different from the best performing technique at 95% are tabulated in boldface font, while results significantly different at 99% are displayed in italic script. Results significant at the 95% level but not at the 99% level are displayed in normal script.

3.5.1 Techniques

The results of the different modeling techniques are compared by first applying a Friedman test, followed by a Bonferroni-Dunn test. The Friedman test resulted in a p-value close to zero (p-values between 0.0000 and 0.0002) indicating the existence of significant differences across the applied techniques in all three cases (MdMRE, Pred_{25}, and r_s). In a next step, the Bonferroni-Dunn test to compare the performance of all the models with the best performing model is applied. The results are plotted in Fig. 3.4. The horizontal axis in these figures corresponds to the average rank of a technique across the different data sets. The techniques are represented by a horizontal line; the more this line is situated to the left, the better performing a technique is. The left end of this line depicts the average ranking while the length of the line corresponds to the critical distance for a difference between any technique and the best performing technique to be significant at the 99% confidence level. In case of 16 techniques and 9 data sets, this critical distance is 7.0829. The dotted, dashed and full vertical lines in the figures indicate the critical difference at respectively the 90%, 95% and 99% confidence level. A technique is significantly outperformed by the best technique if it is located at the right side of the vertical line.

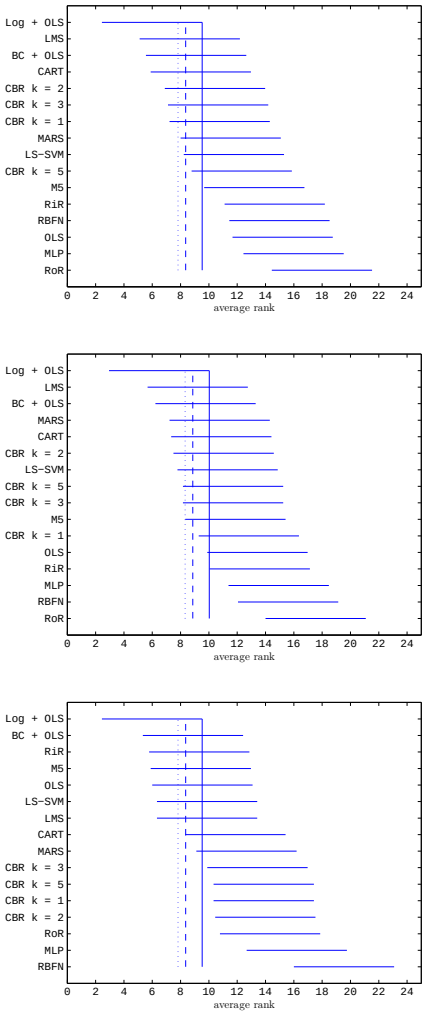


Figure 3.4: Ranking *without* backward input selection for MdMRE, Pred_{25}, and Spearman's rank correlation in resp. top, middle and bottom panel

Data sets in the domain of software effort estimation have specific characteristics [282]. They often have a limited number of observations, are affected by multicollinearity, and are known to be positively skewed and to contain outliers. Different techniques (both linear and non linear models, tree/rule induction techniques and case based reasoning) have been applied in this study

Table 3.6: Test set MdMRE performance

Techn./DS	ISBSG	Exp.	ESA	USP05	Euro.	Cocnasa	Coc81	Desh.	Maxwell	AR	
RiR	*70.4*	*57*	*59.9*	*73.7*	**69**	*87.7*	*336*	**36.9**	**41.9**	*11.1*	*Without* backward input selection
M5	*67*	*47.4*	*56.1*	*73.5*	**64.4**	*61.1*	*107*	**32.4**	*57.1*	9.67	
LMS	*62.3*	*45*	*52.2*	*47.8*	**54.4**	**36.8**	*66.4*	**32.8**	**44.6**	**5.11**	
OLS	*71.6*	*54.5*	*60.8*	*91.4*	**64.9**	*127*	*483*	*31*	*61.6*	*11.7*	
BC + OLS	*51*	*49.2*	*49.3*	**32.8**	**54.1**	*170*	*80.2*	*33.2*	50.8	**5.56**	
Log + OLS	**38.7**	**37.6**	**46.2**	33.7	**60.8**	**30.4**	**27.6**	**27.5**	**36.9**	**2.44**	
CART	*55.7*	*42.8*	*54.7*	*42.8*	**56.5**	*47*	*76.4*	*33.4*	**48.4**	**5.89**	
MARS	*59.6*	*54.2*	*62.3*	*74.8*	**66.9**	*46.6*	*66.6*	**29.8**	**51.9**	**8**	
RoR	*318*	*108*	*139*	*479*	*175*	*700*	*69.7*	*52.7*	*68.8*	*14.4*	
LS-SVM	*60.5*	*62.2*	*50.5*	*43.2*	68.8	**35.4**	*105*	**35.6**	**45.7**	**8.22**	
MLP	*115*	*98.4*	*80.4*	*72.1*	**61**	*80.6*	*99*	36.1	*65.5*	*12.4*	
CBR k = 1	*58*	*62.8*	*56.6*	**28.7**	**47.4**	**43.2**	*75.4*	*47.2*	**53.4**	**7.22**	
CBR k = 2	*52.7*	*59.1*	*53*	**31.5**	**52.7**	**41.7**	*86.1*	*45.8*	61.8	**6.89**	
CBR k = 3	*53.6*	*60.3*	*52*	**32.5**	**54.6**	49.2	*85.6*	*45.7*	**55.4**	**7.11**	
CBR k = 5	*57.3*	*61.9*	*53.9*	33.3	**60.1**	*58.3*	*91.3*	*40*	**56**	**8.78**	
RBFN	*61.6*	*57.6*	*70.2*	*47.7*	**56.6**	*85.6*	*89.2*	*100*	*100*	*11.4*	
RiR	*56.6*	*45.6*	*55.4*	*45.9*	*58*	*44.3*	*95.3*	**33.6**	**32.3**	*10.8*	*With* backward input selection
M5	*46.2*	*41.7*	*54.4*	*53.5*	*58.1*	*48.6*	*82.8*	**29.4**	**46.6**	*10.8*	
LMS	*50.7*	*38.1*	*50.3*	*36.6*	**42.6**	28.1	**47.6**	**27.9**	**38.4**	**4.33**	
OLS	*58.5*	*48.8*	*58.3*	*51.2*	*64.4*	*51.3*	*177*	**29.4**	**48.2**	*13.9*	
BC + OLS	*47.2*	*47.2*	*48.8*	**28.5**	**44.8**	*44.7*	62.9	31.1	**39.8**	**6.56**	
Log + OLS	**34.7**	**36.2**	**45.8**	**28.8**	*56*	**25.2**	**30.5**	*27.5*	**37.6**	**2.89**	
CART	*56.4*	*41*	*53.3*	*38.9*	51.2	*45*	*65.3*	**30.4**	**45.5**	**9.33**	
MARS	*49.9*	*39.6*	*60.2*	*61.5*	*53.8*	*39.3*	*65*	**26.8**	**41**	**8.67**	
RoR	*100*	*40.7*	*51*	*39.7*	**44.6**	**34.9**	*64.9*	31	**44.9**	**7.67**	
LS-SVM	*39.7*	*41.8*	**52.1**	*38.8*	*63.2*	*42.6*	*68.1*	**33.9**	**41**	**8.78**	
MLP	*56.7*	*44.8*	*57.1*	*48.1*	*51.7*	*38.5*	*79*	**25.4**	**44.1**	9.89	
CBR k = 1	*45.6*	*51*	*55.7*	**25.8**	**47.4**	**28.2**	*65.4*	**43.4**	**43.2**	**8.22**	
CBR k = 2	*45.6*	*47.6*	*51.2*	**34.8**	**46.1**	28.3	*75.7*	**37.1**	**39.1**	**7.56**	
CBR k = 3	*46.4*	*45.2*	*50*	**33.1**	**47**	*37.5*	*82.5*	**36.7**	**38.1**	**7.33**	
CBR k = 5	*46.9*	*42.6*	*48.4*	*31.8*	**48**	*44*	*74.8*	**34.6**	**36.2**	**6.78**	
RBFN	*51.6*	*54.2*	*59.5*	*32.6*	54.6	*82.5*	*88.6*	**47.9**	**40.6**	*12.4*	

Table 3.7: Test set $Pred_{25}$ performance

Techn./DS	ISBSG	Exp.	ESA	USP05	Euro.	Cocnasa	Coc81	Desh.	Maxwell	AR	
RiR	*21.9*	*22.9*	22.6	*21.7*	*16.7*	*15.1*	*9.52*	**39.5**	**32.3**	10.1	
M5	*26.1*	*27.5*	**24.5**	*20.7*	*17.8*	*24.7*	*12.7*	**38.3**	*14.5*	**8.33**	
LMS	*21*	29.2	**25.7**	*31.1*	*23.3*	**40.9**	*23.8*	**40.7**	*17.7*	**5.67**	
OLS	*23.9*	*24.4*	22.7	*17.8*	*17.8*	*15.1*	*6.35*	**45.7**	*25.8*	9.89	
BC + OLS	*25.2*	*24.5*	**25.4**	**42.2**	*23.3*	*7.53*	*14.3*	**43.2**	*24.2*	**6.17**	
Log + OLS	**33.1**	**33.8**	**27**	38.9	*20*	**45.2**	**49.2**	**43.2**	*25.8*	**2.94**	
CART	*22.7*	**30.6**	**23.5**	*35.3*	*21.1*	*22.6*	*11.1*	*34.6*	**30.6**	**7.33**	
MARS	*24.5*	*26*	*22.2*	*18.5*	*20*	**35.5**	*19*	**38.3**	27.4	**7.22**	
RoR	*9.43*	*15.4*	*8.97*	*6.92*	*12.2*	*3.23*	*20.6*	*25.9*	*16.1*	*14*	
LS-SVM	*24.7*	*21.5*	**26.5**	*32.8*	*16.7*	**38.7**	*12.7*	35.8	*25.8*	**7.78**	
MLP	*13.9*	*14.7*	*16.8*	*20.3*	*21.1*	*15.1*	*15.9*	35.8	*19.4*	*11.4*	
CBR k = 1	*18.8*	*19.5*	23.4	**48.8**	*18.9*	33.3	*15.9*	*23.5*	*22.6*	**9.28**	
CBR k = 2	*22*	*20.9*	**23.6**	**47.8**	**28.9**	33.3	*19*	*27.2*	*17.7*	**7.5**	
CBR k = 3	*25.8*	*21.6*	**24.4**	**42.2**	*18.9*	*26.9*	*11.1*	*33.3*	*22.6*	**8.22**	
CBR k = 5	*25.5*	*20*	23.2	**43.5**	*18.9*	*19.4*	*11.1*	35.8	**30.6**	**8.17**	
RBFN	*16.3*	*24*	*18.2*	*35.8*	*20*	*15.1*	*9.52*	*2.47*	*14.5*	*12.1*	
RiR	*22.7*	*28.8*	*24.8*	*27.4*	*17.8*	*35.5*	*7.94*	*38.3*	**35.5**	10.4	
M5	*28.1*	31.6	25.3	*26.1*	*20*	*30.1*	*12.7*	*38.3*	*29*	**8.94**	
LMS	*24.9*	**34.5**	*23.9*	*38*	**32.2**	**44.1**	*27*	*42*	*30.6*	**5.94**	
OLS	*23.4*	*27.2*	*24*	*21.1*	*17.8*	*31.2*	*6.35*	*38.3*	*28.6*	*12.4*	
BC + OLS	*25.5*	*24.1*	*24.1*	**46.5**	*21.1*	*29*	*19*	*42*	*30.4*	**8.5**	
Log + OLS	**36.3**	**34.6**	**26.7**	**45.5**	*20*	**49.5**	**39.7**	*43.2*	*32.1*	**3.33**	
CART	*22.8*	**32.2**	*24.3*	*36.2*	*25.6*	*28*	*12.7*	*35.8*	*30.4*	**9.72**	
MARS	*28.3*	**34.9**	*21.2*	*19.5*	*22.2*	*34.4*	*17.5*	**45.7**	*33.9*	**7.11**	
RoR	*19.1*	*30*	*23.4*	*29.8*	*18.9*	*37.6*	*15.9*	44.4	*23.2*	10.5	
LS-SVM	32.8	**33.3**	*24.9*	*40.2*	*17.8*	*30.1*	*17.5*	44.4	*26.8*	**7.67**	
MLP	*23.6*	*30.5*	*23.4*	*30.2*	*23.3*	*37.6*	*17.5*	**49.4**	*23.2*	**8.44**	
CBR k = 1	*25.9*	*24*	*24.7*	**49.8**	*18.9*	**47.3**	*19*	*33.3*	*21.4*	**8.83**	
CBR k = 2	*27.2*	*27.7*	25.3	**45.1**	**28.9**	**45.2**	*14.3*	*30.9*	**39.3**	**6.06**	
CBR k = 3	*26.4*	*26.7*	*24.8*	40.6	*22.2*	*36.6*	*12.7*	*33.3*	**35.7**	**8.17**	
CBR k = 5	*22.7*	*30.6*	**25.7**	**44.8**	**31.1**	*31.2*	*9.52*	*37*	*32.1*	**7.89**	
RBFN	*24.3*	*25.7*	*21.2*	**43.8**	*23.3*	*15.1*	*9.52*	*28.4*	*28.6*	*12*	

Without backward input selection

With backward input selection

Table 3.8: Test set Spearman's rank correlation performance

Techn./DS	ISBSG	Exp.	ESA	USP05	Euro.	Cocnasa	Coc81	Desh.	Maxwell	AR	
RiR	**0.807**	**0.743**	**0.736**	**0.808**	**0.492**	*0.681*	*0.706*	**0.81**	**0.746**	**5.78**	
M5	**0.785**	**0.789**	**0.729**	**0.787**	**0.459**	*0.738*	**0.826**	**0.712**	**0.699**	**5.89**	
LMS	0.749	**0.797**	**0.739**	*0.688*	0.434	0.786	*0.765*	**0.78**	0.603	**6.33**	
OLS	**0.775**	**0.747**	**0.731**	**0.779**	0.447	*0.749*	*0.736*	**0.829**	**0.728**	**6**	
BC + OLS	0.738	**0.729**	**0.742**	**0.845**	**0.62**	0.764	*0.638*	**0.82**	**0.669**	**5.33**	
Log + OLS	**0.865**	**0.837**	**0.788**	**0.851**	*0.0375*	**0.889**	**0.963**	**0.863**	**0.793**	**2.44**	
CART	0.732	**0.799**	**0.72**	**0.769**	0.417	*0.699*	*0.674*	0.614	**0.677**	**8.33**	
MARS	*0.72*	**0.759**	**0.717**	**0.736**	*0.41*	**0.806**	*0.695*	*0.497*	0.592	9.11	
RoR	0.747	**0.79**	*0.614*	*0.369*	*0.243*	*0.561*	**0.827**	*0.132*	*0.346*	*10.8*	
LS-SVM	*0.679*	**0.703**	**0.765**	**0.822**	*0.0118*	**0.814**	0.8	**0.736**	**0.761**	**6.33**	
MLP	*0.484*	*0.472*	*0.508*	*0.661*	*0.187*	*0.588*	*0.494*	**0.752**	**0.651**	*12.7*	
CBR k = 1	*0.618*	*0.462*	*0.611*	**0.803**	**0.569**	*0.741*	*0.75*	*0.443*	*0.496*	*10.3*	
CBR k = 2	*0.651*	*0.516*	*0.649*	**0.823**	**0.487**	*0.706*	*0.567*	*0.527*	*0.49*	*10.4*	
CBR k = 3	*0.66*	*0.528*	0.662	**0.831**	0.449	*0.714*	*0.51*	*0.546*	*0.556*	9.89	
CBR k = 5	*0.666*	*0.537*	0.659	**0.807**	0.453	*0.686*	*0.495*	0.597	0.586	*10.3*	
RBFN	*0.26*	*0.0861*	*0.238*	*0.338*	*-0.055*	*0.327*	*0.233*	*-0.315*	*-0.147*	*16*	
RiR	*0.759*	**0.809**	*0.724*	**0.806**	**0.53**	**0.809**	*0.324*	*0.694*	**0.782**	**7.22**	
M5	0.787	**0.821**	*0.731*	**0.826**	**0.532**	**0.847**	**0.771**	*0.716*	**0.749**	**4.17**	
LMS	**0.807**	**0.809**	*0.687*	*0.728*	**0.592**	**0.817**	**0.887**	**0.807**	**0.782**	**5.56**	
OLS	*0.785*	**0.771**	*0.728*	*0.705*	**0.475**	*0.724*	*0.18*	*0.716*	*0.719*	9.72	
BC + OLS	*0.783*	**0.75**	*0.736*	**0.861**	**0.485**	0.805	*0.664*	**0.802**	**0.763**	**6.56**	
Log + OLS	**0.849**	**0.833**	**0.782**	**0.855**	*-0.084*	**0.91**	**0.947**	**0.864**	**0.809**	**2.78**	
CART	*0.745*	**0.805**	*0.71*	0.794	0.385	*0.76*	*0.702*	*0.588*	*0.624*	9.94	
MARS	**0.802**	**0.832**	*0.697*	0.794	*0.335*	0.797	*0.672*	*0.692*	*0.722*	**8.06**	
RoR	*0.773*	**0.81**	*0.718*	*0.606*	**0.523**	**0.825**	**0.827**	**0.821**	*0.649*	**7**	
LS-SVM	**0.815**	**0.82**	*0.74*	**0.824**	**0.433**	*0.704*	**0.791**	*0.37*	*0.598*	**7.89**	
MLP	*0.664*	**0.758**	*0.696*	*0.713*	*0.085*	*0.737*	*0.476*	**0.765**	*0.595*	*12.2*	
CBR k = 1	*0.735*	*0.663*	*0.645*	*0.763*	**0.569**	**0.833**	*0.66*	*0.526*	*0.616*	*10.7*	
CBR k = 2	*0.713*	0.691	*0.669*	*0.773*	**0.428**	0.8	*0.695*	*0.548*	*0.669*	*11.6*	
CBR k = 3	*0.715*	**0.71**	*0.692*	*0.78*	**0.56**	**0.821**	*0.711*	*0.654*	*0.672*	**9.11**	
CBR k = 5	*0.74*	**0.782**	*0.724*	**0.806**	**0.516**	**0.831**	0.726	*0.683*	*0.615*	**8**	
RBFN	*0.584*	*0.177*	*0.627*	*0.627*	*-0.006*	*0.324*	*0.228*	*0.376*	*0.554*	*15.6*	

Without backward input selection

With backward input selection

that cope with these characteristics in different ways.

It can be seen from Tables 3.6, 3.7, and 3.8 that ordinary least squares regression with logarithmic transformation (Log + OLS) is the overall best performing technique. However, a number of other techniques including least median squares regression, ordinary least squares regression with Box Cox transformation, and CART, are not significantly outperformed by Log + OLS, see Fig. 3.4. There are a few notable exceptions such as the Euroclear data set (all three performance measures), the USP05 data set (in case of MdMRE and Pred_{25}), and both the Desharnais as well as the Maxwell data set (only for Pred_{25}). The good performance of Log + OLS can be attributed to the fact that such a transformation typically results in a distribution which better resembles a normal distribution. The range of possible values is also reduced thus limiting the number of outliers. Applying a Jarque-Bera test for normality on the log transformed data, it was found that in all but three cases (USP05, Euroclear and ISBSG), the null hypothesis of normality could not be rejected at $\alpha = 5\%$. Related to the normality of the distribution is the number of extreme values or outliers. For instance, if an outlier is defined as an observation at a distance of more than 1.5 times the inter quartile range of either the first or the third quartile, applying the logarithmic transformation removes all outliers in case of both the Cocnasa and the Coc81 data sets. In case of the ISBSG, Experience, and ESA data sets, less than 2% of the observations can be regarded as outliers after applying a log transformation. In case of the USP05 data set, such transformation was less able to reduce the number of outliers as still 23% of the observations are outliers. For the other data sets, the fraction of outliers was reduced to below 7% of all data.

The aspect of multicollinearity can be quantified using the Variance Inflation Factor (VIF), which is defined as:

$$VIF_j = \frac{1}{1 - R_j^2} \tag{3.20}$$

with R_j^2 the coefficient of determination obtained by regressing $\mathbf{x}_{(j)}$ on all other independent attributes. A value higher than 5 is typically considered to be an indication of multicollinearity. Most data sets are characterized by limited multicollinearity; only Desharnais, USP05, ISBSG and Euroclear data sets had a VIF higher than 5 for over 50% of their attributes. Still, it can be remarked that in these cases ridge regression, which is specifically designed to cope with multicollinearity, did not score particulary well. A possible explanation is that the regression models did not consider all attributes concurrently but instead used a stepwise selection procedure.

Finally, taking the number of observations into account, one would expect nonlinear techniques such as SVM, RBFN and MLP to perform better in case of larger data sets. However, even in case of the largest data set (ISBSG), state of the art non-linear techniques did not perform particulary well. While LS-SVM did not perform statistically worse than Log + OLS, both MLP and RBFN were found to be outperformed. Both tree/rule induction techniques (CART and M5)

were not statistically outperformed by Log + OLS. A possible explanation is that MLP, LS-SVM, and RBFN take all attributes jointly into account while other techniques consider only one attribute at the time. Hence, these other techniques are less affected by the sparseness of the data sets. Observe that a total of 1,160 observations is not large compared to a number of other domains [23].

It should be noted that different performance metrics measure different properties of the distribution of $\hat{e}_i$ [179], and thus could give inconsistent results if they are used to evaluate alternative prediction models. Similar observations can be made in other studies that used multiple performance metrics, e.g. [62, 244].

The Pred_{25} metric favors techniques that are generally accurate (e.g. fall within 25% of the actual value) and occasionally widely inaccurate. The MdMRE is an analogous measure as it is also outlier resistent. Both can therefore be considered to be measures benefiting from properly calibrated models. This can be seen by the similar results for both the Pred_{25} and the MdMRE, see Fig. 3.4 top and middle panel. The results for the Spearman's rank correlation are slightly more deviant, since for instance RiR scores third and model trees (M5) fourth, see Fig. 3.4, bottom panel. The best and worst performing techniques are however similar. The Spearman's rank correlation is a measure of monotonic relationship between the actual and the predicted values and is therefore insensitive to the precise calibration of the models.

Focussing on the results in terms of MdMRE, cfr. Table 3.6, it can be seen that Log + OLS is the best performing technique as it is ranked first in seven out of nine cases. Hence, the best average rank is attributed to Log + OLS, followed by LMS, BC + OLS, CART, various implementations of CBR, and MARS, none of which is significantly outperformed at the 95% significance level. The excellent results of various implementations of regression allow to build accurate and comprehensible software effort estimation models which can be checked against prior domain knowledge. These findings are consistent with the studies of Briand et al. [45, 46], who found that OLS is a good performing technique on previous versions of both Experience and ESA data sets. From a business perspective, the aspects of understandability and trust are important, thus techniques resulting in comprehensible and justifiable models (i.e. are in line with generally accepted domain knowledge) are preferred [221]. The good result of CART, which is not outperformed by the best performing technique at the 95% significance level, is interesting since CART allows to induce easy to understand piecewise linear prediction functions, and permits to verify whether the learned model is in line with prior domain knowledge, see Fig. 3.2 (d). Note however that models occasionally would need to be recalibrated on newly collected data, as relationships between attributes can change over time [87, 177]. The fact that several learners perform similar is in line with previous benchmarking studies in related domains like software fault prediction [200].

Cocomo

A comparison can be made between formal models such as Cocomo and data

Data set	Technique	MdMRE	Pred_{25}	r_s
Cocnasa	Cocomo	**26.9**	**46.2**	**0.966**
	Log + OLS	30.4	45.2	0.889
Coc81	Cocomo	31.9	36.5	**0.991**
	Log + OLS	**27.6**	**49.2**	0.863

Table 3.9: Results Cocomo models

mining techniques. However, as indicated in Section 3.2, applying Cocomo requires a data set to be in a suitable format. Hence, only two data sets qualify for this comparison (the Coc81 and Cocnasa data sets). Similar to data mining techniques which are typically tuned to better fit the underlying problem area, the coefficient a and the exponent b of the Cocomo model can be adjusted [35, ch. 29]. This requires the use of a separate training and test sample. Again, the same leave one out approach is followed to allow for better comparison to the other results, tuning both parameters on the complete data set minus one observation. The calibrated model is then used to estimate the effort of the test observation. Table 3.9 shows the result of this experiment. Comparing the Cocomo results to other data mining techniques shows that Cocomo yields similar results as regression with logarithmic transformation, which was previously found to be the best technique in both cases. Bold font indicates the best performing technique; no statistical significant differences were found by comparing the results between Cocomo and Log + OLS. Analogous results are to be expected as a Cocomo model is similar in nature to a regression model with logarithmic transformation.

It is interesting to note that the performance of Cocomo on the Coc81 data set can partially be attributed to the way the Cocomo model was constructed; the original Cocomo model was built and calibrated (including the precise values for each of the effort multipliers) on 56 of the 63 observations, with the (independent) test set consisting of 7 observations [35, ch. 29].

It should be noted that not all attributes are equally important in estimating software effort. Therefore in the next section, the results of a backward input selection procedure are discussed.

3.5.2 Backward input selection

The lower panels of Tables 3.6, 3.7, and 3.8 show the results of the generic backward input selection procedure. A Friedman test to compare the results with backward input selection and without backward input selection is performed (in this case will k equal 2 and P equal 9), yielding a p-value close to zero (p-value < 0.02 in all three cases). This indicates that on aggregate, applying input selection yields significantly better performance. While this result might seem counterintuitive at first sight, since information on certain attributes is removed from the data set, it makes sense that learning from a smaller data set, con-

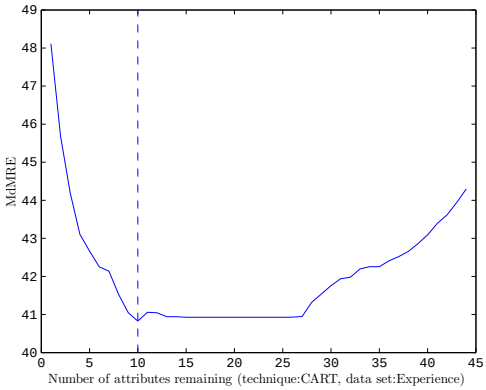
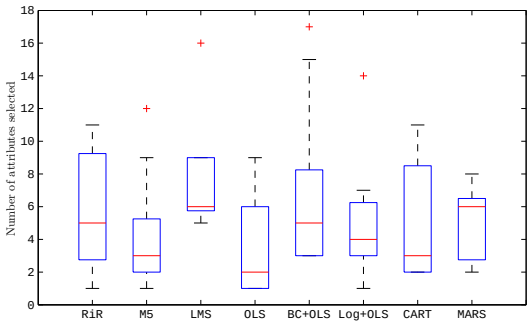


Figure 3.5: Performance evolution of the input selection procedure for CART applied on the ISBSG data set

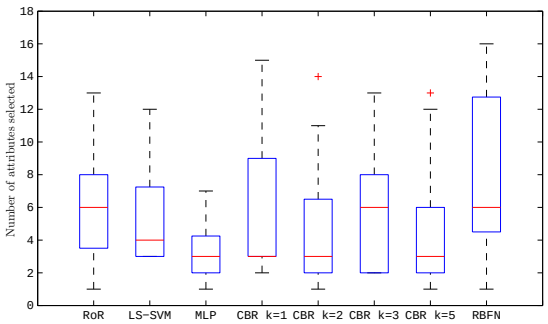
taining a limited set of highly predictive attributes, is easier than learning from a noisy data set containing many redundant and/or irrelevant attributes. The resulting models with input selection will also be more stable, since potential collinearity between attributes will be reduced, and are also preferable from an interpretation point of view. This also confirms the work of e.g. Chen et al. [57], who found that a higher accuracy could be achieved using a wrapper approach in case of Cocomo, and the findings of Azzeh et al. [18], who found similar results in case of CBR on the Desharnais and the ISBSG data set. These findings were also confirmed by Li et al. [205], using a mutual information filter approach on the Desharnais and the Maxwell data set. Fig. 3.5 plots a typical result of the backward input selection procedure which is exemplary for most techniques and data sets. On the horizontal axis, the number of variables remaining in the model are given, while on the vertical axis, the performance measured in MdMRE is provided. The number of remaining attributes is selected by identifying the elbow point in the performance evolution of the different techniques. Fig. 3.6 provides box plots of the number of selected attributes. In each box plot, the central line indicates the median number of selected attributes, while the edges of the box represent the 25^{th} and the 75^{th} percentiles for each technique. The whiskers extend to the most extreme numbers of selected attributes that are not considered to be outliers. Outliers finally are represented by crosses.

Technique evaluation

Drawing on the same statistical procedures, the results of the Friedman test indicate the existence of significant differences between techniques (p-values between 0.0000 and 0.0066) and subsequently, Bonferroni-Dunn tests are applied. The results of these tests are plotted in Fig. 3.7. From these plots, it can be concluded that in all three cases the best performing technique is again Log + OLS.



(a) Boxplots of RiR, M5, LMS, OLS, BC + OLS, Log + OLS, CART, and MARS



(b) Boxplots of RoR, LS-SVM, MLP, CBR k=1, CBR k=2, CBR k=3, CBR k=5, and RBFN

Figure 3.6: Boxplot of the number of attributes selected by the different techniques

Analogous to the case without input selection, the results from both MdMRE and $Pred_{25}$ are similar. The best performing technique is Log + OLS, followed closely by a number of other techniques including LMS, BC + OLS, MARS, LS-SVM, and various implementations of CBR. Again, more deviant results can be observed in case of the Spearman's rank correlation, although the best performing technique is similar to both other metrics. Fig. 3.9 provides an example of an OLS + Log model after application of the generic backward input selection procedure on both the Experience and the ESA data set. An advantage of regression models is the possibility to verify whether the model is in line with domain knowledge. For instance, it can be anticipated that larger projects require more effort, and thus a positive coefficient is more likely. Similarly, in case of more programming experience, a lower effort and thus a negative coefficient is expected.

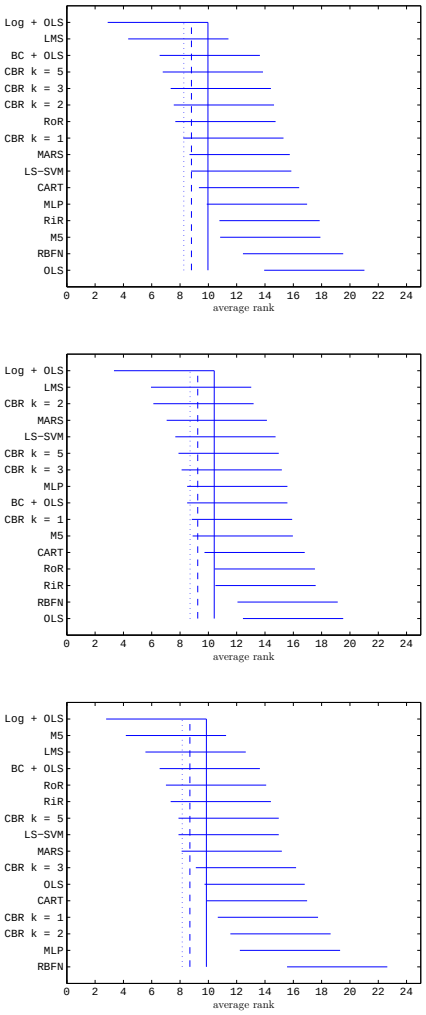


Figure 3.7: Ranking *with* backward input selection for MdMRE, $Pred_{25}$, and Spearman's rank correlation in resp. top, middle and bottom panel

While the Friedman test indicates that on average the results are better with input selection, analogous conclusions can be drawn to which techniques are more or less suited for software effort estimation. Log + OLS is again overall the best performing technique while a number of nonlinear techniques such as RBFN are less able to provide good estimations. Whether a specific technique on a specific data set benefits from selecting a subset of the data needs to be verified empirically.

Selected attributes

It can be seen from Fig. 3.6 that the number of selected attributes by the different techniques is rather low, typically ranging from two to ten. This means

that a surprisingly small number of attributes suffices to construct an effective software effort estimation model. Hence, the largest performance increase can be expected from improving the quality of data, instead of collecting more attributes of low predictive value. Data quality is an important issue in the context of any data mining task [235, 236].

In Section 3.4.1, four attribute types were identified that are present in the software effort estimation data sets: size, environment, project, and development. As a result of the input selection procedure, it is possible to identify the most important attribute types in the data sets, see Fig. 3.8. Size, development, and environment are considered to be attributes of high importance for software effort prediction. It should be noted that the attribute type 'size' typically only covers a limited number of attributes in a data set. Since this type of attribute is selected in nearly all cases, it is therefore considered to be highly predictive.

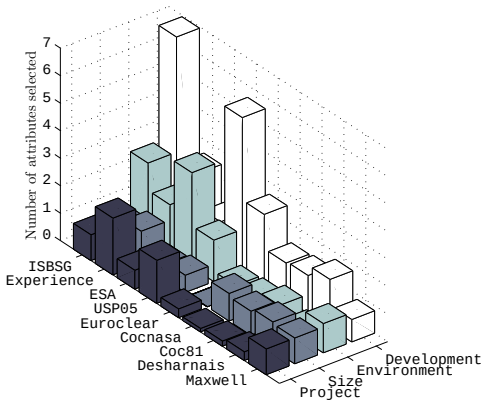


Figure 3.8: Bar chart of the average number of selected attributes per data set and per attribute type

Another observation that can be made, is that all four types of attributes are included in the set of most predictive attributes selected during the input selection procedure. Hence, none of the four attribute types can be omitted from the data sets without incurring some performance loss. Focussing on specific attribute categories, some attributes which are typically good predictors can be identified. For instance, programming language is a 'development' type attribute that is often selected by the input selection procedure. This is to be expected, since programming language was previously found to have an important impact on development effort, e.g. by Albrecht et al. [5]. Concerning environment attributes, variables related to team size and company sector prove to be good predictors to estimate effort as well.

We also considered a minimal Redundancy, Maximum Relevance (mRMR) filter approach [255] as an alternative to the backward input selection in this study, similar to the study of Li et al. [205]. Using this approach, we selected the top ten ranking attributes. This filter approach gave however similar results to the backward input selection approach, i.e. also indicating an increased performance by taking a highly predictive set of attributes and the importance of the size-related attribute. In order not to overload this chapter, the results of this filter are not further detailed.

Data set / Obtained model

Experience - Reduced model
$\text{Log}(\hat{e}_i) = 3.71 + 0.72\times\log(\text{size}) + 0.12\times(\text{IT staff req.})$
$- 0.09\times(\text{Usability req.}) - 0.73\times(\text{Company2})$
$- 0.42\times(\text{Company3})$
ESA - Reduced model
$\text{Log}(\hat{e}_i) = 1.51 + 0.41\times\log(\text{size}) + 0.92\times\log(\text{Team size})$
$+ 0.14\times(\text{Mem. constraint}) + 0.55\times(\text{Fortran/AS}) - 0.77\times(\text{UK})$
$- 0.23\times(\text{Lang. experience}) - 0.82\times(\text{ESA project category 3})$

Figure 3.9: Example of the best performing technique

3.6 Discussion

Technique	Over-estimate	Under-estimate	Aggregate	Comp. time
RiR	47.8	29.8	18.0	Medium
M5	44.1	32.9	11.2	Low
LMS	36.5	38.1	-1.6	Low
OLS	43.1	34.7	8.4	Low
BC + OLS	36.9	37.7	-0.8	Low
Log + OLS	33.5	31.4	2.1	Low
CART	47.5	27	20.5	Low
MARS	44.7	31.3	13.4	High
RoR	39.6	46	-6.4	Low
LS-SVM	45.8	28.6	17.1	High
MLP	51.5	33	18.5	High
CBR k = 1	37.3	38.3	-1	Low
CBR k = 2	41.4	33	8.4	Low
CBR k = 3	45.2	30	15.2	Low
CBR k = 5	46.2	28.6	17.6	Low
RBFN	33.6	49.1	-15.5	Low

Table 3.10: Overview of calibration and computation time

The results of this benchmarking study partially confirm the results of previous studies [45, 46, 244]. Simple, understandable techniques like OLS with log transformation of attributes and target, perform as good as (or better than) nonlinear techniques. Additionally, a formal model such as Cocomo performed at least equally good as OLS with log transformation on the Coc81 and Cocnasa data sets. These two data sets were collected with the Cocomo model in mind. However, this model requires a specific set of attributes and cannot be applied on data sets that do not comply with this requirement. Although the performance differences can be small in absolute terms, a minor difference in es-

timation performance can cause more frequent and larger project cost overruns during software development. Hence, even small differences can be important from a cost and operational perspective [162].

These results also indicate that data mining techniques can make a valuable contribution to the set of software effort estimation techniques, but should not replace expert judgement. Instead, both approaches should be seen as complements. Depending on the situation, either expert driven or analytical methods might be preferable as first line estimation. In case the experts possess a significant amount of contextual information not available to an analytical method, expert driven approaches might be preferred [161]. An automated data mining technique can then be adopted to check for potential subjective biases in the expert estimations. Additionally, various estimations can be combined in alternative ways to improve the overall accuracy, as investigated by e.g. MacDonell et al. [212] which concluded 'that there is indeed potential benefit in using more than one technique'. A simple approach could be to take the average across estimations, while a more advanced approach would investigate in which case a specific technique yields the most accurate estimation. When combining estimates of techniques, the potential bias of the technique, i.e. the tendency to over- or underestimate effort, should be taken into account. Since effort is a continuous attribute, typically some error is to be expected. However, if the estimate is far from the actual value, e.g. more than 25%, the estimate can not be considered 'accurate'. The first two columns in Table 3.10 provide the average over- and underestimation per technique across all data sets. The third column aggregates these two values, indicating whether a technique has an overall tendency to over- or underestimate. Combining different techniques increases computational requirements and hampers comprehensibility. The required computation time is dependent on a number of different aspects, including data set, hardware, technique, empirical setup, and parameter tuning. An indication of computational requirements is also presented in the last column of Table 3.10.

A third conclusion is that the selection of a proper estimation technique can have a significant impact on the performance. A simple technique like regression is found to be well suited for software effort estimation which is particularly interesting as it is a well documented technique with a number of interesting qualities like statistical significance testing of parameters and stepwise analysis. This conclusion is valid with respect to the different metrics that are used to evaluate the techniques. Furthermore, it is shown that typically a significant performance increase can be expected by constructing software effort estimation models with a limited set of highly predictive attributes. Hence, it is advised to focus on data quality rather than collecting as much predictive attributes as possible. Attributes related to the size of a software project, to the development, and environment characteristics, are considered to be the most important types of attributes.

Finally, the comprehensibility of the estimation model is often of paramount importance to instigate model acceptance in a business setting. While log-linear models are understandable to a certain level, rule sets or decision trees are considered more comprehensible to end users [73, 145]. This topic has been

investigated only to a limited extent in software effort estimation; see e.g. [146]. In a follow-up study, we investigated the extraction of regression rules from the ISBSG R11 data set; however, for reasons of brevity, we refer to Setiono et al. for further details [278].

All sorts of computer errors are now turning up. You'd be surprised to know the number of doctors who claim they are treating pregnant men.

Isaac Asimov, 1920 – 1992

4

Comprehensible software fault prediction with Bayesian Network classifiers

In the fourth chapter, we turn towards a second key topic in empirical software engineering: software fault prediction. This time, we move forward in the software development cycle, situating ourselves after the actual coding, and before software testing. Zooming in on the source code, regions which are more likely to contain faults are identified, and can subsequently be made target of heightened software testing efforts. Experience learns that thoroughly testing the code base can pose an insurmountable expense to developers, and software fault prediction offers solutions to alleviate this issue. This chapter considers the popular Naive Bayes classifier, of which predictive performance and comprehensibility are often cited as major strengths, and further contributes to the literature by considering alternative Bayesian Network (BN) classifiers which boost the possibility to construct simpler networks with less nodes and arcs. Furthermore, the applicability of the Markov blanket principle for feature selection, which is a natural extension to BN theory, is investigated. The results, both in terms of the AUC and the recently introduced H-measure, are rigorously tested using again the statistical framework of Demšar. It is concluded that simple and comprehensible networks with less nodes can be constructed using BN classifiers other than the Naive Bayes classifier. Furthermore, it is found that the aspects of comprehensibility and predictive performance need to be balanced out, and also the development context is an item which should be taken into account during model selection.

This chapter is based on the following paper

- K. Dejaeger, T. Verbraken and B. Baesens, "Towards comprehensible software fault prediction models using Bayesian network classifiers," *IEEE Transactions on Software Engineering*, Accepted for publication.

4.1 Introduction

The ubiquitous presence of computers has given rise to novel research fields such as software development, computer engineering and artificial intelligence

[75] while at the same time enabling novel developments in other domains like medicine, telecommunications and image processing [111, 325, 326]. In spite of all the efforts invested in the field of software engineering, the development of software remains jeopardized by high cancelation rates and considerable delays [162]. Chapter 1 already attested the importance of software testing processes and the impact software faults potentially have on corrective maintenance budgets. As such, the importance of software testing has long been recognized; e.g. the waterfall approach presented in Fig. 1.1 specifies the implementation of a separate testing phase [270][1]. The first work on the topic of software testing dates from 1975 [115] and since the pioneering work of Goodenough, numerous books and papers have been published on this topic. Software testing expenses can amount up to 60% of the overall development budget [130], and several approaches to support these efforts have been proposed.

A key finding to software testing is the fact that faults tend to cluster; i.e. to be contained in a limited number of software modules [286]. Gyimóthy et al. found while investigating the open source software web and email suite Mozilla that bugs were present in 42.04% of all software classes [119]. Even more skewed distributions have been reported by others; e.g. Ostrand et al. who investigated several successive releases of a large inventory system, stated that 'At each release after the first, faults occurred in 20% or fewer of the files' [250]. In fact, it has been shown that the distribution of faults over a system can be modeled by a Weibull probability distribution [340]. This motivates the use of software fault prediction models which provide an upfront indication whether code is likely to contain faults; i.e. is fault prone. A timely identification of this fault prone code will allow for a more efficient allocation of testing resources and an improved overall software quality. To construct such a prediction model which discriminates between fault prone code segments and those presumed to be fault free, the use of static code features characterizing code segments has been advocated [52, 54]. Static code features which can be automatically collected from software source code have proven to be useful [289], and are widely used in academic research as well as in industry settings [230, 317].

A myriad of different approaches to assist in the fault prediction task has previously been proposed, including expert driven methods, statistical models and machine learning techniques [54]. In spite of the use of various advanced techniques including association rule mining [25], support vector machines [88], neural networks [262], genetic programming [91], and swarm intelligence [320], it is recognized that their gain compared to simple techniques such as Naive Bayes is limited [230]. The use of Naive Bayes to model the presence of software faults is also advocated by other researchers citing predictive performance and comprehensibility as its major strengths [55, 95, 315]. Underlying to Naive Bayes is the assumption of conditional independency between attributes. Despite the restrictions on the network structure imposed by this conditional independence assumption, Naive Bayes classifiers have been found to perform surprisingly well

[1]Data collected from different application domains by the ISBSG (International Software Benchmarking Standards Group) indicates that a waterfall-like approach remains commonly used in modern software development, www.ISBSG.org.

in e.g. the medical domain [308]. A similar conclusion was also echoed by Holte, who compared the complexity and accuracy of different rule learners [142]. He noted that simple models are often not outperformed by more complex ones and that in such case, the simpler model should be selected. The good performance of Naive Bayes compared to other classifiers inspired several modifications relaxing the conditional independence assumption to allow the construction of more complex network structures. Well-known examples include Augmented Naive Bayes and General Bayesian Network classifiers. The latter even impose no restrictions on the network structure and various algorithms to learn a suitable network structure have been introduced. Two such algorithms, together with various Augmented Naive Bayes classifiers and a selection of benchmark classifiers constitute the set of techniques under consideration in this study. Additionally, the use of the Markov blanket feature selection procedure, which provides a natural way to reduce the set of available features, is also investigated. The results of the analyses are presented both in terms of AUC (Area Under the ROC Curve) and the novel H-measure and are subjected to rigorous statistical testing to verify their significance.

The remainder of this chapter is structured as follows.
Section 4.2 positions this chapter in the software fault prediction literature.
Section 4.3 discusses the working of Naive Bayes classifiers and provides a number of extensions hereon, also giving attention to the Markov blanket feature selection procedure.
Section 4.4 reflects upon empirical setup, and provides the rationale for using the novel H-measure.
Section 4.5 elaborates on the suitability of extending the Naive Bayes classifier in a software fault prediction context from a comprehensibility point of view.
Section 4.6 finally provides a set of general conclusions.

4.2 Related work

The observation that costs incurred to correct faults increase exponentially with the time they remain uncorrected in the system sparked interest into early warning systems, trying to locate fault prone code already during development, before the software goes gold. Section 1.4 offers an overview of possible approaches hereto, the protagonist being software fault prediction which explores the characteristics of individual code segments in order to identify those segments that are fault prone [230] or to predict the number of faults in each segment [251]. In the first, software fault prediction is regarded as a classification problem while the latter approach considers it to be a regression problem. Note that in this study, an emphasis is put on the classification point of view. To this purpose, a large number of software code characteristics (also referred to as 'static code features') have been introduced to the domain of software fault prediction. These include McCabe and Halstead metrics, metrics adopting object-oriented programming concepts such as the Chidamber-Kemerer (CK) metrics suite [61]

or the Conceptual Cohesion of Classes (C3) measure introduced in [218], as well as various file and component based metrics [215, 250]. Note that some of these metrics can be collected on different granularity levels; e.g. McCabe and Halstead measures have been explored on the level of software functions, classes and files. Fenton et al. [94] conjectured that the most widely used static code features include LOC (Lines Of Code) based measures, Halstead metrics and McCabe complexity metrics, which was also echoed by Catal et al. [54]. Evidence hereon can be found in the publicly available software fault prediction data; e.g. all projects in the NASA MDP repository contain these metrics at the level of software modules and also other data sets rely upon these values [174, 316, 317, 344].

There has been considerable debate about the extent to which software fault prediction models constructed from these metrics actually contribute to supporting software testing processes. It is e.g. demonstrated by Fenton and Pfleeger that by using different language constructs, source code providing the same functionality can have different static code values [96]. Furthermore, while several studies failed to validate the usefulness of e.g. McCabe cyclomatic complexity metrics for software fault prediction [280, 281], other studies showed opposite results [54].

More recently, the validity of software fault prediction using static code features has been empirically illustrated by e.g. Menzies et al. who stated that static code features are *useful*, *easy to use* and *widely used* [230]. This observation was later also confirmed by other work, see e.g. [317].

Useful:
Several studies have reported on the inability of real-life fault predictors to obtain similar detection rates as static code based classification techniques.

- A panel consisting of academic and business experts at the IEEE Metrics 2002 symposium concluded that manual software reviews typically account for around 60% of all identified faults, independent of the domain or level of maturity of the organization [289]. Similar (or even worse) defect detection capabilities were observed amongst other industrial defect detectors [230].

- Empirical evidence comparing an expert driven approach with the use of statistical techniques to locate software faults indicated the superior performance of the latter stating that 'When it comes to comparing both methods we found that statistical models outperformed expert estimations' [309]. Other advantages of adopting static code based classifiers that were identified include improved fault prediction efficiency and the ability to cope with large data sets, resulting in possibly finer grained fault prediction. The study also found human experts to be unable to grasp or understand the structure of large systems and as a result unable to e.g. provide a ranking of the fault proneness across *all* system components.

  On the flip side, human experts might be better able to incorporate qualitative information when predicting the fault proneness of a software component; however, this advantage proved not to outweigh the disadvantages.

By contrast, using static code based classification techniques, noticeably better detection rates have been recorded. For instance, Menzies et al. reported an average detection rate of 71% [230].

Easy to use:
Static code features such as McCabe and Halstead metrics can be mined from the source code using automated methods. Several tools have been proposed, including McCabe IQ™,[2], RUBY [55], EMERALD [144] and Prest [183] to assist practitioners in this effort. In addition to static code features characterizing each code segment, labels indicating whether faults were found are needed to construct software fault prediction models. This often requires a matching between data contained in a bug database such as Bugzilla©,[3] and the mined source code. Various text mining techniques exist to facilitate this matching effort [99].

Widely used:
Static code features have been extensively investigated by researchers [52, 54] and their use in industry has been long reckoned, e.g. [94]. It is argued that some large government software contractors will not review code segments unless they are flagged as fault prone [230]. Moreover, the ability to collect data concerning the software development process is also a requirement when trying to achieve Capability Maturity Model Integration® (CMMI) level 2 appraisal. Obtaining such appraisal can be an obligation to compete for (government) contracts [36].

Researchers have adopted a myriad of different techniques to construct software fault prediction models. These include various statistical techniques such as logistic regression and Naive Bayes which explicitly construct an underlying probability model. Furthermore, different machine learning techniques such as decision trees, models based on the notion of perceptrons, support vector machines, and techniques that do not explicitly construct a prediction model but instead look at a set of most similar known cases have also been investigated. A taxonomy of the most often employed classification techniques for software fault prediction is offered in Fig. 4.1. References to earlier work using each technique are provided between square brackets. While this overview does not attempt to be exhaustive, it is clear that Bayesian Network (BN) classifiers are in fact one of the most popular techniques to model the presence of software faults in a system. One of the earliest references to BN classifiers in this context can be found in the work of Fenton et al. who reckoned that these techniques offer several advantages including the ability to explicitly model uncertainty, the good comprehensibility and the avoidance of multicollinearity related problems [94]. They also highlighted the possibility of expert driven BN creation, which however deviates from how such classifiers are commonly utilized in machine learning literature. Persuaded by these remarks, several authors have used such models [200, 230, 315, 317]. Especially the Naive Bayes classifier has been carefully investigated and has been found to perform exceptionally well, despite being a very simple technique. For instance, Menzies et al. found Naive Bayes

[2] www.mccabe.com
[3] www.bugzilla.org

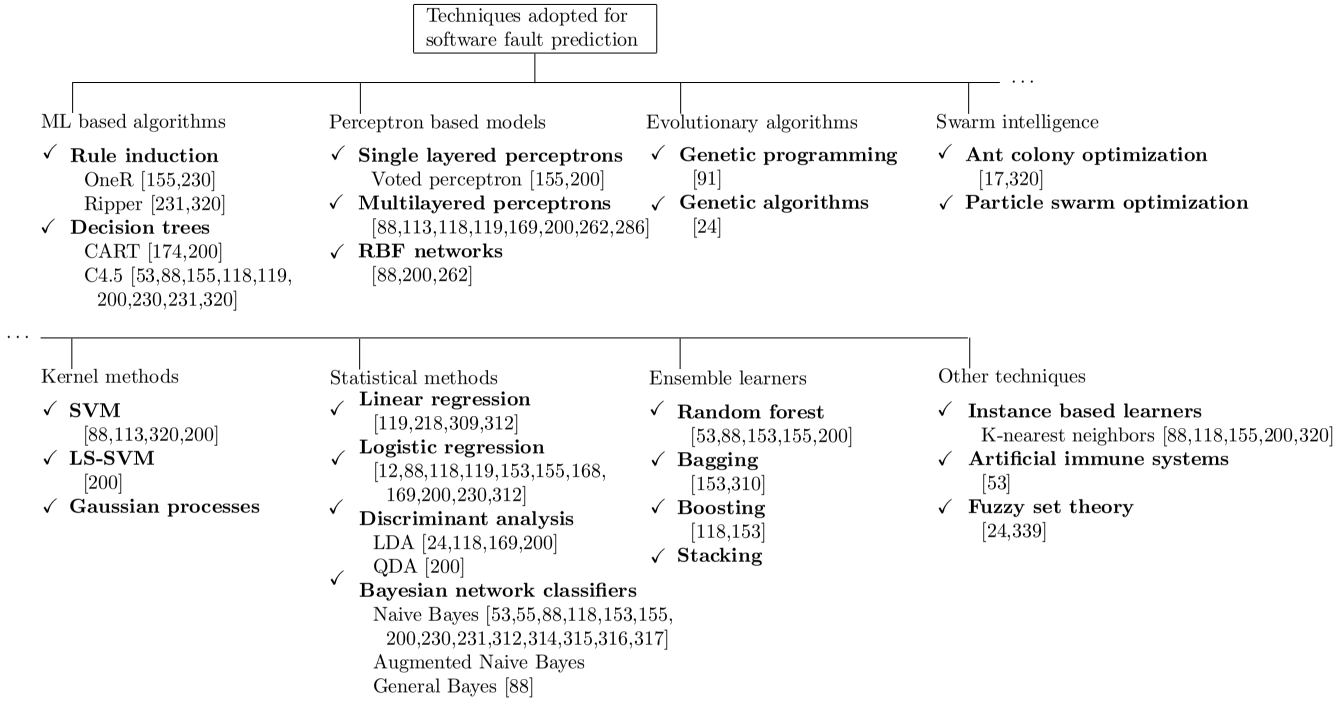


Figure 4.1: Supervised machine learning taxonomy in software fault prediction

with logarithmic transformation of the inputs to be the best performing prediction model as compared to two rule induction techniques, i.e. C4.5 and OneR [230]. This result was later partially confirmed by Lessmann et al. who found an ensemble learner, Random Forest, to be the best performing technique. BN learners however were not found to be statistically outperformed by this ensemble learner [200]. The assumption of conditional independence underlying Naive Bayes is typically not met in a software fault prediction context; different static code features try to measure the same underlying dimensions of the source code. The relaxation of this assumption has been investigated by Turhan et al., who used instead of a univariate gaussian approximation of the unknown distribution of the static code features, a multivariate gaussian approximation [315]. It was concluded that the independence assumption of Naive Bayes is not harmful for software defect data after the data was preprocessed using Principal Component Analysis (PCA) which maps the data on a set of orthogonal axes. Remark that by construction, principal components are not correlated to each other. In our work, the impact of the conditional independence assumption is considered from a different perspective by investigating BN classifiers that explicitly model the conditional independence amongst attributes. In another study, Turhan et al. looked into the use of various attribute weighting heuristics (e.g. heuristics based on concepts of Shannon's information theory and statistical methods such as the PCA scores and the Kullback-Leibler Divergence) to improve BN learners [314, 315]. As such, they adopted a two stage approach by first applying these heuristics to rank all attributes and afterwards providing this ranking to the BN learner. They showed that using weighting heuristics based on Shannon's information theory (information gain and gain ratio) or feature selection techniques yield improved results. Their findings motivated the inclusion of the Markov blanket feature selection into this study, which is a feature selection approach rooted in BN theory.

4.3 Bayesian network classifiers

Complementing the description in Section 2.2.6, this part first presents a general introduction to Bayesian networks, followed by a description of the Naive Bayes classifier. Next, a number of alternative BN classifiers relaxing the assumption of conditional independence are explained. Two alternative machine learning techniques, which serve as a benchmark in this study, are also detailed.

4.3.1 Bayesian networks

A Bayesian Network (BN) represents a joint probability distribution over a set of stochastic variables, either discrete or continuous. It can be visualized as a graph consisting of nodes representing the individual variables $x_{(j)}$ and directed arcs indicating the existence of dependencies between variables. Likewise, the absence of an arc between two nodes $x_{(j)}$ and $x_{(j')}$ indicates the conditional independence between both variables given their parents in the graph. Associated

with each node is a probability table, containing the probability distribution of each variable conditional on the direct parent(s) in the graph [254]. Underlying BN is the Bayes theorem, which formulates the posterior probability of the presence of faults in terms of prior probabilities and the reverse conditional probability:

$$P(y_i = 1|\mathbf{x}_i) = \frac{P(\mathbf{x}_i|y_i = 1)\ P(y_i = 1)}{P(\mathbf{x}_i)}. \tag{4.1}$$

Note that $P(\mathbf{x}_i)$ acts as a normalizing constant herein and can be ignored.

More formally, a BN comprises two parts $B = \langle G, \Theta \rangle$. G is a directed acyclic graph (DAG) conveying the direct dependence relationships within the data set whereas the second part, Θ, represents the conditional probability distribution of each variable. Adopting the notation of Cooper & Herskovits [65], $\Pi_{x_{(j)}}$ represents the set of direct parents of $x_{(j)}$ in G. Θ contains a parameter $\theta_{x_{(j)}|\Pi_{x_{(j)}}} = P_B(x_{(j)}|\Pi_{x_{(j)}})$ for each possible value of $x_{(j)}$, given each possible combination of values of all direct parents. The network B then represents the following joint probability distribution:

$$P_B(x_{(1)}, \ldots, x_{(n)}) = \prod_{j=1}^{n} P_B(x_{(j)}|\Pi_{x_{(j)}}) = \prod_{j=1}^{n} \theta_{x_{(j)}|\Pi_{x_{(j)}}}. \tag{4.2}$$

Typically, the task of learning a BN can be decomposed into two subtasks which are executed sequentially. First, the exact structure G of the network needs to be determined. In general, it is infeasible to iterate over all possible network structures and therefore, several constraints can be imposed, leading to different learning algorithms. After establishing the exact network structure G, the parameters associated with each node need to be estimated. In this study, the empirical frequencies as observed in the training data D_{trn} are used to estimate these parameters:

$$\theta_{x_{(j)}|\Pi_{x_{(j)}}} = \hat{P}_{D_{trn}}(x_{(j)}|\Pi_{x_{(j)}}). \tag{4.3}$$

It can be shown that these estimates maximize the log likelihood of the network B given the training data D_{trn}. Note that these estimates might be further improved by a smoothing operation, e.g. by using a Laplace correction or an M-estimate [106].

Generally, BN classifiers can be considered as probabilistic white-box classifiers. They allow to calculate the (joint) posterior probability distribution of any subset of unobserved stochastic variables, given that the variables in the complementary subset are observed. This functionality enables the use of BN as statistical classifiers which provide a final classification by selecting an appropriate threshold on the posterior probability distribution of the (unobserved) class node. Alternatively, assuming all misclassification costs are equal, a winner-takes-all rule can be adopted [83]. A pivotal ability of these classifiers is the use of graphical artifacts which facilitates the understanding of complex and seemingly contradictory relationships within the data [94].

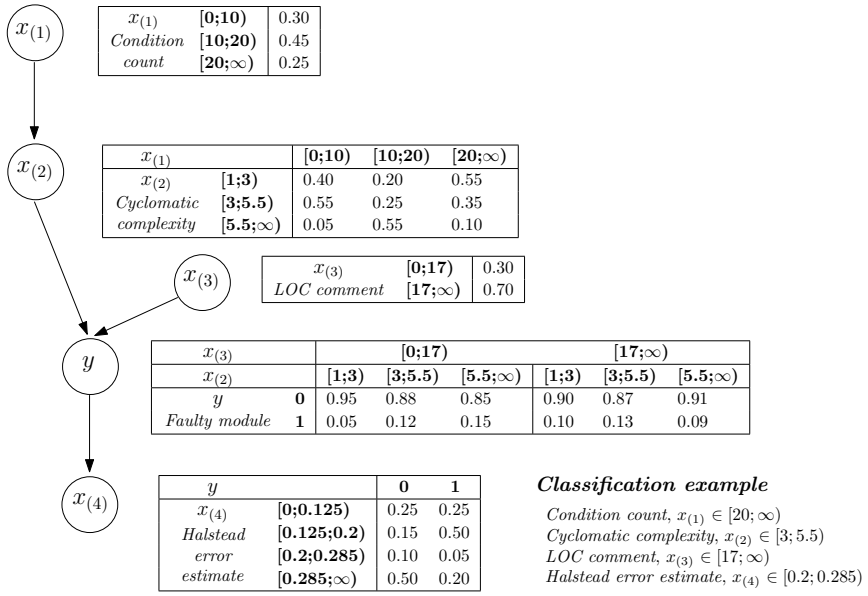


Figure 4.2: Bayesian network classification by example

A simple example of a BN classifier is given in Fig. 4.2, complemented with a example code segment which is presented to this classifier. Remark that by considering the characteristics of this segment and the information conveyed in the Bayesian network, the posterior probability that this particular instance will be faulty can be computed as follows:

$$P\left(y|x_{(1)}, x_{(2)}, x_{(3)}, x_{(4)}\right) = \frac{P\left(y, x_{(1)}, x_{(2)}, x_{(3)}, x_{(4)}\right)}{P\left(x_{(1)}, x_{(2)}, x_{(3)}, x_{(4)}\right)}. \tag{4.4}$$

It can be easily observed from Fig. 4.2 that according to the winner-takes-all rule, the code segment will be classified as being not fault prone. In what follows, several structure learning algorithms for the construction of BN classifiers will be discussed.

4.3.2 The naive bayes classifier

A first classifier built on the principle of Bayesian networks is the Naive Bayes classifier [83]. This classifier merits its connotation to the underlying assumption of conditional independence between attributes, given the class label. As a result of this assumption, the DAG associated with a Naive Bayes classifier is composed of a single parent (the unobserved class label y) and several children, each corresponding to an observed variable in the data set, Fig. 4.3. In spite of this often oversimplifying assumption, the Naive Bayes classifier typically performs surprisingly well. Domingos and Pazzani for instance found the Naive Bayes classifier to sometimes outperform a number of decision tree induction algorithms, even on data sets with considerable variable dependencies [81]. This result was also confirmed in a software fault prediction context by Menzies et al., who found the Naive Bayes classifier to outperform rule based learners [230].

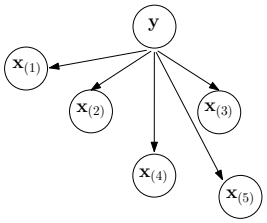


Figure 4.3: Naive Bayes network

The Naive Bayes classifier proceeds by calculating the posterior probability of each class given the vector of observed variable inputs $(x_{i(1)}, \ldots, x_{i(n)})$ of each new code segment using Bayes' rule (4.1). As a result of the conditional independence assumption, the class-conditional probabilities can be restated as:

$$P(x_{i(1)}, \ldots, x_{i(n)}|y_i = y) = \prod_{j=1}^{n} P(x_{i(j)}|y_i = y). \tag{4.5}$$

The probabilities $P(x_{i(j)}|y_i = y)$ are estimated by using frequency counts for the discrete variables and a normal or kernel density based method for continuous variables [158]. Note that as a result of the simplifying assumption of conditional independence, Naive Bayes classifiers are easy to construct since the network structure is given apriori and no structure learning phase is required. Another advantage is its computational efficiency, especially since the model has the form of a product, which can be converted into a sum by using a logarithmic transformation. In this study, both Naive Bayes using a kernel density estimate for continuous variables as well as Naive Bayes after variable discretization are considered.

4.3.3 Augmented naive bayes classifiers

The promising performance of the Naive Bayes classifier inspired several modifications to relax the conditional independence assumption. These modifications are mainly based on adding additional arcs between variables to account for dependencies present in the data or removing irrelevant or correlated variables from the network structure. A well known example is the algorithm presented

by Friedman et al., the *Tree Augmented Naive Bayes (TAN)* classifier, which allows every variable in the network to have one additional parent next to the class node [106]. One other such algorithm is the *Semi-Naive Bayesian* classifier developed by Kononenko [188] which partitions the variables into pairwise disjoint groups. The latter assumes that $x_{(j)}$ is conditionally independent of $x_{(j')}$ if and only if they belong to different groups. By contrast, the *Selective Naive Bayes* classifier tries to improve the Naive Bayes classifier by omitting certain variables to deal with strong correlation among attributes [193].

In this study, the Augmented Naive Bayes classifiers developed by Sacha [271] are used. Building upon the ideas introduced by Friedman et al., this family of Bayesian classifiers provides a further relaxation on the TAN approach: not all attributes need to be dependent on the class node and there does not necessarily need to be an undirected path between two attributes that does not pass through the class node. The Augmented Naive Bayes algorithms consist of a combination of two dependency discovery operators and two augmenting operators, summarized in Table 4.1. The measure of dependency between two attributes, the conditional mutual information $I(x_{(j)}, x_{(j')})$, which is used by both augmenting operators is defined as follows:

$$\begin{cases} \sum\limits_{x_{(j)},x_{(j')}} p\left(x_{(j)}, x_{(j')}|y\right) \log\left(\dfrac{p\left(x_{(j)}, x_{(j')}|y\right)}{p(x_{(j)}|y)p\left(x_{(j')}\right)}\right) & \text{if } y \prec x_{(j)} \\ \sum\limits_{x_{(j)},x_{(j')}} p\left(x_{(j)}, x_{(j')}|y\right) \log\left(\dfrac{p\left(x_{(j)}, x_{(j')}|y\right)}{p(x_{(j)})p\left(x_{(j')}|y\right)}\right) & \text{if } y \prec x_{(j')} \\ \sum\limits_{x_{(j)},x_{(j')}} p\left(x_{(j)}, x_{(j')}|y\right) \log\left(\dfrac{p\left(x_{(j)}, x_{(j')}|y\right)}{p(x_{(j)}|y)p\left(x_{(j')}|y\right)}\right) & \text{if } y \prec x_{(j)}, x_{(j')} \\ \sum\limits_{x_{(j)},x_{(j')}} p\left(x_{(j)}, x_{(j')}\right) \log\left(\dfrac{p\left(x_{(j)}, x_{(j')}\right)}{p(x_{(j)})p\left(x_{(j')}\right)}\right) & \text{if } y \perp x_{(j)}, x_{(j')} \end{cases} \tag{4.6}$$

Combining the dependency discovery operators with different augmenting operators from Table 4.1 yields four possible combinations. Note that both augmenting operators can also be applied directly on a Naive Bayes network, providing the following six Bayesian network classifiers:

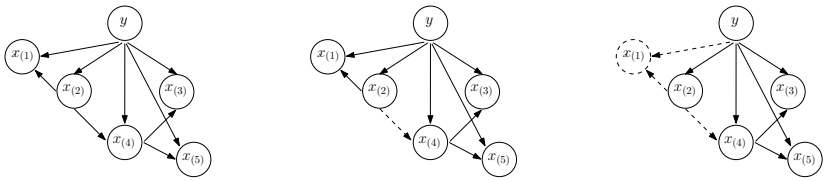
- TAN: Tree Augmented Naive Bayes
- FAN: Forest Augmented Naive Bayes
- STAN: Selective Tree Augmented Naive Bayes
- STAND: Selective Tree Augmented Naive Bayes with Discarding
- SFAN: Selective Forest Augmented Naive Bayes
- SFAND: Selective Forest Augmented Naive Bayes with Discarding

The aim of these classifiers is to find a tradeoff between the simplicity of the Naive Bayes classifiers (with a limited number of parameters) and the more realistic and complex case of full dependency between the attributes.

Dependency Discovery Op.	**Description**
Selective Augmented Naive Bayes (SAN)	Operator which connects the class node with the attributes it depends on. Starting with an empty set, SAN greedily searches for possible arcs from the class variable y to other variables $x_{(j)}$ optimizing a certain quality measure, see Table 4.2. The selected variable together with an associated arc are added to the network which is then passed to one of the *augmenting* operators. The latter establishes the dependencies among *all* variables $x_{(j)}$, irrespective of their connection with the class node.
Selective Augmented Naive Bayes with Discarding (SAND)	Operator which connects the class node with the attributes it depends on, as the SAN operator does. The difference, however, lies in that SAND will discard all variables which are not dependent on the class node before passing the network to one of the *augmenting* operators. As a result, the discarded variables are not part of the network; the difference between a network resulting from the SAN operator and SAND operator is illustrated in Fig. 4.4b and 4.4c respectively. Dashed lines indicate absent arcs or nodes in a network.
Augmenting Operators	**Description**
Tree-Augmenter	Operator which builds the maximum spanning tree among a given set of attributes. The algorithm is based on a method developed by Chow and Liu [63], but differs in the way how the mutual information is calculated. Sacha uses the conditional or unconditional probability of $x_{(j)}$ and $x_{(j')}$ depending on whether there exists an arc between the class node and the attribute (see formula 4.6). The resulting network can be regarded as a generalization of the network obtained using a TAN classifier, *not* requiring all variables to be connected with the class variable.
Forest-Augmenter	Operator which is also used to establish dependencies between attributes, but allowing for more flexibility. The forest-augmenter can create dependencies between variables in the form of a number of disjoint trees not requiring the existence of an undirected path between two attributes that does not pass through the class node. The difference between both augmenting operators is shown in Fig. 4.4a and 4.4b.

Table 4.1: Augmented Naive Bayes approach: different operators

Except for TAN, all of the above procedures adopt a quality measure to assess the fitness of a network given the data. Commonly, a distinction is made between global and local quality measures. The former evaluate the complete



(a) Selective Tree Augmented Naive Bayes network

(b) Selective Forest Augmented Naive Bayes network

(c) Selective Forest Augmented Naive Bayes with Discarding network

Figure 4.4: Examples of augmented Bayesian network structures

Quality Measures	**Description**
Standard Bayesian measure (SB)	This *global* quality measure was first proposed by [51] and is proportional to the posterior probability distribution $p(G, \Theta \mid D_{trn})$ with an added penalty term for network size. The network size or dimensionality is defined as the number of free parameters required to fully specify the joint probability distribution, $P_B(x_{(1)}, ..., x_{(n)})$.
Local Leave-One-Out-Cross Validation (LOO)	This *local* quality measure calculates iteratively the class probability conditional on the data, $P(y \mid x_{(1)}, \ldots, x_{(n)})$, using all observations minus one to estimate all parameters. The remaining observation is then used to assess the network quality in the class node [271].

Table 4.2: Augmented Naive Bayes approach: different quality measures

network while the latter only evaluate the network at the class node. As the task of software fault prediction is in fact a classification task requiring the prediction of a single class attribute, local quality measures would seem the most preferable. In this study, a representative from both categories is used, see Table 4.2. Both quality measures were combined with the five algorithms defined above, resulting in ten different BN learners.

The implementation of Sacha has been used for both the Naive Bayes and the Augmented Naive Bayes classifiers [271]. This implementation is available as a set of Weka bindings which allows to execute the software directly from within the Weka environment[4].

[4]www.jbnc.sourceforge.net

4.3.4 General Bayesian network classifiers

All previously discussed methods restrain the network structure G in order to limit the search space of allowed DAGs. Omitting these restrictions, *General Bayesian Networks (GBN)* can adopt any DAG as G. Selecting the optimal structure is however known to be an NP-hard problem since the possible sets of parents for each variable grow exponentially with the number of candidate parents [60]. Several algorithms have been proposed to limit the computational expense of finding a suitable network structure. Commonly, these algorithms can be subdivided into two broad categories; i.e. those using a heuristic search procedure ('*Search-and-Score algorithms*') and algorithms which employ statistical tests to infer the conditional independence relationships amongst variables ('*Constraint-Based algorithms*') [189].

K2

Several Search-and-Score algorithms have been proposed in the literature, e.g. the Greedy Equivalence Search (GES) algorithm [59] and algorithms based on the use of genetic operators [195]. In this study, the K2 algorithm of Cooper and Herskovits which employs a greedy search procedure is investigated [65]. While greedy search can become trapped in local minima, it has been shown that K2 yields comparable results to other Search-and-Score algorithms [338].

The K2 algorithm adopts a bottom-up search strategy assuming equal prior probabilities for all possible network structures and considers all variables one by one, assuming some ordering in the variables. For each variable $x_{(i)}$, the posterior probability of the network structure where $x_{(i)}$ is conditionally independent of all other variables is evaluated. Next, the parents whose addition increases the posterior probability of the resulting network structure the most are sequentially added. When no further parents that increase the posterior probability of the network can be added, the algorithm traces back until all variables have been considered. The K2 algorithm, available in the Weka workbench, is used in this study [334].

MMHC

Opposite to the Search-and-Score paradigm is the use of statistical tests to verify whether certain conditional independencies between variables hold. Examples are the PC algorithm [295] and the Three Phase Dependency Analysis (TPDA) algorithm [58]. In our analysis, a hybrid combining the advantages of Search-and-Score and Constraint-Based algorithms is considered; i.e. the Max-Min Hill-Climbing (MMHC) algorithm proposed in [313]. Comparison with, amongst others, GES, TPDA and PC empirically illustrated the strength of this algorithm [313].

The MMHC algorithm first constructs the skeleton of a Bayesian network (i.e. a network structure containing only undirected edges) by adopting a local

discovery algorithm called Max-Min Parents and Children (MMPC) to determine the parent-children set (**PC**) of every node. MMPC proceeds by sequentially adding nodes to a candidate **PC** set (**CPC**) as selected by a heuristic procedure. The set may contain false positives, which are removed in a second step. The algorithm tests whether any variable in **CPC** is conditionally independent of the target variable, given a blocking set $\mathbf{S} \subseteq \mathbf{CPC}$. If such variables are found, they are removed from **CPC**. As a measure of conditional (in)dependence, the G^2 measure, as described by Spirtes et al. [295], is used. This measure is asymptotically following a χ^2 distribution with appropriate degrees of freedom under the null hypothesis of conditional independence, which allows to calculate a p-value indicating the probability of falsely rejecting this null hypothesis. Conditional independence is assumed when the p-value is more than the significance level α (α equals 0.15 in this study). Once the skeleton is determined, the final network structure is learned using a greedy hill-climbing search which is constrained to add only an edge if it was discovered by MMPC. The *BDeu* score [136] is used to guide this greedy search. The network structure is induced using the Causal Explorer package for Matlab[5] while the Bayesian Net Toolbox[6] is used for inference afterwards.

4.3.5 Benchmark classifiers

As illustrated by Fig. 4.1, numerous techniques other than BN classifiers have been used to construct software fault prediction models. As a reference, two such techniques are included; i.e. random forest and logistic regression which are both implemented in the Weka toolbox [334]. These techniques are selected on the basis of illustrated performance in software fault prediction and other domains [23, 200], and were already discussed in Chapter 2.

4.3.6 Markov blanket feature selection

Learning from high dimensional data often poses considerable difficulties to machine learning techniques due to the presence of irrelevant or redundant features. Moreover, when considering more features, typically comparatively more parameters need to be estimated which in turn induces additional uncertainty in these estimations [301].

Previous work on mining static code features indicated that a single best set of features does not exist, but instead depends on the specific data set [230]. As a result, several software fault prediction studies consider the complete input space and let the learner decide which features should be selected [200,317]. Such approach is often feasible as most techniques include some sort of embedded feature selection, or can be adjusted to this goal by e.g. including a penalty on the size of the parameters [6]. This is however not the case for the Naive Bayes classifier and some of the Augmented Naive Bayes Classifiers which thus also include uninformative variables.

[5] www.dsl-lab.org/causal_explorer
[6] www.code.google.com/p/bnt

The use of a Markov blanket based feature selection approach provides a natural solution to this issue. The Markov blanket (MB) of a node y is the union of y's parents, y's children and the parents of y's children and is the minimal variable subset conditioned on which all other variables are independent of y. In other words, no other variables than those contained in the MB of y need to be observed to predict the value of y. The concept is illustrated in Fig. 4.5 where the MB of y is indicated by the shaded area. For instance, the value of $x_{(6)}$ can be ignored when predicting the value of y as it is the child of a parent of y and thus is no part of the MB of y.

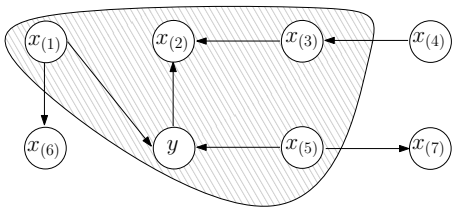


Figure 4.5: The Markov blanket of a classification node **y**

The HITON algorithm is used for the Markov blanket feature selection which adopts the same test of conditional (in)dependence as the MMHC algorithm, the G^2 measure, and has been applied to the data sets at a significance level of 5% and 15%, referred to as MB.05 and MB.15 respectively [7]. Note that if attribute selection is performed, it is applied prior to training and testing the classifiers. Hence, every classifier is applied three times to each data set. The feature selection algorithm has been implemented in the *Causal Explorer* package.

4.4 Empirical setup

4.4.1 Data sets

The data considered in this study stems from two independent sources; i.e. from the NASA IV&V facility and the open source Eclipse Foundation. Both data sources are in the public domain, enabling researchers to validate our findings. Note that the set of static code features is not homogenous, including McCabe complexity, Halstead, object oriented (OO) and lines of code (LOC) metrics, depending on the origin of the data set. Notwithstanding this dissimilarity, the purpose of both data collection efforts is to investigate the relationship between static code features and software faults.

It should be noted that static code features are known to be correlated; previous work examining the different static code features e.g. indicated that these could be grouped into four categories [202]. A first category related to metrics derived from flowgraphs (i.e. McCabe metrics) while a second category contained metrics related to the size and item count of a program. The two other categories represented different types of Halstead metrics. This again motivates the use of a feature selection procedure, especially when applying techniques that do not include some sort of embedded feature selection.

NASA IV&V facility

The NASA data sets can be freely obtained directly from the NASA MDP (Metrics Data Program) repository which is hosted at the NASA IV&V facility website[7] or from the Promise repository[8]. Recently, it was pointed out that differences exist between the data from both sources, a topic to which we will return in Chapter 5. In this study, eight data sets taken from the NASA MDP repository have been preprocessed as detailed in Section 4.4.2 and studied. As machine learning typically benefits from more data, only data sets with more than 1,000 observations have been selected, see Table 4.3, top panel.

Table 4.4 provides an overview of all available features for each of the NASA data sets included in this study and indicates how they relate to each other. As noted earlier, such data can be mined directly from the source code using several purpose-built tools. The tool selected for this task was McCabe IQ 7.1 and has been used for all data sets, providing a common measurement framework. The set of available static code features include LOC, Halstead and McCabe complexity metrics. The first is arguably one of the widest used proxies for software complexity in fault prediction studies and has been used as an approximation of software size since the late sixties [94]. As LOC counts have been recognized to be dependent on the selected programming language, a number of alternative measures were introduced in the 70s to quantify software complexity. Two such sets of metrics are McCabe complexity metrics and Halstead software science metrics. The first maps a program or module to a flowchart where each node corresponds to a block of code where the flow is sequential and the arcs correspond to branches in the program. Software complexity is then related to the number of linearly independent paths through a program. Halstead metrics take a different perspective by considering a program or module as a sequence of tokens, i.e. a sequence of operators and operands. Based on the counts of these tokens, a number of derivative measures have been defined which are sometimes referred to as 'software science' metrics [122]. Note that the projects stemming from the NASA IV&V facility were mainly developed using procedural programming, and typically only contain LOC, Halstead and McCabe complexity metrics. For some projects, requirement metrics (projects 'PC1', 'CM1' and 'JM1') [155] and class level metrics (project 'KC1') [53] have also been collected. These additional metrics have not been considered in this study as these have not been collected on the same granularity level as the rest of the data.

Eclipse foundation

The Eclipse platform project was founded in 2001 by IBM with the support of a consortium of software vendors. In early 2004, the Eclipse Foundation

[7] www.mdp.ivv.nasa.gov
[8] www.promisedata.org

was instated to support the growing Eclipse community which constitutes both individuals and companies[9]. Data from three major releases (release 2.0, 2.1 and 3.0) have been collected by the University of Saarland on the granularity of files and packages [345]. As fault prediction models built on a finer granularity provide more information to developers, only file level data sets are considered in this study. More information on the origin of the Eclipse data sets can be found in Table 4.3, bottom panel.

Static code features have been collected using the built-in Java parser of Eclipse; some features were only collected at a finer granularity (i.e. at the granularity of methods or classes) and were thus aggregated taking the average, total and maximum value of the metrics. Table 4.5 provides an overview of all available features. These include LOC and McCabe complexity metrics as well as counts on the use of object oriented constructs. The source code is matched with 6 months of post release failure data from the Eclipse bug repository [345].

NASA	*Origin data set*	*Project size*	*# faulty modules / # modules*
JM1	Real time project in C with eight years of defect data associated	315 KSLOC	2,102 / 10,878 (19.32 %)
KC1	Storage management system for ground data in C++ with five years of defect data associated	43 KSLOC	325 / 2,107 (15.42 %)
MC1	Software developed in C & C++ for a combustion experiment on the space shuttle	63 KLOC	68 / 4,625 (1.47 %)
PC1	Flight software from an earth orbiting satellite in C which is no longer operational	40 KLOC	76 / 1,059 (7.18 %)
PC2	Software of a dynamic simulator for attitude control systems in C	26 KLOC	23 / 4,505 (0.51 %)
PC3	Flight software from an earth orbiting satellite in C	40 KLOC	160 / 1,511 (10.59 %)
PC4	Flight software from an earth orbiting satellite in C	36 KLOC	178 / 1,347 (13.21 %)
PC5	Software of safety enhancements of a cockpit upgrade in C++	164 KLOC	503 / 15,414 (3.26 %)
Eclipse	*Origin data set*	*Project size*	*# faulty files / # files*
Ecl2.0a	The Eclipse platform 2.0 was released on 27^{th} of June 2002	796.9 KLOC	975 / 6,729 (14.49 %)
Ecl2.1a	The Eclipse platform 2.1 was released on 27^{th} of March 2003	987.6 KLOC	854 / 7,888 (10.83 %)
Ecl3.0a	The Eclipse platform 3.0 was released on 25^{th} of June 2004	1,305.9 KLOC	1,568 / 10,593 (14.80 %)

Table 4.3: Overview of data sets: Origin

[9] www.eclipse.org

Table 4.4: Overview of NASA data sets: Selection of attributes and their calculation

Data sets *Metrics Data Program (MDP) repository*			JM1	KC1	MC1	PC1	PC2	PC3	PC4	PC5
LOC based metrics	*Handle*	*Calculation*								
LOC_Total	LOC		✓	✓	✓	✓	✓	✓	✓	✓
LOC_Blank	BLOC		✓	✓	✓	✓		✓	✓	✓
LOC_Executable	SLOC		✓	✓	✓	✓	✓	✓	✓	✓
LOC_Comments	CLOC		✓	✓	✓	✓	✓	✓	✓	✓
LOC_Code_and_Comment	C&SLOC		✓	✓	✓	✓	✓	✓	✓	✓
Number_of_Lines	nl				✓	✓	✓	✓	✓	✓
Percent_Comments	% Comments	$\frac{\text{CLOC+C\&SLOC}}{\text{SLOC+CLOC+C\&SLOC}}$			✓	✓	✓	✓	✓	✓
McCabe metrics	*Handle*	*Calculation*								
Cyclomatic_Complexity	$v(G)$		✓	✓	✓	✓	✓	✓	✓	✓
Cyclomatic_Density	$vd(G)$				✓	✓	✓	✓	✓	✓
Decision_Density	$dd(G)$	$\frac{\text{Cond\_C}}{\text{Dec\_C}}$				✓	✓	✓	✓	
Design_Complexity	$iv(G)$		✓	✓	✓	✓	✓	✓	✓	✓
Design_Density	$id(G)$	$\frac{iv(G)}{v(G)}$			✓	✓	✓	✓	✓	✓
Essential_Complexity	$ev(G)$		✓	✓	✓	✓	✓	✓	✓	✓
Essential_Density	$ed(G)$	$\frac{ev(G)-1}{v(G)-1}$			✓	✓	✓	✓	✓	✓
Global_Data_Complexity	$gdv(G)$				✓					✓
Global_Data_Density	$gd(G)$	$\frac{gdv(G)}{v(G)}$			✓					✓
Norm_Cyclomatic_Compl	*Norm* $v(G)$	$\frac{v(G)}{nl}$			✓	✓	✓	✓	✓	✓
Maintenance Severity	Maint_Sev	$\frac{ev(G)}{v(G)}$			✓	✓	✓	✓	✓	✓
Halstead metrics	*Handle*	*Calculation*								
Num_Operators	N_1		✓	✓	✓	✓	✓	✓	✓	✓
Num_Operands	N_2		✓	✓	✓	✓	✓	✓	✓	✓
Num_Uniq_Operators	n_1		✓	✓	✓	✓	✓	✓	✓	✓
Num_Uniq_Operands	n_2		✓	✓	✓	✓	✓	✓	✓	✓
Length	N	$N_1 + N_2$	✓	✓	✓	✓	✓	✓	✓	✓
Difficulty	D	$\frac{n_1 \times N_2}{2 \times n_2}$	✓	✓	✓	✓	✓	✓	✓	✓
Level	L	$\frac{1}{\text{D}}$	✓	✓	✓	✓	✓	✓	✓	✓
Volume	V	$\text{N} \times \log_2(n_1 + n_2)$	✓	✓	✓	✓	✓	✓	✓	✓
Programming_Effort	E	D×V	✓	✓	✓	✓	✓	✓	✓	✓
Programming_Time	T	$\frac{\text{E}}{18}$	✓	✓	✓	✓	✓	✓	✓	✓
Error_Estimate	B	$\frac{\text{V}}{3000}$	✓	✓	✓	✓	✓	✓	✓	✓
Content	I	$\frac{\text{V}}{\text{D}}$	✓	✓	✓	✓	✓	✓	✓	✓
Miscellaneous metrics	*Handle*	*Calculation*								
Branch_Count	Branch_C		✓	✓	✓	✓	✓	✓	✓	✓
Call_Pairs	Call_C				✓	✓	✓	✓	✓	✓
Condition_Count	Cond_C				✓	✓	✓	✓	✓	✓
Decision_Count	Dec_C				✓	✓	✓	✓	✓	✓
Edge_Count	Edge_C				✓	✓	✓	✓	✓	✓
Node_Count	Node_C				✓	✓	✓	✓	✓	✓
Parameter_Count	Parameter_C				✓	✓	✓	✓	✓	✓
Multiple_Condition_Count	Mul_Cond_C				✓	✓	✓	✓	✓	✓
Modified_Condition_Count	Mod_Cond_C				✓	✓	✓	✓	✓	✓

4.4.2 Data preprocessing

To enable a fair assessment of the learners discussed earlier, each of the the eleven data sets are subjected to the following preprocessing procedure. Each

Data sets *Eclipse foundation*

Method level metrics	*Handle*	*Available aggregators*
Fan out	FOUT	avg, max, total
Method lines of code	MLOC	avg, max, total
Nested block depth	NBD	avg, max, total
Number of parameters	PAR	avg, max, total
Cyclomatic complexity	VG	avg, max, total
Classes level metrics	*Handle*	*Available aggregators*
Number of fields	NOF	avg, max, total
Number of methods	NOM	avg, max, total
Number of static fields	NSF	avg, max, total
Number of static methods	NSM	avg, max, total
File level metrics	*Handle*	*Available aggregators*
Number of anonymous type declarations	ACD	value
Number of interfaces	NOI	value
Number of classes	NOT	value
Total lines of code	TLOC	value

Table 4.5: Overview of Eclipse data sets: Selection of attributes

observation (software module or file) in the data sets consists of a unique ID, several static code features and an error count. First, the data used to learn and validate the models are selected and thus, the *ID* as well as attributes exhibiting zero variance are discarded. Moreover, observations with a total line count of zero are deemed logically incorrect and are removed. In case of the NASA data sets, the *error density* is also removed. The *error count* is discretized into a boolean value where 0 indicates that no errors were recorded for this software module or file and 1 otherwise, in line with e.g. [200, 230, 315, 317, 320].

As some of the Bayesian learners are unable to cope with continuous features, a discretized version of each data set was constructed using the algorithm of Fayyad and Irani [93]. This supervised discretization algorithm uses entropy to select subintervals that are as pure as possible with respect to the target attribute. Most techniques use the discretized data sets; if a technique employs the continuous data instead, it is labeled accordingly.

Finally, it should be noted that machine learning techniques typically perform better if more data to learn from are available. On the other hand, part of the data needs to be put aside as an independent test set in order to provide a realistic assessment of the performance. As can be seen from Table 4.3, the smallest data set contains 1,059 observations, while the largest contains up to 15,414 observations. Each of the data sets is randomly partitioned into two disjoint sets, i.e. a training and test set consisting of respectively 2/3 and 1/3 of the observations, using stratified sampling in order to preserve the class distribution. To account for a potential sampling bias, this partitioning procedure

is repeated ten times. Please note that as a side benefit of the automated collection of the static code features, the data sets are complete; i.e. there is no need for missing value handling.

After performing these steps, the data sets are passed to the learners described in Section 4.3 with and without first applying the Markov blanket feature selection procedure.

4.4.3 Classifier evaluation

The induced models are compared to each other in terms of classification performance, Section 2.3.1, and comprehensibility, Section 2.3.1. Note that the latter is often neglected during model selection, but is of critical importance when building software fault prediction models in practice [95].

Classifier performance

Chapter 2 told us that a battery of classification performance measures have been put forward, of which ROC analysis is an often considered member. More specifically, Table 2.3 shows that about half of the most recent papers on software fault prediction embraced the AUC for model evaluation; Eq. 2.24 already defined the AUC, drawing upon its equivalence to the Wilcoxon ranked sum test. Let $f_l(s)$ be the probability density function of the scores s for the classes $l \in \{0,1\}$[10], and $F_l(s)$ the corresponding cumulative distribution function. Then, AUC can also be formulated as:

$$AUC = \int_{-\infty}^{\infty} F_0(s) f_1(s) ds. \tag{4.7}$$

Although the AUC has been extensively used, it was pointed out that the AUC is flawed as a measure of aggregated classification performance [125]. He developed a performance measure based on the expected minimum misclassification loss, whereby the misclassification costs are not exactly known but follow a probability distribution. Assume that misclassifying a faulty instance as not fault prone has a misclassification cost c_0, whereas a fault free instance classified as fault prone costs c_1. Then, the expected minimum misclassification loss is defined as:

$$L = \mathrm{E}[b] \int_0^1 \left[c\pi_0(1 - F_0(T)) + (1-c)\pi_1 F_1(T) \right] u(c) \mathrm{d}c, \tag{4.8}$$

with π_0 and π_1 the prior probabilities for fault prone and not fault prone instances respectively, and T the optimal threshold t for a given value of c. Moreover, a variable transformation for the cost parameters has been applied, implying $b = c_0 + c_1$ and $c = c_0/(c_0 + c_1)$. The probability distribution of c (the ratio

[10] Note that for notational convenience, it will be assumed that higher scores correspond to fault free instances; if this is not the case, the scores need to be multiplied by minus one.

of the costs) is given by $u(c)$, and is assumed independent from the distribution of b (the level of the costs) leading to the expected value of b, E[b], outside the integral. Hand has shown that an AUC based ranking is equivalent to a ranking based on the expected minimum misclassification loss, for an appropriate choice of $u(c)$. The problem is that the probability distribution implied by the AUC measure varies with the empirical score distribution and thus with the classifiers. However, beliefs on the likely values of c should depend on contextual information, not on the used classification tools. Therefore, Hand proposes the H-measure which takes a beta distribution with parameters α and β for $u(c)$, and is defined by:

$$H = 1 - \frac{\int_0^1 \left[c\pi_0(1 - F_0(T)) + (1 - c)\pi_1 F_1(T)\right] u_{\alpha,\beta}(c)\mathrm{d}c}{\pi_0 \int_0^{\pi_1} c \cdot u_{\alpha,\beta}(c)\mathrm{d}c + \pi_1 \int_{\pi_1}^1 (1 - c) \cdot u_{\alpha,\beta}(c)\mathrm{d}c}. \tag{4.9}$$

This is a normalized measure based on the expected minimum misclassification loss, ranging from zero for a random classifier to one for a perfect classifier. Hand gives a number of examples which clearly illustrate how the AUC implies cost distributions which vary between classifiers [125].

Recently, an alternative and coherent interpretation of the AUC as a measure of aggregated classification performance was put forward by Flach et al. [100], relaxing the assumption made by Hand of selecting the optimal threshold T for a given value of c. More specifically, they showed that the AUC can be reformulated as being linearly related to expected misclassification loss by instead of selecting this optimal threshold T, considering as many thresholds as there are examples, taking a uniform distribution over the data points and setting the threshold equal to the score of the selected instance, $t = s(\mathbf{x}_i)$. In fact, recent work connected the AUC with 0-1 loss, Brier score and other often used metrics via the concept of operating condition, to which we return upon proposing an update to the benchmarking framework of Lessmann et al. in Section 5.3. The H-measure on the other hand has the benefit of explicitly balancing the losses arising from classifying fault prone as not fault prone instances against the opposite type of misclassification, an aspect the AUC does not allow for.

In this study, the performance of the classification algorithms will be quantified using both measures. The AUC will be reported to verify the results of other studies and is shown to be less discriminative as the H-measure, supporting earlier findings in the literature [200, 231]. We also report the H-measure and the impact of varying the parameters for the beta distribution underlying the H-measure.

H-measure parameters

When no additional knowledge of the likely values of c is available, Hand proposes to use a symmetric beta distribution with $\alpha = \beta = 2$. As no specific costs have been specified in the fault prediction literature, the H-measure will be calculated with these default values. However, depending on the context, it can be argued that misclassifying a faulty instance as non fault prone is more serious than the opposite, e.g. when considering high risk software. On the

other hand, e.g. in open source software development, the opposite is true: in order to keep participants motivated, it is advised to release early and often and thus the cost of missing defects is perhaps lower than the cost of delays due to unnecessary testing [266]. It can be seen from Table 4.3 that the data obtained from NASA relate to high risk projects while the Eclipse project is an example of the latter.

In [153], the impact of different cost ratios using the MetaCost framework of Domingos was investigated. Misclassification costs ranging from $(c_0, c_1) = (75, 1)$, i.e. risk averse, to $(c_0, c_1) = (1, 75)$, i.e. delay averse, were selected. A similar approach is followed in this study, allowing c_0 (resp. c_1) to take discrete values from 1 to 75 while keeping the opposite misclassification cost equal to one. As such, the robustness of the H-measure with respect to changes in the software development context is investigated. The findings of this analysis are reported in Section 4.5.

Classifier comprehensibility

As BN classifiers, with the exception of the Naive Bayes learner with kernel density estimation, share the possibility to be represented as a combination of a DAG and associated conditional probability tables, the discriminating aspect lies in the complexity of the network structure and the number of entries in the probability table. Furthermore, Section 2.3.1 argued that smaller, less complex models are to be preferred. As such, the complexity of the network structure of each algorithm with and without Markov blanket feature selection is quantified by the number of nodes and arcs in the DAG. Note that Naive Bayes and certain Augmented Naive Bayesian learners are unable to exclude variables from the network structure and thus invariably contain as many nodes as there are variables in the data set. The aggregated size of the probability table is measured by the network dimensionality. This is defined as the number of parameters required to fully specify the joint probability distribution encoded by the network and is calculated as:

$$DIM = \sum_{j=1}^{n} (r_j - 1) \cdot q_j \tag{4.10}$$

with r_j being the cardinality of variable $x_{(j)}$ and:

$$q_j = \prod_{x_{(j')} \in \Pi_{x_{(j)}}} r_{j'} \tag{4.11}$$

with $\Pi_{x_{(j)}}$ the direct parent set for node $x_{(j)}$. Note that for Naive Bayes using a kernel density estimate, the network dimensionality cannot be calculated in a meaningful way and thus is excluded from this comparison.

4.4.4 Statistical testing

The statistical testing framework of Demšar is again adopted in analysing the results of this study [77]. In a first step, the Friedman test, see Eq. 2.16, is used to investigate whether classification performance is influenced by specific factors. Two factors are of interest here; i.e. the type of (Bayesian Network) classifier and the use of feature selection prior to model construction. Adopting the notation of Section 2.3.3, k equals 3 and P equals $11{\times}17 = 187$ when comparing the results without feature selection with those after applying MB.15 and MB.05. As the MB.05 feature selection procedure turned out to negatively impact predictive performance, $k = 17$ and $P = 11{\times}2 = 22$ when analyzing the impact of classifiers. Upon rejection of the Friedman null hypothesis, an appropriate post-hoc test is effected. As the goal is to compare the BN learners amongst themselves and against the benchmark classifiers of Section 4.3.5, a post-hoc Nemenyi test is applied which compares all treatments to each other [246].

During the comprehensibility assessment, a post-hoc Bonferroni-Dunn test [84] is adopted which compares all classifiers with the single best performing classifier. This test is similar to the Nemenyi test but adjusts the confidence level to control for family-wise testing. As explained before, only for a selection of BN learners it is possible to calculate the network dimension in a meaningful way and thus, k equals 14 and P equals $11 \times 2 = 22$ in this situation. The Bonferroni-Dunn test has been discussed in Section 3.4.6.

4.5 Results and discussion

This section reports the results of the techniques discussed in Section 4.3. The average performance (the shaded rows) and standard deviation of the ten independent iterations in terms of AUC and H-measure taking $\beta(2, 2)$ as distribution parameters are presented in Tables 4.6 and 4.7, respectively. The upper panel displays the results prior to Markov blanket feature selection while the bottom panel shows the performance after Markov blanket (MB) feature selection at a significance level of 15%. The last column of each table displays the Average Rank (AR) of each technique. The best performing technique is reported in bold and underlined. The AR of a technique that is not significantly different from the best performing technique at 5% is tabulated in boldface font, while results significantly different at 1% are displayed in italic script. Classifiers differing at the 5% level but not at the 1% level are displayed in normal script. The Bonferroni-Dunn test is used during these assessments.

4.5.1 Empirical results

The results without MB feature selection and with MB.15 and MB.05 are first compared to each other using a Friedman test. The outcome of this test indicated that feature selection did have a significant impact on the results (p-value

Table 4.6: Comparison of classifier performance: out-of-sample AUC performance

Technique / **Data set**	**JM1**	**KC1**	**MC1**	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**	**Ecl2.0a**	**Ecl2.1a**	**Ecl3.0a**	**AR**	
Log. Regr.	0.71	0.79	0.81	0.78	0.70	0.79	0.89	0.95	0.80	0.74	0.76	9.00	
using continuous data	*0.010*	*0.018*	*0.073*	*0.046*	*0.126*	*0.037*	*0.020*	*0.011*	*0.007*	*0.010*	*0.007*		
RndFor	0.74	0.82	0.91	0.84	0.73	0.82	0.93	0.97	0.82	0.75	0.77	**2.18**	
using continuous data	*0.006*	*0.015*	*0.041*	*0.022*	*0.129*	*0.024*	*0.010*	*0.005*	*0.008*	*0.011*	*0.004*		
Naive Bayes	0.70	0.80	0.86	0.77	0.84	0.76	0.82	0.95	0.79	0.73	0.77	*10.91*	
	0.009	*0.021*	*0.030*	*0.044*	*0.097*	*0.027*	*0.018*	*0.006*	*0.006*	*0.012*	*0.006*		
Naive Bayes kernel	0.69	0.80	0.81	0.77	0.81	0.77	0.79	0.95	0.80	0.74	0.76	*12.18*	
using continuous data	*0.010*	*0.020*	*0.057*	*0.038*	*0.129*	*0.044*	*0.018*	*0.008*	*0.006*	*0.012*	*0.006*		
TAN	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**7.18**	
	0.009	*0.014*	*0.035*	*0.041*	*0.099*	*0.028*	*0.007*	*0.005*	*0.012*	*0.015*	*0.006*		
FAN-SB	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**7.68**	
	0.009	*0.014*	*0.035*	*0.041*	*0.099*	*0.028*	*0.007*	*0.005*	*0.012*	*0.015*	*0.006*		
FAN-LCV_LO	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**7.64**	
	0.009	*0.014*	*0.043*	*0.041*	*0.099*	*0.028*	*0.007*	*0.005*	*0.012*	*0.015*	*0.006*		
SFAN-SB	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**6.95**	
	0.009	*0.014*	*0.036*	*0.041*	*0.099*	*0.028*	*0.007*	*0.005*	*0.012*	*0.015*	*0.006*		
SFAN-LCV_LO	0.72	0.78	0.88	0.80	0.84	0.78	0.90	0.96	0.79	0.74	0.76	**7.82**	
	0.010	*0.018*	*0.030*	*0.034*	*0.104*	*0.023*	*0.016*	*0.007*	*0.009*	*0.011*	*0.007*		
SFAND-SB	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**6.86**	
	0.009	*0.014*	*0.036*	*0.041*	*0.099*	*0.028*	*0.007*	*0.005*	*0.012*	*0.015*	*0.006*		
SFAND-LCV_LO	0.72	0.78	0.88	0.80	0.84	0.78	0.90	0.96	0.79	0.74	0.76	**7.50**	
	0.010	*0.018*	*0.030*	*0.034*	*0.104*	*0.023*	*0.016*	*0.007*	*0.009*	*0.011*	*0.007*		
STAN-SB	0.70	0.75	0.84	0.74	0.62	0.79	0.89	0.95	0.77	0.72	0.75	*14.27*	
	0.008	*0.019*	*0.048*	*0.106*	*0.201*	*0.025*	*0.010*	*0.011*	*0.010*	*0.015*	*0.009*		
STAN-LCV_LO	0.71	0.77	0.87	0.78	0.79	0.79	0.90	0.96	0.79	0.73	0.76	*10.36*	
	0.011	*0.015*	*0.048*	*0.056*	*0.095*	*0.023*	*0.014*	*0.008*	*0.007*	*0.013*	*0.009*		
STAND-SB	0.71	0.79	0.88	0.81	0.83	0.79	0.89	0.97	0.80	0.74	0.75	**6.59**	
	0.009	*0.014*	*0.036*	*0.041*	*0.099*	*0.028*	*0.007*	*0.005*	*0.012*	*0.015*	*0.006*		
STAND-LCV_LO	0.72	0.78	0.88	0.80	0.83	0.78	0.90	0.96	0.79	0.74	0.76	**8.41**	
	0.010	*0.018*	*0.030*	*0.034*	*0.101*	*0.023*	*0.016*	*0.007*	*0.009*	*0.011*	*0.007*		
K2	0.70	0.78	0.88	0.82	0.83	0.78	0.89	0.97	0.79	0.72	0.74	*11.36*	
	0.011	*0.018*	*0.041*	*0.055*	*0.127*	*0.025*	*0.010*	*0.006*	*0.007*	*0.011*	*0.008*		
MMHC15	0.71	0.72	0.69	0.72	0.61	0.77	0.88	0.95	0.71	0.69	0.74	*16.09*	
	0.012	*0.092*	*0.150*	*0.064*	*0.119*	*0.022*	*0.021*	*0.010*	*0.086*	*0.063*	*0.046*		
Log. Regr.	0.71	0.79	0.82	0.77	0.80	0.79	0.87	0.95	0.80	0.73	0.76	**11.09**	
using continuous data	*0.009*	*0.021*	*0.051*	*0.052*	*0.123*	*0.035*	*0.019*	*0.011*	*0.007*	*0.009*	*0.008*		
RndFor	0.74	0.80	0.92	0.81	0.66	0.78	0.89	0.97	0.82	0.73	0.77	**5.82**	
using continuous data	*0.007*	*0.021*	*0.047*	*0.037*	*0.129*	*0.053*	*0.036*	*0.003*	*0.007*	*0.016*	*0.006*		
Naive Bayes	0.70	0.79	0.87	0.77	0.82	0.79	0.85	0.95	0.79	0.73	0.76	**9.91**	
	0.008	*0.020*	*0.035*	*0.045*	*0.096*	*0.022*	*0.014*	*0.006*	*0.006*	*0.014*	*0.005*		
Naive Bayes kernel	0.69	0.80	0.81	0.75	0.79	0.78	0.80	0.95	0.79	0.74	0.76	12.91	
using continuous data	*0.011*	*0.021*	*0.051*	*0.058*	*0.158*	*0.037*	*0.012*	*0.008*	*0.006*	*0.012*	*0.007*		
TAN	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**6.36**	
	0.009	*0.018*	*0.039*	*0.039*	*0.105*	*0.030*	*0.016*	*0.005*	*0.010*	*0.013*	*0.008*		
FAN-SB	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**6.14**	
	0.009	*0.018*	*0.039*	*0.039*	*0.105*	*0.030*	*0.016*	*0.005*	*0.010*	*0.013*	*0.008*		
FAN-LCV_LO	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**6.27**	
	0.009	*0.018*	*0.039*	*0.038*	*0.105*	*0.030*	*0.016*	*0.005*	*0.010*	*0.013*	*0.008*		
SFAN-SB	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**7.14**	
	0.009	*0.018*	*0.040*	*0.040*	*0.105*	*0.030*	*0.016*	*0.005*	*0.010*	*0.013*	*0.008*		
SFAN-LCV_LO	0.72	0.79	0.87	0.78	0.81	0.79	0.90	0.96	0.79	0.74	0.76	**7.55**	
	0.010	*0.017*	*0.038*	*0.041*	*0.100*	*0.029*	*0.014*	*0.006*	*0.011*	*0.012*	*0.005*		
SFAND-SB	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**7.14**	
	0.009	*0.018*	*0.040*	*0.040*	*0.105*	*0.030*	*0.016*	*0.005*	*0.010*	*0.013*	*0.008*		
SFAND-LCV_LO	0.72	0.79	0.87	0.78	0.81	0.79	0.90	0.96	0.79	0.74	0.76	**7.73**	
	0.010	*0.017*	*0.038*	*0.041*	*0.100*	*0.029*	*0.014*	*0.006*	*0.011*	*0.012*	*0.005*		
STAN-SB	0.70	0.75	0.83	0.76	0.75	0.78	0.89	0.95	0.77	0.71	0.75	*15.36*	
	0.008	*0.032*	*0.079*	*0.044*	*0.109*	*0.021*	*0.014*	*0.010*	*0.009*	*0.017*	*0.009*		
STAN-LCV_LO	0.71	0.78	0.87	0.78	0.82	0.79	0.89	0.96	0.79	0.74	0.76	**8.73**	
	0.011	*0.027*	*0.034*	*0.040*	*0.093*	*0.027*	*0.017*	*0.008*	*0.008*	*0.015*	*0.008*		
STAND-SB	0.71	0.80	0.88	0.78	0.81	0.79	0.90	0.97	0.80	0.74	0.76	**7.32**	
	0.009	*0.018*	*0.040*	*0.040*	*0.105*	*0.030*	*0.016*	*0.005*	*0.010*	*0.013*	*0.008*		
STAND-LCV_LO	0.72	0.79	0.87	0.78	0.81	0.79	0.90	0.96	0.79	0.74	0.76	**7.91**	
	0.010	*0.017*	*0.038*	*0.041*	*0.098*	*0.029*	*0.014*	*0.006*	*0.011*	*0.012*	*0.005*		
K2	0.70	0.79	0.88	0.77	0.81	0.78	0.89	0.97	0.80	0.73	0.74	**10.00**	
	0.012	*0.019*	*0.041*	*0.038*	*0.103*	*0.028*	*0.016*	*0.006*	*0.006*	*0.015*	*0.009*		
MMHC15	0.71	0.74	0.84	0.77	0.77	0.77	0.88	0.95	0.75	0.71	0.75	*15.64*	
	0.012	*0.060*	*0.067*	*0.040*	*0.115*	*0.024*	*0.013*	*0.011*	*0.050*	*0.042*	*0.010*		

Without Markov Blanket feature selection

With Markov Blanket feature selection at 15%

Table 4.7: Comparison of classifier performance: out-of-sample H-measure performance

Technique / **Data set**	**JM1**	**KC1**	**MC1**	**PC1**	**PC2**	**PC3**	**PC4**	**PC5**	**Ecl2.0a**	**Ecl2.1a**	**Ecl3.0a**	**AR**	
Log. Regr.	0.113	0.193	0.197	0.115	0.022	0.124	0.367	0.270	0.183	0.094	0.126	**5.09**	*Without* Markov Blanket feature selection
using continuous data	*0.0129*	*0.0273*	*0.1079*	*0.0555*	*0.0452*	*0.0468*	*0.0453*	*0.0235*	*0.0076*	*0.0129*	*0.0095*		
RndFor	0.136	0.202	0.379	0.233	0.007	0.171	0.430	0.380	0.220	0.088	0.154	**2.45**	
using continuous data	*0.0099*	*0.0365*	*0.0881*	*0.0609*	*0.0079*	*0.0334*	*0.0431*	*0.0344*	*0.0141*	*0.0076*	*0.0109*		
Naive Bayes	0.093	0.159	0.198	0.108	0.010	0.082	0.205	0.157	0.162	0.093	0.134	*10.73*	
	0.0090	*0.0315*	*0.0980*	*0.0573*	*0.0087*	*0.0246*	*0.0224*	*0.0111*	*0.0142*	*0.0118*	*0.0087*		
Naive Bayes kernel	0.087	0.163	0.010	0.098	0.023	0.104	0.149	0.217	0.164	0.087	0.120	*10.36*	
using continuous data	*0.0089*	*0.0323*	*0.0050*	*0.0544*	*0.0228*	*0.0363*	*0.0416*	*0.0249*	*0.0132*	*0.0115*	*0.0067*		
TAN	0.098	0.151	0.235	0.124	0.010	0.097	0.289	0.280	0.170	0.087	0.119	**7.45**	
	0.0072	*0.0203*	*0.0789*	*0.0467*	*0.0145*	*0.0317*	*0.0178*	*0.0170*	*0.0230*	*0.0113*	*0.0087*		
FAN-SB	0.098	0.151	0.235	0.124	0.010	0.096	0.289	0.280	0.170	0.087	0.119	**8.23**	
	0.0072	*0.0203*	*0.0789*	*0.0466*	*0.0143*	*0.0314*	*0.0178*	*0.0170*	*0.0230*	*0.0113*	*0.0087*		
FAN-LCV_LO	0.098	0.151	0.235	0.124	0.010	0.097	0.289	0.280	0.170	0.087	0.119	**7.82**	
	0.0072	*0.0203*	*0.0788*	*0.0467*	*0.0145*	*0.0317*	*0.0178*	*0.0170*	*0.0230*	*0.0113*	*0.0087*		
SFAN-SB	0.098	0.151	0.235	0.124	0.010	0.097	0.289	0.280	0.170	0.087	0.119	**7.95**	
	0.0072	*0.0205*	*0.0790*	*0.0467*	*0.0143*	*0.0315*	*0.0178*	*0.0177*	*0.0228*	*0.0115*	*0.0086*		
SFAN-LCV_LO	0.104	0.144	0.247	0.117	0.015	0.081	0.306	0.277	0.161	0.088	0.117	**8.45**	
	0.0102	*0.0286*	*0.0862*	*0.0545*	*0.0225*	*0.0188*	*0.0435*	*0.0306*	*0.0130*	*0.0167*	*0.0103*		
SFAND-SB	0.098	0.151	0.235	0.125	0.010	0.096	0.289	0.280	0.170	0.087	0.119	**8.05**	
	0.0072	*0.0205*	*0.0790*	*0.0474*	*0.0143*	*0.0312*	*0.0178*	*0.0177*	*0.0228*	*0.0115*	*0.0086*		
SFAND-LCV_LO	0.104	0.144	0.247	0.117	0.015	0.081	0.306	0.277	0.161	0.088	0.117	**8.50**	
	0.0102	*0.0286*	*0.0862*	*0.0545*	*0.0225*	*0.0188*	*0.0435*	*0.0306*	*0.0130*	*0.0167*	*0.0103*		
STAN-SB	0.092	0.127	0.238	0.064	0.010	0.093	0.282	0.222	0.136	0.071	0.108	*13.36*	
	0.0067	*0.0243*	*0.0869*	*0.0353*	*0.0204*	*0.0282*	*0.0259*	*0.0270*	*0.0153*	*0.0104*	*0.0104*		
STAN-LCV_LO	0.101	0.153	0.231	0.099	0.005	0.092	0.301	0.271	0.157	0.080	0.115	*11.45*	
	0.0076	*0.0267*	*0.0838*	*0.0390*	*0.0045*	*0.0207*	*0.0326*	*0.0323*	*0.0192*	*0.0108*	*0.0073*		
STAND-SB	0.098	0.151	0.235	0.125	0.010	0.097	0.289	0.280	0.170	0.087	0.119	**7.50**	
	0.0072	*0.0205*	*0.0790*	*0.0474*	*0.0145*	*0.0315*	*0.0178*	*0.0177*	*0.0228*	*0.0115*	*0.0086*		
STAND-LCV_LO	0.104	0.144	0.247	0.118	0.010	0.081	0.305	0.277	0.161	0.088	0.117	9.32	
	0.0102	*0.0286*	*0.0862*	*0.0536*	*0.0120*	*0.0188*	*0.0406*	*0.0306*	*0.0130*	*0.0167*	*0.0103*		
K2	0.093	0.150	0.264	0.125	0.013	0.084	0.275	0.308	0.158	0.072	0.104	9.82	
	0.0090	*0.0320*	*0.0832*	*0.0590*	*0.0188*	*0.0346*	*0.0314*	*0.0145*	*0.0139*	*0.0105*	*0.0057*		
MMHC15	0.090	0.086	0.049	0.081	0.000	0.066	0.270	0.217	0.093	0.057	0.097	*16.45*	
	0.0100	*0.0441*	*0.0807*	*0.0515*	*0.0003*	*0.0119*	*0.0449*	*0.0275*	*0.0546*	*0.0267*	*0.0259*		
Log. Regr.	0.111	0.180	0.114	0.118	0.023	0.112	0.282	0.269	0.180	0.093	0.127	**5.36**	*With* Markov Blanket feature selection at 15%
using continuous data	*0.0121*	*0.0425*	*0.1300*	*0.0604*	*0.0490*	*0.0294*	*0.0689*	*0.0238*	*0.0123*	*0.0110*	*0.0091*		
RndFor	0.133	0.181	0.397	0.220	0.016	0.113	0.304	0.380	0.220	0.070	0.147	**2.55**	
using continuous data	*0.0102*	*0.0439*	*0.0879*	*0.0577*	*0.0386*	*0.0450*	*0.0840*	*0.0377*	*0.0173*	*0.0112*	*0.0101*		
Naive Bayes	0.093	0.160	0.236	0.087	0.007	0.084	0.247	0.160	0.167	0.090	0.131	**8.64**	
	0.0088	*0.0354*	*0.0730*	*0.0427*	*0.0056*	*0.0162*	*0.0321*	*0.0123*	*0.0161*	*0.0105*	*0.0067*		
Naive Bayes kernel	0.087	0.162	0.029	0.103	0.029	0.106	0.167	0.217	0.170	0.080	0.121	*10.36*	
using continuous data	*0.0090*	*0.0304*	*0.0207*	*0.0576*	*0.0397*	*0.0318*	*0.0449*	*0.0248*	*0.0165*	*0.0117*	*0.0050*		
TAN	0.097	0.164	0.235	0.096	0.006	0.079	0.290	0.277	0.172	0.089	0.122	**7.05**	
	0.0098	*0.0289*	*0.0875*	*0.0431*	*0.0048*	*0.0213*	*0.0317*	*0.0202*	*0.0210*	*0.0134*	*0.0100*		
FAN-SB	0.097	0.164	0.235	0.096	0.006	0.079	0.290	0.277	0.172	0.089	0.122	**6.95**	
	0.0098	*0.0289*	*0.0875*	*0.0431*	*0.0048*	*0.0213*	*0.0318*	*0.0202*	*0.0210*	*0.0134*	*0.0100*		
FAN-LCV_LO	0.097	0.164	0.235	0.096	0.006	0.079	0.291	0.277	0.172	0.089	0.122	**6.77**	
	0.0098	*0.0289*	*0.0875*	*0.0431*	*0.0048*	*0.0213*	*0.0312*	*0.0202*	*0.0210*	*0.0134*	*0.0100*		
SFAN-SB	0.097	0.164	0.231	0.096	0.006	0.079	0.290	0.278	0.172	0.089	0.122	**6.95**	
	0.0098	*0.0290*	*0.0929*	*0.0435*	*0.0048*	*0.0213*	*0.0318*	*0.0200*	*0.0209*	*0.0136*	*0.0099*		
SFAN-LCV_LO	0.104	0.149	0.218	0.090	0.006	0.081	0.289	0.277	0.159	0.085	0.118	9.82	
	0.0104	*0.0337*	*0.0806*	*0.0425*	*0.0049*	*0.0208*	*0.0317*	*0.0302*	*0.0162*	*0.0154*	*0.0106*		
SFAND-SB	0.097	0.164	0.231	0.096	0.006	0.079	0.290	0.278	0.172	0.089	0.122	**6.95**	
	0.0098	*0.0290*	*0.0929*	*0.0435*	*0.0048*	*0.0213*	*0.0318*	*0.0200*	*0.0209*	*0.0136*	*0.0099*		
SFAND-LCV_LO	0.104	0.149	0.218	0.090	0.006	0.081	0.289	0.277	0.159	0.085	0.118	9.91	
	0.0104	*0.0337*	*0.0806*	*0.0425*	*0.0049*	*0.0208*	*0.0317*	*0.0302*	*0.0162*	*0.0154*	*0.0106*		
STAN-SB	0.092	0.128	0.201	0.086	0.004	0.070	0.282	0.222	0.138	0.072	0.109	*15.27*	
	0.0077	*0.0251*	*0.0956*	*0.0478*	*0.0054*	*0.0157*	*0.0336*	*0.0274*	*0.0168*	*0.0134*	*0.0100*		
STAN-LCV_LO	0.101	0.149	0.221	0.095	0.006	0.080	0.287	0.273	0.154	0.084	0.117	*11.73*	
	0.0082	*0.0370*	*0.0645*	*0.0430*	*0.0048*	*0.0192*	*0.0297*	*0.0306*	*0.0202*	*0.0142*	*0.0069*		
STAND-SB	0.097	0.164	0.231	0.096	0.006	0.079	0.290	0.278	0.172	0.089	0.122	**7.05**	
	0.0098	*0.0290*	*0.0929*	*0.0435*	*0.0048*	*0.0213*	*0.0317*	*0.0200*	*0.0209*	*0.0136*	*0.0099*		
STAND-LCV_LO	0.104	0.149	0.218	0.088	0.006	0.081	0.289	0.277	0.159	0.085	0.118	*10.27*	
	0.0104	*0.0337*	*0.0806*	*0.0445*	*0.0049*	*0.0208*	*0.0320*	*0.0302*	*0.0163*	*0.0154*	*0.0106*		
K2	0.093	0.160	0.225	0.093	0.006	0.074	0.288	0.308	0.158	0.077	0.108	*11.09*	
	0.0101	*0.0321*	*0.0726*	*0.0445*	*0.0051*	*0.0173*	*0.0293*	*0.0143*	*0.0145*	*0.0109*	*0.0107*		
MMHC15	0.090	0.100	0.189	0.087	0.005	0.070	0.267	0.206	0.118	0.063	0.105	*16.27*	
	0.0100	*0.0252*	*0.0984*	*0.0355*	*0.0051*	*0.0132*	*0.0206*	*0.0266*	*0.0373*	*0.0211*	*0.0090*		

of 1.025×10^{-4} in case of AUC and smaller than 10^{-10} in case of the H-measure). Using the post-hoc Bonferroni-Dunn test to compare the results without input selection with those obtained by performing the MB feature selection procedure prior to model construction, it was found that MB.15 did not result in significantly lower performance; MB.05 did however result in significantly worse performing models. Hence, the results of MB.05 are omitted in the remainder of this discussion.

Software fault prediction techniques

All software fault prediction techniques are compared by first applying a Friedman test, followed by a post-hoc Nemenyi test, as explained in Section 4.4.4. The Friedman test resulted in a p-value smaller than 10^{-10} for both AUC and the H-measure. The null hypothesis of equal performance amongst all techniques is thus strongly rejected and in a next step, the post-hoc Nemenyi test assessing all pairwise differences between techniques is performed. The outcome of this test is given in Fig. 4.6. The horizontal axis in these figures corresponds to the average rank (AR) of a technique across all data sets. The techniques are represented by a horizontal line; the more this line is situated to the left, the better performing a technique is. The left end of this line depicts the AR while the length of the line corresponds to the critical distance for a difference between any two techniques to be significant at the 1% significance level. In case of 17 techniques and 11 data sets, this critical distance equals 5.959. The first set of dotted and full vertical lines in the figure indicates the critical difference at respectively the 5% and 1% significance level with the

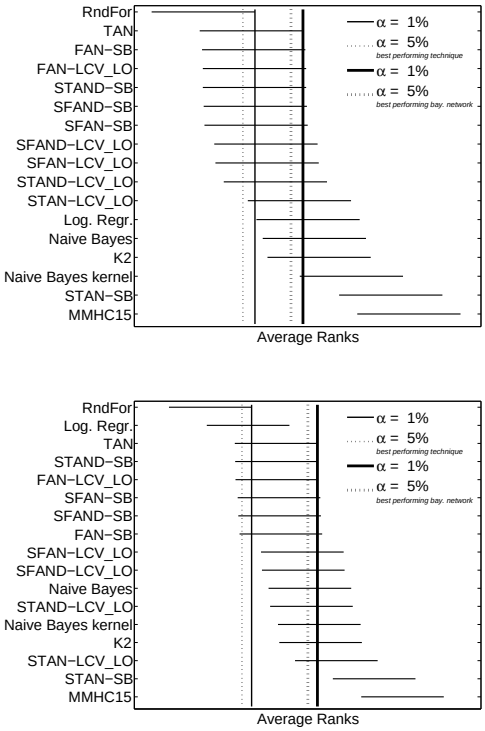


Figure 4.6: Ranking of software fault prediction models for the AUC and H-measure with $\beta(2, 2)$ using the post-hoc Nemenyi test in the top and bottom panel respectively

overall best performing technique. The second set of vertical lines, displayed in bold, represents the differences with the best performing Bayesian Network learner. A technique is significantly outperformed if located at the right side of the vertical line.

Recently, an alternative to the AUC as a measure of aggregated classification performance was proposed, allowing to specify a probability distribution over the misclassification losses: the H-measure. The results of both metrics exhibit a similar pattern as random forest (RndFor) is found to be the overall best performing technique, both in terms of the AUC and H-measure, confirming a.o. the work of Lessmann et al. [200] and Guo et al. [118]. Furthermore, one can observe a similar ranking across techniques, indicating the same techniques as worst performing. One notable exception is logistic regression (Log. Reg.); in terms of AUC, this technique is found to be outperformed by RndFor at the 1% significance level while for the H-measure, Log. Reg. ranks second. This can partially be explained by the leveling out effect observed for AUC; i.e. several Augmented Naive Bayesian learners perform similarly in terms of AUC and are thus attributed a similar ranking. Log. Reg., which performs slightly worse in terms of AUC than these learners is thus ranked much lower. The fact that the rankings are similar when no additional information on misclassification costs is included in the H-measure is interesting.

Interestingly, when considering the AUC metric, most of the BN learners are not significantly outperformed at the 1% significance level by RndFor, see Fig. 4.6, top panel. However, unlike the conclusions of Lessmann et al. [200], who only considered the AUC, it is found that the Naive Bayes learner, which is often used in software fault prediction research, is outperformed at the 1% significance level. Similar results can be found when focussing on the H-measure; giving more discriminative results, the Naive Bayes learner as well as a number of augmented Naive Bayes classifiers are found to be significantly outperformed at the 1% level. As such, other BN learners which provide a more informative network structure can indeed be regarded as a valid alternative to Naive Bayes. Considering BN learners only, Tree Augmented Naive Bayes (TAN) was found to be the best performing classifier, while STAN-SB and MMHC15 are found to perform significantly worse than TAN at the 1% level, both in case of the AUC and the H-measure. Especially the fact that MMHC15 scores last is noteworthy, as this BN learner allows to construct any possible DAG as network structure. This can be explained by the fact that MMHC15 uses conditional independence tests to determine the network structure; even small amounts of noise in the data set can lead to incorrect conclusions reached by such tests [277].

As explained in Section 4.4.3, the H-measure relies on a beta distribution characterized by two parameters which determine the likelihood of different cost ratios. It can be argued that this cost ratio is in fact context specific, and distribution parameters reflecting different cost ratios should be considered [153]. Parameter settings reflecting a different development context have thus been adopted, investigating the robustness of the H-measure in the context of software fault prediction. The outcome is presented in Fig. 4.7. The horizontal axis of this figure represents the expected value of the cost ratio while the vertical

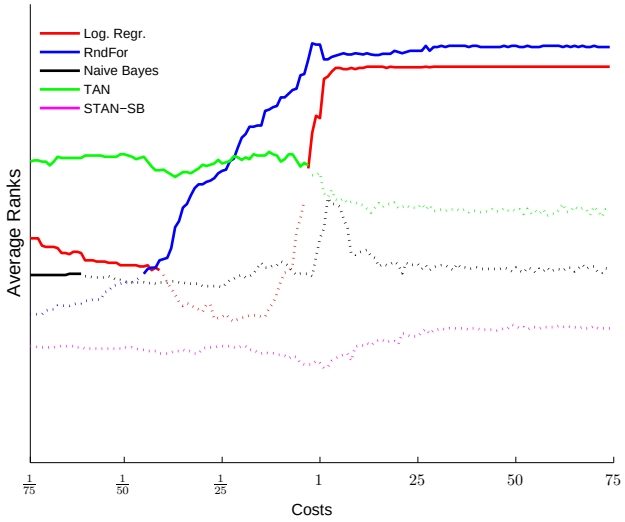


Figure 4.7: Robustness of the H-measure

axis corresponds to the average ranks (AR). Techniques are represented by a line; if a technique does not perform statistically worse than the best performing technique at the 5% significance level, a full line is used and a dotted line otherwise. Bonferroni-Dunn tests are used in assessing the techniques at each cost ratio. Note that to improve readability, only a selection of techniques is shown, combining the best techniques both from a comprehensibility and performance point of view. It can be seen that RndFor remains the overall best performing technique when the cost ratio is larger than one, which corresponds to a risk averse development context. When considering a delay averse context however, very different conclusions can be reached. In such a context, Augmented Bayesian Network classifiers are found to be best performing. A cost ratio of one seems to be pivotal in this respect. One possible explanation to these findings lies in the fact that BN learners are known to be biased, exhibiting a tendency towards overconfidence in their predictions [128]. This reaffirms earlier conclusions concerning the importance of taking development context into account in software fault prediction [153, 154].

Markov blanket feature selection

It is known that several static code features are correlated and e.g. principal component analysis or factor analysis has previously been used to reduce the number of features [169, 202, 315]. A possible downside of such approach is a decrease in comprehensibility as several static code features are aggregated into a single feature. An alternative explored by e.g. Menzies et al. in the context of the NASA data sets is the use of a filter approach to select the most informative subset of features prior to model construction [230]. Catal et al. also considered a filter approach and compared it to directly discarding aggregated features such

as derived Halstead measures [53]. Both confirmed the possibility of selecting a set of most informative features from the data without incurring a performance penalty. In the first study, the authors were able to build fault prediction models based on three features, while the filter employed in the more recent study of Catal et al. selected between three and eight metrics prior to model construction. The MB feature selection procedure of this study can be regarded as an example of a filter approach. It was in some cases able to select as little as five attributes; however, on some data sets the MB included up to 23 features. The MB.05 filter effectively further reduced the number of selected features, but resulted in lowered performance. Menzies et al. reported Halstead and LOC based metrics to be the most often selected features[11]. Fig. 4.8 reports our findings hereon; the bar chart depicts the average number of attributes selected by the MB.15 procedure per data set and per group of static code features. It can be seen that in case of the NASA data sets, Halstead and LOC based metrics are most often selected by the MB.15 filter. A notable exception is the PC5 data set, for which McCabe complexity metrics were found to be the second most important group. Remark that this last data set was not included in their study. The Eclipse data sets, containing an alternative set of static code features, provide another picture as method level attributes are prevalent. In all three Eclipse data sets, metrics collected at different granularity were selected. Investigation at the level of individual attributes reveals significant differences in selected features between data sets. This supports the findings of Menzies et al. who concluded that 'The best attributes to use for defect prediction vary from data set to data set'. Finally, it can be argued that depending on

[11]Note that our selection of data sets is not identical to Menzies' study and that minor differences exist in the grouping of static code features, see Table 4.4. E.g. 'Percent_comments' was regarded as a LOC based static code feature, in line with the documentation of the NASA MDP.

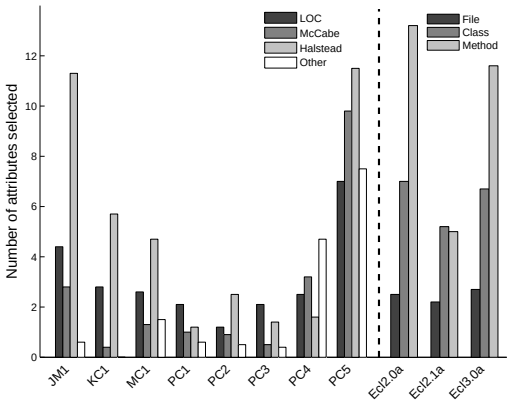


Figure 4.8: Bar chart of the average number of selected attributes per data set and per attribute group

the data, other feature selection techniques seem more effective than the MB procedure. This can in part be explained by the requirement to discretize the data when considering BN learners. Note that RndFor and some of the BN learners explored in this study also include embedded feature selection, the impact of which is further discussed in the next section.

4.5.2 Comprehensibility of the Bayesian networks

Several characteristics constitute a good software fault prediction model, of which performance is only one element. Model comprehensibility is also important, especially if such a model would be deployed in a real life setting [95]. As argued by e.g. Kotsiantis [189], BN classifiers are amongst the most comprehensible classifiers, but their comprehensibility can be hampered by the complexity of the network structure. Fig. 4.9 reports on this aspect by plotting the number of nodes, arcs and the network dimensionality of each BN learner, both with and without prior application of the MB feature selection procedure. Techniques are ordered according to their classification performance using the H-measure; a technique situated above the dotted line was not found to be significantly outperformed at the 5% level by RndFor, the best performing learner.

The graphs illustrate the impact of MB feature selection on network complexity by reducing the number of nodes and arcs in the network and lowering the number of parameters to be estimated, or network dimension. Similar to the performance assessment, a Friedman test is first carried out to establish whether differences observed in the net-

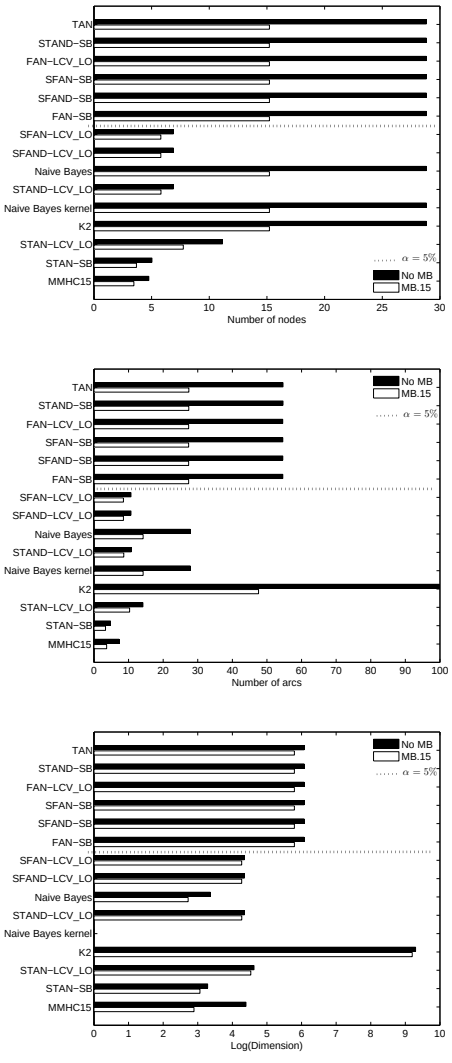


Figure 4.9: Comparison of Bayesian networks: comprehensibility

work dimension are significant. As the null hypothesis of no significant differences is strongly rejected with a p-value smaller than 10^{-10}, a Bonferroni-Dunn test is performed comparing the best BN learner to all others. The results are depicted in Fig. 4.10 and indicate that Selective Tree Augmented Naive Bayes using the Standard Bayesian quality measure, STAN-SB, is the BN learner associated with the lowest network dimension. As such, one can argue that models induced by this learner are the simplest and most comprehensible. Naive Bayes, MMHC15 and several Augmented Naive Bayes learners using the Local Leave-One-out Cross Validation quality measure to assess network fitness are found to be not significantly more complex. Taking however into account the fact that STAN-SB is outperformed by the best performing Bayesian learner, TAN, it can be argued that the use of a local quality measure is arguably better than using the Standard Bayesian quality measure as it results in similar performance to TAN while resulting in networks which are not significantly more complex than STAN-SB. More general, one can observe that the Augmented Naive Bayes classifiers, which provide a relaxation to the TAN assumption, are able to reduce the number of nodes and arcs compared to TAN, without a loss in predictive power.

General Bayesian Networks which are able to adopt any DAG as network structure, are found to be less appealing. MMHC15, which performed significantly worse than other BN learners, typically constructs very simple networks. These networks are often even overly simple, containing only a very limited set of features (nodes). The other General Bayesian Network learner, K2, performed much better but at the expense of very complex network structures[12].

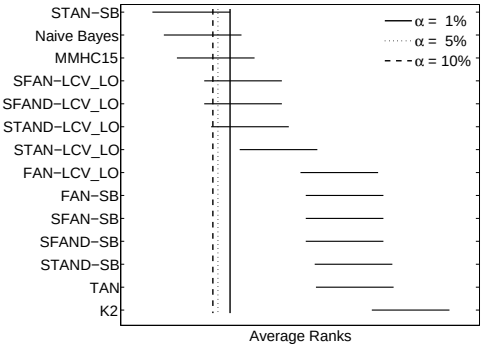


Figure 4.10: Ranking of software fault prediction models for the network dimension using the Bonferroni-Dunn test

When selecting the optimal learner to construct software fault prediction models, a tradeoff is typically made between model comprehensibility and classification performance. It should be noted that the techniques found to result in the most comprehensible models are also found to be outperformed by random forest. Hence, it can be argued that when gaining insight into what drives software faults is of key importance, BN classifiers offer considerable advantages to other, more opaque models. More specifically, the NB learner as well as several

[12]The K2 algorithm allows to limit the number of parents for each node but as the objective was to test this algorithm as a GBN, this restriction was not imposed.

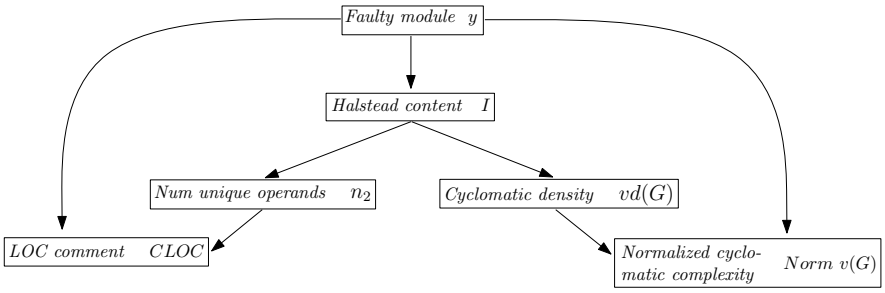


Figure 4.11: Bayesian network learned by the Augmented Naive Bayes classifier 'STAND LCV_LO' without MB feature selection on the PC1 data set

Augmented Bayesian Learners using the LOO-CV quality criterion during network construction are to be recommended. An important note in this respect is that Naive Bayes is typically easy to implement and can be written as a sum of logs to obtain a linear model [128, 315]. However, this learner is unable to discard uninformative attributes which can prove important in gaining further insight into fault prediction. On the other hand, when classification performance is crucial, other techniques such as random forest would seem more appropriate. As discussed in the previous section however, the question of which technique induces the most appropriate model depends on the development context.

As an example, Fig. 4.11 shows the network for the PC1 data set learned by the STAND LCV_LO classifier (without prior input selection), a technique not found to be outperformed by the best Bayesian learner while typically not resulting in significantly more complex networks than STAN-SB. As one can observe, the algorithm was able to make accurate predictions retaining only five features. When interpreting the network, it is important to realize that the existence of an arc not necessarily implies causality, but rather should be seen as (conditional) dependence between the variables. In this network, the presence of software faults is directly governed by CLOC (number of commentary lines), I (Halstead content) and the normalized cyclomatic complexity. Further correlations between e.g. I and the number of unique operands can be discerned, which is plausible when considering that the latter serves as input to calculate the first. The relations present in the network can be helpful when for instance issuing guidelines on software complexity to programmers.

4.6 Conclusion

Time and cost effective software development are decisive for today's developers and since the pioneering work from the 70s, several avenues to tackle problems related hereto have been investigated. Software fault prediction can be regarded as one piece of the solution to these issues. It is argued by Lessmann et al. that fault prediction techniques should not be judged on their predictive performance alone, but that other aspects such as computational efficiency, ease of use, and

especially comprehensibility should also be paid attention to [200].

This chapter tries to answer this call by comparing 15 Bayesian network learners, both in terms of the Area Under the ROC Curve (AUC) and the recently introduced H-measure. The results of the experiments show that Augmented Naive Bayes classifiers can yield similar or better performance than the commonly used Naive Bayes classifier. This additional performance however comes at the expense of more complex models. Considering comprehensible models only, Augmented Naive Bayes classifiers using the Local Leave-One-out Cross Validation quality measure are to be recommended. The Naive Bayes classifier, which can be turned into a linear model is also a valid alternative, despite its simple network structure. General Bayesian Networks were found to be either outperformed by other Bayesian learners, or to result in overly complex network structures. It can be argued that networks which focus on a smaller set of highly predictive features provide practitioners the means to gain insights more easily into the drivers of software faults and to further capitalize hereon, the use of Markov blanket (MB) feature selection was also tested. The outcome indicates that MB is able to reduce the number of variables while not negatively impacting performance. However, other feature selection approaches are possibly able to select an even smaller set of highly predictive features.

Depending on the development context, and the associated costs of misclassifying a (non) faulty instance, other more opaque models are found to be more discriminative. Our findings support earlier results identifying the random forest learner as the most appropriate to model the presence of faults if the cost of not detecting them outweighs the additional testing effort. In the opposite situation, Augmented Bayesian Network classifiers are found to be the better choice. The question how other techniques such as support vector machines or neural networks perform under these circumstances remains to be explored.

Recently, several researchers turned their attention to another topic of interest; i.e. the inclusion of information other than static code features into fault prediction models such as information on inter module relations [316] and requirement metrics [155]. The relation to the more commonly used static code features remains however unclear. Using e.g. Bayesian network learners, important insights into these different information sources could be gained which is left as a topic for future research.

Someone told me that each equation I included in the book would halve the sales.

Stephen Hawking, 1988

5

Revisiting earlier results: the NASA MDP case

Empirical software engineering has greatly benefited from the sharing of data through public repositories such as the Promise website and the NASA Metrics Data Program (MDP). This dependency on public data offers both great opportunities and poses serious threats; taking public data sources for granted, one may be encouraged to formulate and test hypotheses on ill understood data. As signposted earlier, data collection and preprocessing are however not to be regarded as subordinate steps in the KDD process, but in fact should form the cornerstone of any meaningful analysis. This chapter explores a recent critique of Gray et al. on the validity of the NASA MDP data, revisiting the study of Lessmann et al., while also proposing an update to their well-established benchmarking framework. We verify that different data set cleaning procedures have significant impacts upon experimental results, but believe this should not overshadow previous findings in the literature. In order to develop our understanding of, and confidence in, competing defect prediction systems, experimenters are encouraged to be more systematic and transparent in their data set preprocessing.

This chapter is based on the following manuscript

- Karel Dejaeger, Stefan Lessmann, Martin Shepperd and Bart Baesens, "Benchmarking Classification Models for Software Defect Prediction: Revisiting Earlier Results," *IEEE Transactions on Software Engineering*, In submission.

5.1 Introduction

As mentioned in the previous chapter, developing high-quality software systems is a complex and usually very expensive process in which software verification and validation is a critical step, and in spite of the attention it received over the years, it remains a largely human-centric process that relies on observing a sample of executions. This is even more true with object oriented software, in which concepts like encapsulation and inheritance tend to introduce new risks and increase the need and complexity of testing [33].

Typically, defect prediction models are used to estimate the likelihood of

code segments or modules containing defects based on a set of static code characteristics or change data and been successfully put to the test in many software companies. Evidence on this can be found in the many industry case studies [251, 310]. For instance, a case study by Khoshgoftaar et al. [173] in a large legacy telecommunication system indicated a strong positive return on investment when adopting defect prediction models compared to a random selection of test candidates. A similar conclusion was reached by Arisholm et al. [12] who investigated a middleware system of a large telco operator, reporting that using a prediction model to focus verification efforts could result in a cost saving of about 29%.

Chapter 1 already elaborated upon the research activity instilled by the potential value of accurate defect prediction. To facilitate this line of research, a number of public data repositories have been created containing data from various software development projects; popular examples include the NASA Metrics Data Program (MDP) repository and the Promise repository [288]. Having public data repositories enhances the current state of software engineering research by offering the possibility to reproduce studies, refining or validating earlier findings [232]. For a detailed evaluation of current research see the two recent systematic reviews by Catal et al. [54] and Hall et al. [121].

It is important to appreciate that the application of machine learning algorithms is only one particular step in the knowledge discovery process presented in Section 2.4 and that data quality and data cleaning or preprocessing also plays a crucial role [72]. Poor data quality can have a detrimental effect on the findings of a study and in light of this observation, the availability of public data repositories both pose great opportunities and threats to researchers. The widespread use of the NASA data sets is striking. For instance, Catal et al. [54] found that public data sets mostly originate from Promise and NASA MDP repositories, while Hall et al. [121], from a total of 208 defect prediction studies, identified 58 primary studies that used NASA data sets, or more than 25%.

Recently, doubts have been cast on the quality of these data sets by Gray et al. [117], leading them to conclude that the "bulk of defect prediction experiments based on the NASA Metrics Data Program data sets may have led to erroneous findings". Responding to their call to repeat studies that might be affected by these data quality issues, we reevaluate the findings of Lessmann et al. [200]. We choose this study for four reasons. One, it covers an extensive (22) set of different learning algorithms. Two, it is based upon a rigorous statistical methodology using a Friedman repeated measures design coupled with Nemenyi *post-hoc* testing as recommended by Demšar [77]. Three, it is published in the community's flagship journal. Lastly, repeatability is facilitated by virtue of the assistance of two authors of the original study.

A second, and related, theme that arises is the finding that the performance of many machine learning algorithms are statistically indistinguishable. This has inspired new research directions, e.g., towards optimization of defect prediction models on goals other than classification performance [231].

Therefore our study seeks to answer two questions. First, what is the impact

of the reported data quality problems on previously published defect prediction studies that have been based, at least in part, upon the NASA defect data sets? Second, can we identify statistical procedures that better distinguish between the prediction performance of these competing defect classification models?

The remainder of this chapter is structured as follows.
Section 5.2 outlines the data quality issues present in the NASA data sets and details the preprocessing procedure for addressing these issues.
Section 5.3 explains the empirical setup and the updated benchmarking framework.
Section 5.4 discusses the findings of this study.
Section 5.5 summarizes our findings and offers a discussion on their implications.

5.2 Data quality in the NASA data sets

As indicated, the motivation for this study lies in the recent work of Gray et al., which signaled several data quality issues with the NASA data sets which could potentially invalidate earlier findings [117]. As these data sets have been frequently used for defect prediction, the importance of thoroughly investigating these issues and quantifying their impact cannot be overstated. Table 5.1 provides an overview of the usage of the NASA data sets; for brevity, only studies published in journals identified as most influential to defect prediction research are included.

From this overview, it is clear that multiple versions of the same data sets exist; this is true for the CM1, JM1, KC1 and PC1 data sets. Moreover, the JM1 data set hosted on the Promise repository[1] contains 5 observations with missing values on 5 attributes. Menzies et al. [230] seemed to have used yet another version of these data sets, containing one additional observation compared to other studies. Remarkably, the majority of defect prediction studies did not consider these differences, directly using data obtained from either the MDP or Promise repository. Even more surprising, there are also differences in the collection of data sets offered by both repositories; KC2 is not hosted on the NASA MDP repository while, until 16 November 2011, the KC4 data set was unavailable on the Promise repository. While it should be noted that the NASA MDP repository was recently taken off-line, parallel versions of the same data sets are still in circulation[2].

Gray et al. [117] outlined a second problematic issue, namely the large number of repeated data points, resulting in the repetition of training observations in test and validation sets. It has been argued that when tuning a learner towards the specific characteristics of a data set using a validation set containing duplicates, learners are prone to over-fitting [187]. Duplicates in the test set on the other hand can result in overly optimistic performance estimates [334].

[1] Note the Promise repository hosts a version of the NASA data sets, http://promisedata.org.
[2] See for example http://nasa-softwaredefectdatasets.wikispaces.com.

Furthermore, the validity of specific data values such as non-integer values for counts such as LOC was also questioned, a problem present in the CM1, JM1, KC1 and PC1 data sets hosted on the Promise repository. Additionally, Gray et al. indicated issues pertaining to missing values, inconsistent observations (i.e. observations with an identical attribute vector, but different class labels), and data integrity. The latter issue can be tackled by imposing domain specific constraints on the data, taking into account the correlated nature of certain attributes [217]. Also the logical correctness can be checked; e.g. modules with a total line count of zero can be deemed suspicious; e.g., Lessmann et al. [200] and Dejaeger et al. [76] discarded observations with a total line count of zero.

Depending on the exact learning algorithm under investigation, execution of one or more preprocessing steps is a *sine qua non* to enable meaningful analysis. For instance, including a constant feature can break techniques such as logistic regression which are based on first or second order derivatives. Other techniques, e.g., decision tree algorithms, will simply ignore uninformative attributes. In order to assess the issues outlined by Gray et al., the following preprocessing steps are initially applied to each data set. Each observation (software module) in the data sets consists of a unique ID, several static code features and an error count. First, the data used to learn and validate the models are selected and thus, the *ID* and *error density* as well as attributes exhibiting zero variance are discarded. Next, the *error count* is discretized into a boolean value where 0 indicates that no errors were recorded for this software module and 1 otherwise, in line with e.g. [200,230,315,317,320]. Finally, it should be specified how to deal with missing values; the next paragraph offers a commonplace solution, resulting in the MDP$^{\circ}$ and Promise$^{\circ}$ data sets. When investigating the claims of Gray et al., an alternative preprocessing schema is adopted in this study, yielding MDP$'$, MDP* and MDP$''$. An overview of all data sets is shown in Table 5.2, while Procedure *Initial preprocessing* summarizes the initial preprocessing steps, and Procedure *Further preprocessing* sums up the base preprocessing and alternative preprocessing schema in the top and bottom panel respectively.

5.2.1 Basic preprocessing schema

This study quantifies, amongst other objectives, the differences between data sets derived from the Promise and the MDP repository, and thus only data sets originally available in both repositories have been selected. Subsequent to the initial preprocessing steps outlined above, a missing value handling procedure is performed since some techniques are unable to cope with missing values (e.g. logistic regression). Note that seven data sets contain a large number of missing values on the same attribute, *decision density*. To retain as much data as possible, the attribute associated with the missing values is removed from the data set if more than 10% of its values are missing. Otherwise, the observations associated with the missing values are discarded.

5.2.2 Alternative preprocessing schema

In response to Gray et al., an alternative data preprocessing schema was proposed by Shepperd et al. [285][3] who proposed the removal of problem data (e.g. logically incorrect observations) as a first step, and the subsequent removal of duplicate or inconsistent observations, resulting in DS′ and DS″ respectively.

The NASA data was mined using the McCabe IQ suite, and contains varied quantities of correlated attributes, which can be used to verify data integrity. An example can be found e.g. in the Halstead measures, which are formulaic expressions of operator and operand counts. Observations containing attributes violating referential constraints are deemed logically incorrect and are removed (lines 3-5).

Note that observations with a total line count of zero are retained, since Gray et al. argue that empty modules can constitute a valid part of the system, and advise against removal. Subsequent to removing observations violating referential constraints, lines 6-8 discard observations with missing values, as suggested by Shepperd et al. [285], yielding the DS′ data sets.

The DS′ data sets can be further processed by removing duplicates and inconsistent observations; Gray et al. point out that the presence of duplicates can perhaps reflect real life situations, but is problematic in the context of evaluating the future generalisability of prediction models. Inconsistent observations on the other hand reflect the situation of multiple modules with similar length and complexity, some being faulty and others not. Learners will be unable to discriminate between both, resulting in an upper bound on the generalizing ability of defect models. DS⋆ features only duplicate removal, see lines 9-12, while in DS″ also inconsistent observations are discarded (lines 13-17).

Procedure Initial preprocessing

Initial preprocessing steps

inputs : MDP - NASA MDP data
Promise - Promise data

```
// DS represents one specific data set
1  for each DS ∈ MDP ∪ Promise do
2      Remove attributes ID and error density
3      for j = 1 to n do                    // Removing constant attributes
4          if variance (DS.x_(j)) = 0 then
5              DS.x_(j) = []
6      for i = 1 to N do                    // Discretizing error count
7          if DS.y_i > 0 then
8              DS.y_i = 1
```

output: DS

[3] http://nasa-softwaredefectdatasets.wikispaces.com

Procedure Further preprocessing

Basic preprocessing schema

inputs : DS - Data after initial preprocessing

```
// DS represents one specific data set
for each DS do
    for j = 1 to n do                                  // Handling missing values
        if count (missing (DS.x_(j))) > N×0.1 then
            DS.x_(j) = []
        else
            (DS.x_i, DS.y_i) = []
    output: DS°
```

Alternative preprocessing schema

```
Select MDP data sets
for each DS do
    for i = 1 to N do                  // Removing logically incorrect obs.
        if incorrect (DS.x_i) then
            (DS.x_i, DS.y_i) = []
    for i = 1 to N do                  // Removing obs.  with missing values
        if count (missing (DS.x_i))> 0 then
            (DS.x_i, DS.y_i) = []
        output: DS′
    for i = 1 to N do                                  // Removing duplicate obs.
        for k = i+1 to N do
            if (DS.x_i,DS.y_i) = (DS.x_k,DS.y_k) then
                (DS.x_k, DS.y_k) = []
        output: DS⋆
    for i = 1 to N do                                  // Removing inconsistent obs.
        for k = i+1 to N do
            if DS.x_i = DS.x_k then
                (DS.x_i, DS.y_i) = []
                (DS.x_k, DS.y_k) = []
        output: DS″
```

Table 5.1: Overview of previous usage of NASA data sets

Title	**Year**	**Data set** & *Observation count*						**Notes**
IEEE Transactions on Software Engineering								
Towards comprehensible software fault prediction models using Bayesian network classifiers [76]	2012	JM1 *10,878* KC1 *2,107* MC1 *4,625*	PC1 *1,059* PC2 *4,505* PC3 *1,511*		PC4 *1,347* PC5 *15,414*			MDP repository. Removed observations with a total line count of zero.
A general software defect-proneness prediction framework [293]	2011	CM1 *505* JM1 *10,878* KC1 *2,107* KC3 *458* KC4 *125*	MC1 *9,466* MC2 *161* MW1 *403* PC1 *1,107*		PC2 *5,589* PC3 *1,563* PC4 *1,458* PC5 *17,186*			MDP repository.
Evolutionary optimization of software quality modeling with multiple repositories [209]	2010	CM1 *505* JM1 *8,850* KC1 *2,107*	KC2 *520* KC3 *458*		MW1 *403* PC1 *1,107*			MDP repository. Removed redundant, obvious noise and observations with missing values for JM1.
Benchmarking classification models for software defect prediction: A proposed framework and novel findings [200]	2008	CM1 *505* JM1 *9,537* KC1 *1,571* KC3 *458*	KC4 *125* MW1 *403* PC1 *1,059*		PC2 *4,505* PC3 *1,511* PC4 *1,347*			MDP repository. Removed observations with a total line count of zero.
Empirical analysis of software fault content and fault proneness using Bayesian methods [252]	2007	KC1 *145*						MDP repository. KC1 refers to class level observations.
Data mining static code attributes to learn defect predictors [230]	2007	CM1 *506* KC3 *459* KC4 *126*	MW1 *404* PC1 *1,108* PC2 *5,590*		PC3 *1,564* PC4 *1,458*			Promise repository.
Empirical analysis of object-oriented design metrics for predicting high and low severity faults [341]	2006	KC1 *145*						MDP repository. KC1 refers to class level observations.

Journal of Systems and Software								
A symbolic fault-prediction model based on multiobjective particle swarm optimization [70]	2010	CM1 JM1	*498* *10,885*	KC1 KC2	*2,109* *522*	PC1	*1,109*	MDP repository.
Applying machine learning to software fault-proneness prediction [113]	2008	JM1	*10,878*					MDP repository.
Mining software repositories for comprehensible software fault prediction models [320]	2008	KC1	*1,571*	PC1	*1,059*	PC4	*1,347*	MDP repository.
Predicting defect-prone software modules using support vector machines [88]	2008	CM1 KC1	*496* *2,107*	KC3	*458*	PC1	*1,107*	MDP repository.
Empirical Software Engineering								
On the relative value of cross-company and within-company data for defect prediction [317]	2009	CM1 KC1 KC2	*498* *845* *522*	KC3 MC2	*458* *61*	MW1 PC1	*403* *1,109*	Promise repository.
Techniques for evaluating fault prediction models [154]	2008	CM1 JM1 KC1	*498* *10,885* *2,109*	KC2 KC4 MC2	*523* *125* *161*	PC1 PC5	*1,109* *17,186*	MDP repository.
Software Quality Journal								
An industrial case study of classifier ensembles for locating software defects [310]	2011	CM1 PC1	*489* *1,109*	PC3	*1,563*	PC4	*1,458*	Promise repository.
Empirical validation of object-oriented metrics for predicting fault proneness models [290]	2010	KC1	*145*					MDP repository. KC1 refers to class level observations.

	MDP°			Promise°			MDP′			MDP⋆			MDP″		
	N	n	$\%fp$	N	n	$\%fp$	N	n	$\%fp$	N	n	$\%fp$	N	n	$\%fp$
CM1	505	36	*9.5*	498	21	*9.8*	344	37	*12.2*	327	37	*12.8*	327	37	*12.8*
JM1	10,878	21	*19.3*	10,880	21	*19.3*	9,593	21	*18.3*	7,842	21	*21.3*	7,782	21	*21.5*
KC1	2,107	21	*15.4*	2,109	21	*15.5*	2,096	21	*15.5*	1,203	21	*26.1*	1,183	21	*26.5*
KC3	458	38	*9.4*	458	39	*9.4*	200	39	*18.0*	194	39	*18.6*	194	39	*18.6*
MC1	9,466	38	*0.7*	9,466	38	*0.7*	9,277	38	*0.7*	1,998	38	*2.3*	1,988	38	*2.3*
MC2	161	38	*32.3*	161	39	*32.3*	127	39	*34.6*	125	39	*35.2*	125	39	*35.2*
MW1	403	36	*7.7*	403	37	*7.7*	264	37	*10.2*	255	37	*10.6*	253	37	*10.7*
PC1	1,107	36	*6.9*	1,109	21	*6.9*	759	37	*8.0*	711	37	*8.6*	705	37	*8.7*
PC2	5,589	35	*0.4*	5,589	36	*0.4*	1,585	36	*1.0*	745	36	*2.1*	745	36	*2.1*
PC3	1,563	36	*10.2*	1,563	37	*10.2*	1,125	37	*12.4*	1,079	37	*12.4*	1,077	37	*12.4*
PC4	1,458	37	*12.2*	1,458	37	*12.2*	1,399	37	*12.7*	1,288	37	*13.7*	1,287	37	*13.8*
PC5	17,186	38	*3.0*	17,186	38	*3.0*	17,001	38	*3.0*	1,723	38	*27.3*	1,711	38	*27.5*
	Basic preprocessing schema						*Alternative preprocessing schema*								

Table 5.2: The impact of preprocessing on data sets

5.3 Empirical setup

This section first describes the experimental design of the study, and subsequently introduces the concept of operating condition to the fault prediction literature. Finally, an overview of the statistical testing procedures is given, constituting our updated benchmarking framework.

5.3.1 Experimental design

As this study entails the assessment of different data preprocessing schemas, and to increase comparability to earlier results, the selection of classification techniques is identical to that of Lessmann et al. [200]. That is, the same set of 22 learners is evaluated across all data sets. Table 5.3 provides a general overview of these learners, of which more detailed descriptions can be found in Section 2.2. In the previous chapter, it was concluded that while a number of alternative learning paradigms have recently been proposed in the context of defect prediction, such as ant colony optimization [320], particle swarm intelligence [70], genetic programming [91, 209] and artificial immune recognition systems [53], none of them has seen widespread adoption for a variety of reasons including lack of comprehensibility, unavailability of software or limited gain compared to more widespread techniques [76].

Furthermore, note that several learners exhibit adjustable parameters, also termed hyperparameters, which allow learners to better model data set characteristics. Where appropriate, a grid-search procedure is adopted, based on a set of candidate values for each hyperparameter. This model selection step is guided by the AUC as criterion of choice, see also Section 2.4.5, comparing all hyperparameter combinations by means of a 10-fold cross validation on the training data.

The study of Lessmann et al. adopted a holdout splitting procedure, partitioning each data set into a separate training and test set. While this is an established procedure in case of larger data sets, the present selection of data

sets demands for a stronger randomized 10-fold cross validation procedure to mitigate the risk of sampling bias during model building [334].

5.3.2 Updated benchmarking framework

Classification performance indicators

Earlier, several performance indicators were covered, including single threshold metrics such as 0-1 loss, and measures aggregating over all possible thresholds such as ROC analysis and the H-measure. Observe that if the class distribution and the cost of misclassifying a (non) faulty module are known, an optimal threshold can be determined, yielding an optimal classifier. In line with Hernández-Orallo et al. [139], the combination of class distribution and misclassification costs is referred to as an operating condition[4], and if this information is unknown when evaluating a model, the use of alternative metrics is advised [200]. In such situation, software defect models should be assessed across all possible operating conditions, using e.g ROC analysis.

Recently, the AUC has been shown to be well suited in the common situation when the operating condition is unknown during model evaluation, while at deployment time, further information becomes available [139]. This corresponds to the situation of developing defect prediction models on historic data without having access to specific knowledge on e.g. the misclassification costs in future projects. When incorporating model output in the software development process, e.g. to streamline testing efforts, typically detailed information on the seriousness of different types of misclassification becomes available [76, 153]. Assume that misclassifying a faulty instance as not fault prone has a misclassification cost c_0, whereas a fault free instance classified as fault prone costs c_1. The costs can be normalized by setting $b = c_0 + c_1$ and $c = c_0/b$, giving the following definition of expected minimum misclassification loss:

$$L_c = \int_0^1 Q_c(T_c(c); c) w_c(c) dc, \tag{5.1}$$

with $Q_c(T_c(c); c)$ the loss at a specific decision threshold $T_c(c)$ and cost proportion c. $w_c(c)$ is a weight distribution on the cost proportions, and is assumed to be a continuous uniform distribution[5]. Depending on the available information on the operating condition at evaluation and deployment time, the threshold $T_c(c)$ will be determined differently. E.g. if the deployment operating condition is already known at evaluation time, an optimal threshold $t^\star$ can be determined, setting $T_c(c) = t^\star$. However, this assumption is rarely met in reality, and we consider the more common situation of knowing the precise operating condition only at deployment time. During model deployment, a threshold will be defined

[4] Throughout this discussion, the class distribution will be considered fixed, such that *operating condition* and *cost proportion* are interchangeable. Note that an equivalent reasoning can be followed when taking both cost and class distributions into account.

[5] Note that recently, e.g. Hand introduced a metric relaxing the assumption of a continuous uniform distribution over the cost proportions [126].

Table 5.3: Overview of classifiers, adopted from Lessmann et al. [200]

Classifier		Philosophy
Statistical classifiers		Strive to construct a Bayes optimal classifier by estimating either posterior probabilities directly (LogReg), or class-conditional probabilities (LDA, QDA, NB, BayesNet) which are subsequently converted into posterior probabilities using Bayes' theorem. LDA/QDA assume a multivariate Gaussian density function, whereas NB is based on the assumption that attributes are conditionally independent, so that class-conditional probabilities can be estimated individually per attribute. BayesNet extends NB by explicitly modeling statements about independence and correlation among attributes. LARS adopts a different approach and consists of a multivariate linear regression model and heuristics to shrink the number of features. RVM has been proposed as an extension of the SVM (see below) which avoids the need to tune certain hyperparameters and may incorporate kernel functions SVMs are unable to process.
Linear Discriminant Analysis	LDA[1]	
Quadratic Discriminant Analysis	QDA[1]	
Logistic Regression	LogReg[1]	
Naïve Bayes	NB[2]	
Bayesian Networks	BayesNet[2]	
Least-Angle Regression	LARS[1]	
Relevance Vector Machine	RVM[1]	
Nearest neighbor methods		Belong to the group of analogy-based methods which classify a module by considering the k most similar examples. The definition of similarity differs among algorithms. An Euclidian distance is used in k-NN whereas K* employs an entropy-based distance function.
k-Nearest Neighbor	k-NN[2]	
K-Star	K*[2]	
Neural Networks		Mathematical representations loosely inspired by the functioning of the human brain. They depict a network structure which defines a concatenation of weighting, aggregation and thresholding functions that are applied to a software module's attributes to obtain an approximation of its posterior probability of being fault prone. The study includes two types of MLP learners which incorporate different approaches to avoid overfitting the training data, i.e. weight decay and Bayesian Learning.
Multi-Layer Perceptron	MLP[1,3]	
Radial Basis Function Network	RBF net[2]	
Support vector machine-based classifiers		Utilize mathematical programming to optimize a linear decision function that discriminates between (non) fault prone modules. A kernel function enables more complex decision boundaries by means of an implicit, nonlinear transformation of attribute values. This kernel function is polynomial for the VP classifier, whereas SVM and LS-SVM consider a radial basis function. L-SVM and LP result in linear classification models.
Support Vector Machine	SVM[1]	
Lagrangian SVM	L-SVM[1]	
Least Squares SVM	LS-SVM[1]	
Linear Programming	LP[1]	
Voted Perceptron	VP[1]	
Decision tree approaches		Recursively partition the training data by means of attribute splits. The algorithms differ mainly in the splitting criterion which determines the attribute used in a given iteration to separate the data. C4.5 induces decision trees based on the information-theoretical concept of entropy, whereas CART uses the Gini criterion. ADT distinguishes between alternating splitter and prediction nodes. A prediction is computed as the sum over all prediction nodes an instance visits while traversing the tree.
C4.5 Decision Tree	C4.5[2]	
Classification and Regression Tree	CART[1]	
Alternating Decision Tree	ADT[2]	
Ensemble methods		Meta-learning schemes that embody several base-classifiers. These are built independently and participate in a voting procedure to obtain a final class prediction. RndFor incorporates CART as base learner, whereas LMT utilizes LogReg. Each base learner is derived from a limited number of attributes. These are selected at random within the RndFor procedure, whereby the user has to predefine their number. LMT considers only univariate regression models, i.e. uses one attribute per iteration, which is selected automatically.
Random Forest	RndFor[1]	
Logistic Model Tree	LMT[2]	

[1] Implemented in the Matlab environment.
[2] Implemented in the Weka environment.
[3] Two multi-layer perceptron learning schemas were considered; MLP-1 refers to the case where the neural network has been trained with a weight decay penalty to prevent overfitting whereas MLP-2 employs a Bayesian learning paradigm.

proportional to the operating condition; however, when assessing the different models, this information is unknown, and models should be evaluated for a wide range of cost proportions. Assuming scores are expressed on an undefined scale, the threshold will be set in such a way that the fraction of cases predicted as fault prone is proportionate to the operating condition, $T_c = \pi_0 F_1(c) + \pi_1 F_1(c)$ with π_l the prior probability of belonging to class l. Then, it can be shown that the expected loss is linearly related to the AUC [139].

Alternatively, one can also consider the situation where further information on the operating condition is unavailable both at deployment and evaluation time, reflecting the difficulty of determining the seriousness of different types of misclassification, even when incorporating the prediction model into the development process. In such case, a reasonable choice could be to set the threshold to a relative quantity taken from a uniform distribution. Again the expected misclassification loss will be linearly related to the AUC.

Finally, note that the AUC is a ranking measure, and thus unaffected by a monotonic transformation on the output of a model; in case the models' scores express a conditional probability of belonging to the class of fault prone modules, alternative measures such as the Brier score can also be considered [48].

Statistical inference

Empirical studies try to assess the impact of one or more factors on an outcome variable, while each factor constitutes multiple levels or treatments. Data is collected per treatment via one or more test attempts and while it seems straightforward to identify the best treatment, the data can be subjected to noise and a statistical framework should be adopted when making statements on the relative performance of treatments. Section 2.3.3 detailed a nonparametric framework consisting of a Friedman test paired with a post-hoc Bonferroni-Dunn test (Section 3.4.6) or a post-hoc Nemenyi test (Section 4.4.4). However, a large number of other statistical tests have been developed, and their inclusion into the statistical framework of a study depends on its precise context. Key to the statistical framework is the minimization of the probability of incorrectly rejecting the null hypothesis (i.e. maximizing statistical power), given a specific Type I error rate. When observations are being assessed across all treatments, paired tests can be used, incorporating the additional information to enhance the power of the statistical framework.

In software defect prediction, the factor typically being investigated is the type of classifier, while the outcome variable is its classification performance across one or more data sets. In such case, the data sets can be seen as blocking factor and paired tests can be used such as the well known repeated measures ANOVA or its nonparametric counterpart, the Friedman test. In this chapter, two factors are of interest: the impact of various levels of data preprocessing and the impact of different classifiers.

A second challenge lies in the fact that prediction studies typically compare many learners over multiple data sets, sometimes using multiple performance indicators. The number of pairwise inferential tests will thus increase accord-

ingly. Whilst for a single test, the likelihood of committing a Type I error is determined by α which is customarily set at 0.05, the Type I error will inflate as more tests are performed. As such, results may not be trustworthy and we cannot be confident that we really should reject a particular null hypothesis. The solution lies in adjusting the value of α to reflect the number of tests being conducted. The most conservative procedure, suggested by Bonferroni, is to divide α by the number of hypotheses; however, more appropriate procedures exist, which are adopted in this study.

Whilst the Friedman test frequently served as an underpinning of empirical studies, Iman and Davenport noted that this test can be overly conservative and propose the following statistic instead

$$F_F = \frac{(P-1)\chi_F^2}{P(k-1)-\chi_F^2} \sim F_{(k-1)[(k-1)\times(P-1)]} \tag{5.2}$$

which is adopted instead of the Friedman test. When comparing the results across different levels of data preprocessing, k equals 4 and P equals $12 \times 22 = 264$. When considering a specific level of data preprocessing, and focussing on the classifiers, k equals 22 and P equals 12. Note that as a rule of thumb, Lehman [199] specified the inequality $k \times P > 30$, which we satisfy in both cases.

If the null hypothesis of equal performance across all treatments is rejected by the Iman and Davenport test, we proceed with a $1\times k$ post-hoc Bonferroni procedure, comparing one treatment (the control treatment) to all others. The test statistic when comparing treatment m with treatment m' is:

$$z = \frac{AR_m - AR_{m'}}{\sqrt{\frac{k(k+1)}{6P}}} \sim N(0,1). \tag{5.3}$$

The probability that the value of z comes from a normal distribution is then compared to an appropriate error rate. The simplest approach, the Bonferroni-Dunn test, divides the family wise error rate α by the number of hypotheses, $k-1$. However, this test is often too conservative, and in case of independent hypotheses, it has been shown that the Bonferroni-Rom test is more powerful [108, 248]. This test recursively identifies an appropriate adjusted significance level α' per hypothesis, controlling the family wise error rate α at exactly the nominal level (e.g. $\alpha = 0.05$).

Note that e.g. Lessmann et al. [200] adopted a $k\times k$ Nemenyi test, comparing all treatments (classifiers) to each other. When comparing against a single control treatment, the number of test hypotheses is much smaller ($k-1$ instead of $\binom{k}{2}$), giving an increase in statistical power.

Finally, we add the rider that all significance tests must be interpreted with some caution since they say nothing about the effect size. When dealing with very large sample sizes it is possible to obtain very significant results of very small effects, in other words, differences that have little practical importance.

5.4 Experimental results

This section quantifies the dissimilarity between data stemming from the MDP and the Promise repository in the next paragraph. Subsequently, as models built on either repository are not found to differ significantly, the attention is shifted towards the alternative preprocessing schema using only the MDP repository data. Reconciling the issues raised by Gray et al. [117], and defect prediction reality, it is argued that the proposed MDP* preprocessing approach is the most feasible and therefore, the third part presents the detailed results hereof.

5.4.1 Basic preprocessing schema

While the NASA data have been repeatedly investigated, the discrepancies amongst data sets presented in Section 5.2 are often all but ignored. Firstly, the predictive performance of defect models built on data stemming from MDP and Promise repositories, referred to as MDP$^\circ$ and Promise$^\circ$ respectively, is compared and we find little impact upon the average ranks (ARs) of individual learners across both repositories. Typically, the AR differ by less than one point, with a maximum of 2.16 points. Therefore, we decided to focus on the MDP$^\circ$ data as baseline for the remainder of this study. Note the data sets were originally published by the NASA as part of a software measurement project to assist software practitioners in gaining insight into software defect occurrence, motivating our choice of baseline. Nevertheless this does not mean scientists can ignore data quality particularly for the data-driven techniques we are investigating in this study.

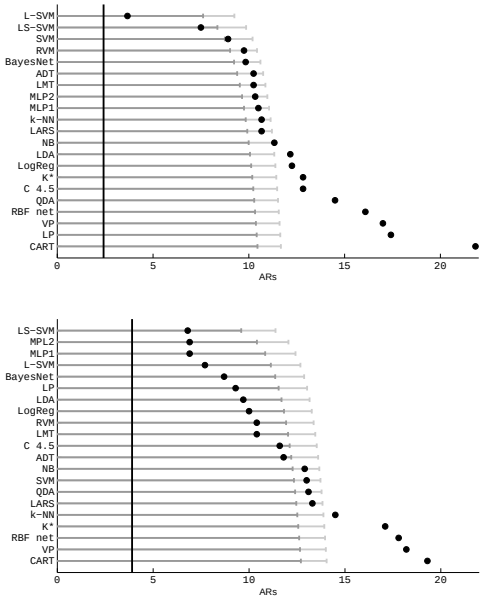


Figure 5.1: Bonferroni-Rom ranking on the MDP$^\circ$ data and on the results taken from Lessmann et al., in top and bottom panel respectively

When investigating the performance of individual learners on the MDP$^\circ$ data, the Davenport-Iman test which assesses whether there are any differences in predictive performance amongst learners is employed in a first phase. As this test resulted in a p-value smaller than 10^{-10}, the null hypothesis of equal performance amongst all learners is rejected and subsequently, a post-hoc

Bonferroni-Rom test is performed, comparing one control classifier to all others. As Random Forest (RndFor) is found to induce the best performing defect prediction models, in line with e.g. [76, 118], this learner is selected as control classifier. The outcome of this test is presented in Fig. 5.1, top panel. The horizontal axis in this figure corresponds to the AR of a technique across all data sets. The dots represent the AR of each technique, while the vertical line corresponds to the AR of the control classifier (RndFor). Note the Bonferroni-Rom test sets an appropriate adjusted significance level for each comparison with the control classifier; if a dot is located in the darker (lighter) shaded area, it is found to be not outperformed by RndFor at the 5% (1%) level.

Fig. 5.1, bottom panel, provides the original results of Lessmann et al., reevaluated using the Bonferroni-Rom test. Note the close resemblance in the relative performance, identifying a similar set of techniques as best/worst performing. Minor differences can be attributed to an alternative data selection, different training/test set splitting and implementation details of individual learners. One notable exception is Linear Programming (LP), which is attributed the second highest AR on the MDP° data, while found not being outperformed by RndFor in the previous study. As can be observed, extending the selection of data sets to 12 results in an increased number of techniques found to be outperformed. Earlier, it was concluded that 'the importance of the classification model may have been overestimated in the previous research' [200], which is supported by the statistically insignificant difference at the 1% level between RndFor and e.g. various support vector machine implementations, both neural network learners and the Logistic Model Trees (LMT) ensemble classifier.

5.4.2 Alternative preprocessing schema

Addressing the concerns raised by Gray et al. in a stepwise fashion, alternative versions of each data set can be derived, referred to as MDP', MDP⋆ and MDP" respectively, see Section 5.2. By considering the combination of technique and data set as blocking factor, and data versions as treatment, the impact of the alternative preprocessing schema can be quantified using the same statistical framework. In a first step, the Iman-Davenport test results in a p-value smaller than 10^{-10}, rejecting the null hypothesis of equal performing defect predictors across versions. Subsequently, the Bonferroni-Rom procedure

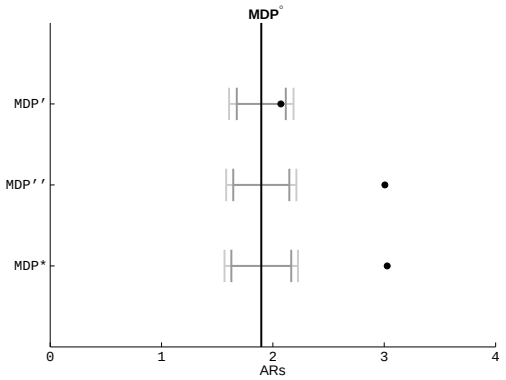


Figure 5.2: Bonferroni-Rom ranking on the alternative preprocessing schema

compares the performance of defect predictors built on the alternative data sets to those induced on the MDP° data, Fig. 5.2. The vertical line in this figure again represents the AR of the control treatment, MPD°, while the ARs of other data versions are marked by dots. It is apparent that especially duplicate removal yields a significant deterioration in classification performance, cf. MDP⋆ and MDP″; defect prediction models built on the MDP′ data are not found to be significantly worse performing at the 5% level than those built on the MDP° data, while both MDP⋆ and MDP″ data result in significantly lower performing models at the 1% level. Furthermore, a contrast estimation was performed, showing the expected differences between two data versions' performance in terms of AUC, see Table 5.4. Here, the contrast estimation procedure outlined by García et al. is adopted, which makes an all pairwise comparison between the medians of samples of results [108]. It can be seen that the drop in terms of AUC is rather limited; e.g. comparing MDP° and MDP⋆, an estimated drop of 3.578 points can be observed.

Focussing on individual data sets, different patterns can however be observed. Fig. 5.3a and Fig. 5.3b display the impact of subsequent preprocessing steps on each data set; the bold line illustrates the overall average while separate data sets are represented by a dashed, dotted or full line respectively. The data sets with the largest fraction of duplicates include PC5 (89.9%), MC1 (78.5%) and KC1 (42.6%), which all exhibit a sharp performance drop upon discarding these observations in MDP⋆. Note that while data sets with large quantities of duplicates typically exhibit a similar pattern, also other aspects come into play, such as the total number of instances and the proportion of fault prone instances. Figures 5.3c, 5.3d and 5.3e further zoom in on PC1, PC2 and PC5 respectively, illustrating these aspects. The bars indicate the proportion of observations retained at different preprocessing steps while the bold line again provides the classification performance of defect predictor models built on a specific version of the data. PC1 and PC2 show that discarding logically incorrect cases and cases with missing values can positively affect classification performance while on the other hand, PC5 illustrates the impact duplicate removal can have on duplicate-heavy data sets: a sharp drop in classification performance can be seen upon discarding these observations. Note that PC2 is the most imbalanced data set, which partially explains the observed trend for this data set as discarding observations seems to improve performance. The impact of removing conflicting values is typically limited, but is still noteworthy in some cases; this is e.g. true for PC1. Finally, it should be pointed out that some data sets, e.g. PC2⋆ and PC2″, are identical, but result in slightly different performing models due to sampling effects.

In reality, software modules with similar static code characteristics are often present in a software project due to the limited discriminatory abilities of some static code characteristics [117], however, since machine learning models should be validated on a set of unseen samples, Gray et al. argued in favor of removing duplicate instances. Inconsistent instances however, i.e. modules with similar static code features but different class label, can be seen as an expression of the uncertainty that a module with specific characteristics will be faulty. The

fraction of inconsistent instances which cannot be classified correctly can be regarded as a theoretic upper bound of the generalization performance on a data set and removal of these instances artificially mitigates this upper bound. This situation coincides with the MDP* data, and the results of individual learners on this data are further investigated in the following paragraph.

	MDP$^\circ$	**MDP$'$**	**MDP***	**MDP$''$**
MDP$^\circ$	–	0.635	3.579	3.661
MDP'	-0.635	–	2.944	3.026
MDP*	-3.579	-2.944	–	0.082
MDP"	-3.661	-3.026	-0.082	–

Table 5.4: Contrast estimation on the alternative preprocessing schema

5.4.3 Benchmarking on the MDP* data

Assessing models built on the MDP* data, individual learners constitute the treatments, and the collection of MDP* data sets represents the test attempts. Following the rejection of the null hypothesis of the Iman-Davenport test, with a p-value of again smaller than 10^{-10}, the Bonferroni-Rom test is applied, of which the outcome is given in Fig. 5.4. RndFor is attributed the lowest AR, and is thus again considered as control classifier.

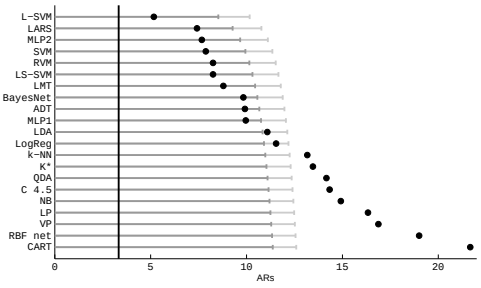
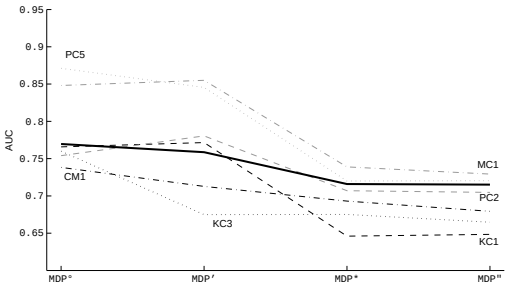
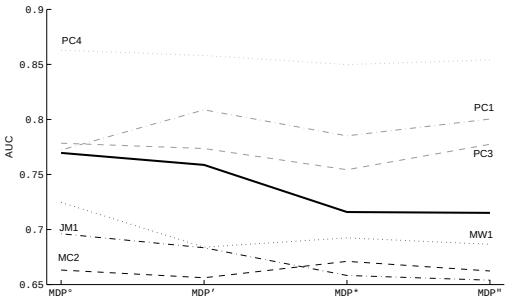


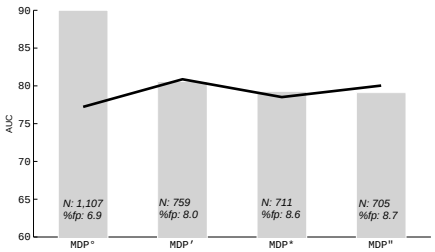
Figure 5.4: Bonferroni-Rom ranking on the MDP* data

Table 5.5 presents the detailed classification performance in terms of AUC on the MDP* data. The last column provides the Average Rank (AR) of each technique, with the best performing technique, lowest AR, indicated in bold and underlined. The AR of a technique that is not significantly different from the best performing technique at 5% is tabulated in boldface font, while results significantly different at 1% are displayed in italic script. Classifiers differing at the 5% level but not at the 1% level are displayed in normal script. The Bonferroni-Rom test is used during these assessments. From these results, it can be seen that the set of best performing techniques is similar to those determined by Lessmann et al. [200]. Identifying RndFor as best performing learner, the set of techniques being significantly outperformed by this learner on the MDP* data is in fact a subset of the 2008 results. Furthermore, the relative performance of individual techniques is comparable to a large extent, Linear Programming (LP) and Least-Angle Re-

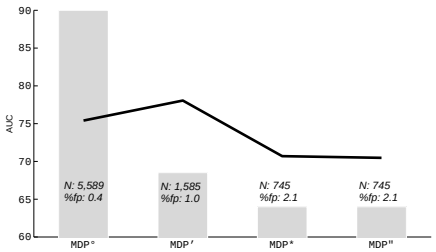


(a) Data sets exhibiting more pronounced variations

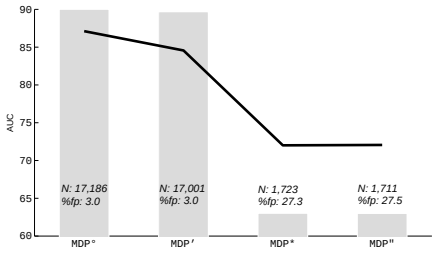




(c) Preprocessing impact on PC1



(d) Preprocessing impact on PC2



(e) Preprocessing impact on PC5

Figure 5.3: Decomposition of the results: evolution of individual data sets

gression (LARS) exhibiting the largest discrepancies. In the 2008 study, LARS was found to be outperformed at the 5% level but not at the 1% level, while it is found to be one of the top performing techniques on the MDP* data. While many techniques are not significantly outperformed by RndFor, it is interesting to note that a technique such as BayesNet, which results in comprehensible models, is also amongst these. Univariate decision tree algorithms such as C4.5 and CART are however significantly outperformed, indicating models with axis-parallel decision boundaries are unable to fully capture the underlying patterns of the data. Multivariate linear models such as LogReg and LDA are only outperformed at the 1% level.

Adopting again the contrast estimation procedure outlined by García [108], Table 5.6 provides the estimated difference between two classifiers' performance. As concluded by the Bonferroni-Rom procedure, all learners have inferior performance to RndFor, the best performing classifier. The worst classifier is CART, which is outperformed by all other techniques by a large margin. This learner is unable to deal with the class imbalance present in most data sets, returning models which perform equal to random guessing, AUC ≈ 0.5. Note that also the second to last technique, RBF net, is outperformed by RndFor by a considerable margin, i.e. 12.5 AUC points. The performance differences between other techniques are less striking, but even small differences can result in significant cost savings in the software development process [310]. This further illustrates the necessity of having proper statistical procedures in place, allowing to verify whether small performance differences are due to chance or signal a recurring trend [108].

It should also be pointed out that a large number of alternative classification metrics have previously been considered in software defect prediction (see e.g. Table 2.3 for an overview). As it is a general finding that alternative metrics often lead to different conclusions, see e.g. [13, 325], the findings of this study were validated in terms of the Brier score, a classification metric recently shown to be suitable when model output represents the posterior class probability of belonging to the class of (non) fault prone modules [139]. The Brier score results support the AUC-based results presented above. They show a very similar trend with respect to the impact of data preprocessing and individual learners' performance, respectively.

Table 5.5: Comparison of classifier performance: out-of-sample performance on MDP*

AUC	**CM1***	**JM1***	**KC1***	**KC3***	**MC1***	**MC2***	**MW1***	**PC1***	**PC2***	**PC3***	**PC4***	**PC5***	**AR**
LDA	74.5	67.3	66.4	66.6	77.6	63.8	75.3	79.6	79.5	76.6	82.8	72.0	11.08
QDA	73.9	65.1	65.0	68.2	72.6	67.7	70.5	78.8	74.1	75.0	78.9	67.1	*14.17*
LogReg	74.4	67.3	66.4	65.5	76.7	64.3	75.2	80.0	81.0	76.1	81.9	72.0	11.54
NB	66.1	63.2	66.8	62.7	66.3	71.4	73.4	74.7	72.5	72.4	82.5	70.6	*14.92*
BayesNet	75.7	67.7	64.7	65.7	79.1	64.2	69.3	82.5	82.1	76.7	87.8	74.9	**9.83**
LARS	73.8	67.7	68.0	71.1	79.3	68.5	71.7	83.5	74.4	82.8	88.9	74.6	**7.42**
RVM	72.1	67.5	67.6	70.5	50.0	71.8	70.5	84.2	80.9	79.5	90.9	74.8	**8.25**
k-NN	65.8	65.6	63.9	70.3	71.1	69.4	71.3	76.1	64.5	76.4	82.9	74.7	*13.17*
*K**	68.2	64.5	63.7	67.9	88.2	62.6	68.6	80.4	81.9	75.3	81.4	72.5	*13.46*
MLP1	70.6	68.5	65.2	64.3	77.5	74.1	70.0	84.5	79.5	75.6	90.6	73.7	**9.96**
MLP2	75.0	68.3	66.6	72.2	79.8	75.4	72.5	85.5	71.2	73.8	88.0	72.8	**7.67**
RBF net	60.5	65.9	62.6	56.9	69.1	61.8	62.9	58.7	40.2	72.0	76.7	72.1	*19.00*
SVM	59.3	67.2	67.3	75.8	82.4	74.8	75.0	80.4	83.2	71.5	91.1	74.6	**7.88**
L-SVM	76.1	67.9	67.9	74.3	79.5	73.5	75.4	85.1	77.9	76.9	90.7	74.5	**5.17**
LS-SVM	73.1	69.5	67.9	59.5	82.5	67.0	66.6	85.5	48.1	82.8	91.1	77.6	**8.25**
LP	66.1	63.5	64.4	63.7	65.6	67.0	58.8	67.7	64.7	79.0	86.8	67.1	*16.33*
VP	67.6	62.8	64.8	68.6	47.1	62.6	74.1	74.3	47.6	71.3	81.9	69.4	*16.88*
C 4.5	60.6	64.8	59.7	71.5	78.5	66.4	61.3	83.9	64.2	76.1	89.5	71.5	*14.33*
CART	51.2	49.5	45.1	56.1	47.2	46.4	51.5	52.0	49.5	52.5	49.8	50.4	*21.67*
ADT	77.4	67.5	63.3	67.7	83.6	69.3	64.2	79.6	76.9	76.7	91.4	73.4	**9.92**
RndFor	73.2	69.2	68.3	75.4	91.4	73.5	71.5	85.0	82.7	83.3	93.4	79.1	**3.33**
LMT	69.6	67.6	65.6	70.8	80.5	60.8	73.6	85.1	78.5	77.4	90.8	74.5	**8.79**

Table 5.6: Contrast estimation on the MDP* data

	NB	**RVM**	**LogReg**	**LDA**	**QDA**	**BayesNet**	**K***	**k-NN**	**MPL2**	**RBF net**	**L-SVM**	**LP**	**VP**	**ADT**	**C 4.5**	**LMT**	**SVM**	**MLP1**	**CART**	**RndFor**	**LARS**	**LS-SVM**
NB	–	-4.859	-3.059	-3.242	-1.290	-3.834	-1.783	-1.536	-4.816	4.525	-5.742	1.384	0.988	-3.807	-1.444	-4.341	-5.395	-4.146	19.073	-8.016	-5.036	-5.296
RVM	4.859	–	1.800	1.617	3.569	1.025	3.077	3.323	0.043	9.384	-0.882	6.244	5.847	1.052	3.416	0.518	-0.535	0.713	23.933	-3.157	-0.177	-0.437
LogReg	3.059	-1.800	–	-0.183	1.769	-0.775	1.276	1.523	-1.757	7.584	-2.683	4.443	4.046	-0.748	1.615	-1.282	-2.336	-1.087	22.132	-4.957	-1.978	-2.237
LDA	3.242	-1.617	0.183	–	1.952	-0.592	1.459	1.706	-1.574	7.767	-2.500	4.626	4.229	-0.565	1.798	-1.099	-2.153	-0.904	22.315	-4.774	-1.795	-2.054
QDA	1.290	-3.569	-1.769	-1.952	–	-2.544	-0.492	-0.246	-3.526	5.815	-4.452	2.675	2.278	-2.517	-0.154	-3.051	-4.104	-2.856	20.363	-6.726	-3.746	-4.006
BayesNet	3.834	-1.025	0.775	0.592	2.544	–	2.051	2.298	-0.982	8.359	-1.908	5.218	4.821	0.027	2.390	-0.507	-1.561	-0.312	22.907	-4.182	-1.203	-1.462
K*	1.783	-3.077	-1.276	-1.459	0.492	-2.051	–	0.247	-3.033	6.307	-3.959	3.167	2.770	-2.025	0.339	-2.559	-3.612	-2.363	20.856	-6.234	-3.254	-3.514
k-NN	1.536	-3.323	-1.523	-1.706	0.246	-2.298	-0.247	–	-3.280	6.061	-4.206	2.920	2.523	-2.271	0.092	-2.805	-3.859	-2.610	20.609	-6.480	-3.500	-3.760
MPL2	4.816	-0.043	1.757	1.574	3.526	0.982	3.033	3.280	–	9.341	-0.926	6.200	5.803	1.009	3.372	0.475	-0.579	0.670	23.889	-3.200	-0.221	-0.480
RBF net	-4.525	-9.384	-7.584	-7.767	-5.815	-8.359	-6.307	-6.061	-9.341	–	-10.267	-3.140	-3.537	-8.332	-5.969	-8.866	-9.919	-8.671	14.548	-12.541	-9.561	-9.821
L-SVM	5.742	0.882	2.683	2.500	4.452	1.908	3.959	4.206	0.926	10.267	–	7.126	6.729	1.935	4.298	1.401	0.347	1.596	24.815	-2.274	0.705	0.446
LP	-1.384	-6.244	-4.443	-4.626	-2.675	-5.218	-3.167	-2.920	-6.200	3.140	-7.126	–	-0.397	-5.192	-2.828	-5.726	-6.779	-5.530	17.689	-9.401	-6.421	-6.681
VP	-0.988	-5.847	-4.046	-4.229	-2.278	-4.821	-2.770	-2.523	-5.803	3.537	-6.729	0.397	–	-4.795	-2.431	-5.329	-6.382	-5.133	18.086	-9.004	-6.024	-6.284
ADT	3.807	-1.052	0.748	0.565	2.517	-0.027	2.025	2.271	-1.009	8.332	-1.935	5.192	4.795	–	2.363	-0.534	-1.587	-0.339	22.880	-4.209	-1.229	-1.489
C 4.5	1.444	-3.416	-1.615	-1.798	0.154	-2.390	-0.339	-0.092	-3.372	5.969	-4.298	2.828	2.431	-2.363	–	-2.897	-3.951	-2.702	20.517	-6.572	-3.593	-3.852
LMT	4.341	-0.518	1.282	1.099	3.051	0.507	2.559	2.805	-0.475	8.866	-1.401	5.726	5.329	0.534	2.897	–	-1.053	0.195	23.414	-3.675	-0.695	-0.955
SVM	5.395	0.535	2.336	2.153	4.104	1.561	3.612	3.859	0.579	9.919	-0.347	6.779	6.382	1.587	3.951	1.053	–	1.249	24.468	-2.622	0.358	0.098
MLP1	4.146	-0.713	1.087	0.904	2.856	0.312	2.363	2.610	-0.670	8.671	-1.596	5.530	5.133	0.339	2.702	-0.195	-1.249	–	23.219	-3.870	-0.891	-1.150
CART	-19.073	-23.933	-22.132	-22.315	-20.363	-22.907	-20.856	-20.609	-23.889	-14.548	-24.815	-17.689	-18.086	-22.880	-20.517	-23.414	-24.468	-23.219	–	-27.089	-24.110	-24.369
RndFor	8.016	3.157	4.957	4.774	6.726	4.182	6.234	6.480	3.200	12.541	2.274	9.401	9.004	4.209	6.572	3.675	2.622	3.870	27.089	–	2.980	2.720
LARS	5.036	0.177	1.978	1.795	3.746	1.203	3.254	3.500	0.221	9.561	-0.705	6.421	6.024	1.229	3.593	0.695	-0.358	0.891	24.110	-2.980	–	-0.260
LS-SVM	5.296	0.437	2.237	2.054	4.006	1.462	3.514	3.760	0.480	9.821	-0.446	6.681	6.284	1.489	3.852	0.955	-0.098	1.150	24.369	-2.720	0.260	–

5.5 Conclusion

This chapter tried to answer the call of Gray et al. [117] , who pointed out several data quality issues present in the NASA defect prediction data sets. Building upon the study of Lessmann et al., this chapter first identifies a number of differences between data stemming from the NASA MDP and the Promise repository. Overall the effect of these differences are not found to be substantial, so we proceed by outlining a stepwise preprocessing procedure on the MDP data, addressing specific data quality issues. That said, we believe researchers should still make strenuous efforts to deal with poor data quality and make their data preprocessing explicit when reporting experimental results.

The presence of duplicate instances significantly impacts software defect prediction performance, indicating earlier studies might have overestimated the predictive power of such models. A contrast estimation procedure further quantifies this impact by indicating a drop of less than 4 points in terms of AUC, thus confirming the importance of proper data preprocessing procedures. In general, adopting a machine learning approach is identified as an important piece in the solution to tackle software quality issues. Zooming in on individual data sets paints however a different picture; data sets with more than 40% duplicates (i.e. PC5, MC1 and KC1) seem more seriously impacted by the issues outlined by Gray et al. and removal of these duplicate cases leads to a performance penalty of around 10 AUC points. Proposing a specific data preprocessing procedure which removes duplicates while retaining conflicting cases, referred to as MDP*, the large-scale benchmarking experiment of Lessmann et al. is repeated. While Random Forest is again found to be the best performing technique, multivariate linear models are found to be outperformed at the 5% level while learners resulting in univariate decision trees are outperformed at the 1% level, partially confirming earlier results.

The motivation of our study was the warning of Gray et al. that the presence of duplicates may have invalidated earlier findings; our results provide an answer to this by considering a plethora of classifiers and a stepwise preprocessing approach, signalling these data quality problems are less of an issue to most data sets and learners. Moreover, by repetition of our 2008 study on the MDP* data, we have shown that general findings on the relative performance of classifiers are in fact still valid.

Finally, this study serves as a complement to Lessmann et al. by proposing an alternative statistical framework, paired with the introduction of contrast estimation into software fault prediction, incorporating the recent work of García et al. [108].

There are three roads to ruin; women, gambling and technicians. The most pleasant is with women, the quickest is with gambling, but the surest is with technicians.

Georges Pompidou, 1911 – 1974

6

Cross release validation: a case study on the Android platform

In the final chapter of this dissertation, we challenge the perennial believe that historic data can be utilized to predict future faults. In contrast with previous chapters, the situation in which only information regarding the fault proneness of precursory releases is available is investigated and compared to a more common cross validation approach, see also Section 2.3.2. This naturally constraints our selection of data sets, discarding e.g. the NASA MDP data as possibility. As such, a collection effort on the open source Android platform was instigated, mining its development history. Acknowledging the importance of reproducibility, details of our data collection effort are also presented which can offer a useful guideline to other researchers. Note that while development archives often have their own peculiarities, many of the underlying ideas remain universally applicable.

This chapter is based on the following manuscript

- K. Dejaeger, and B. Baesens, "On the validation of software defect models," *Technical document*

6.1 Introduction

Software development is often perceived to be a difficult and time intensive activity, and various research tracks are assisting practitioners herein by leveraging the available historic information on prior development efforts, including software reliability modeling [112], effort forecasting (Chapter 3) and software fault prediction (Chapters 4 and 5). Practitioners typically adhere to a software development methodology to better structure this activity and the waterfall approach is a prime example hereof. Alternatives include rational unified processes [191] and methodologies inspired on open source software (OSS) development [166], and almost invariantly, they exhibit the common property of including a software verification and validation step in which the quality of the code is tested. When focussing on OSS development, an often recurring theme seems to be that the code is developed by geographically distributed teams of volun-

teers who do not receive any direct compensation for their efforts. As such, projects often lack any form of central project management and violate also other well established software development practices such as the formation of small teams, the enforcement of specific development guidelines, or the need for requirement analyses [131, 307]. Recently, the shift towards OSS has been gaining momentum, rivaling commercial software solutions. Protagonists include the Eclipse foundation, Mozilla Firefox, OpenOffice, Linux, and the Android platform, which are also often stimulated by contributions from companies like IBM (Eclipse) and Google (Android). Byproduct to this evolution, extrinsic incentives such as monetary rewards and developer visibility are nowadays playing an increasingly important role, suggesting that OSS communities are largely welcoming commercial efforts [267].

Android is an open source platform for mobile devices such as smartphones and tablet computers and is based on the Linux kernel while also drawing upon other open source projects such as squeak (a bluetooth package) and yaffs (Yet Another Flash File System). Evidently, the source code of the Android platform is made available to the public, licensed under the Apache Licence v2, except for the Linux kernel and its modifications, which are published under the GNU Public Licence v2. The source code was first released in 2007, when Google founded the Open Handset Alliance together with 33 others, including Intel, T-Mobile and Samsung, and is currently being further developed as part of the Android Open Source Project. Furthermore, an estimated 700.000 android devices are being activated daily, signposting the economic importance of this OSS project. Over time, many individuals contributed to this project, although it was recently concluded that 'while there is healthy contribution from non-google/android community, these contributions are restricted to pockets of the code base and not very widespread' [292]. As such, the Android Open Source Project can be catalogued as an OSS project with strong commercial influences. Remark that a 6 month development life cycle is in place for major releases, each receiving a specific nickname, which are followed by several minor releases. Fig. 6.1 provides a chronological overview on the development of the Android platform.

It has been pointed out that the introduction of software testing processes to identify software faults in a timely manner is crucial since corrective maintenance costs inflate exponentially if faults are detected later in the software development life cycle [38]. Moreover, recent studies indicated that an important part of the expenses are made during the testing phase, reporting fractions as high as 60% of the total development cost [29,85]. Early warning mechanisms which flag error prone parts in the code base can provide a (partial) solution hereto by allowing to streamline testing efforts. If such mechanisms take static code features derived from source code as an input, and draw upon machine learning literature, they are also commonly referred to as fault prediction models and their implementation in the Android platform seems especially desirable given the fast succession of releases and its commercial background.

This study contributes to the literature by investigating the feasibility of fault prediction in the context of the Android platform by application of a

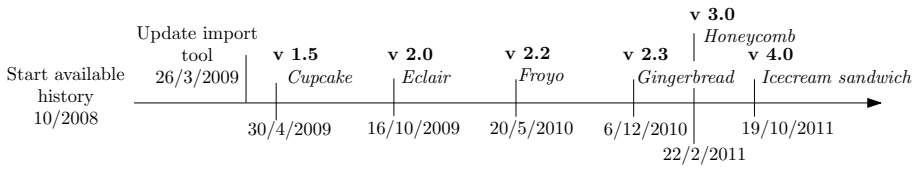


Figure 6.1: Android platform chronological overview

broad scala of machine learning techniques, corresponding to different learning paradigms. More specifically, our selection resembles that of the previous chapter, while also including two learners which, based on prior software fault prediction literature, could prove to make a valuable addition to the current set of techniques, i.e. rotation forest and an implementation of oblique decision trees (oblique classifier 1). A second question raised by this study concerns the validation of empirical results when data on multiple releases are available. Typical validation procedures include holdout splitting [200], x-fold cross validation [4, 70] and leave-one-out cross validation; however, the recently introduced notion of concept drift in software fault prediction underscored the necessity of adopting an alternative cross release approach, as it was concluded that the quality of fault prediction varies over time [87]. As such, a more viable cross release validation accounting for the ordering in software releases is also investigated. Building upon earlier work [173, 251, 342], both research questions are rigorously explored by adopting again the updated benchmarking framework presented in Chapter 5.

Note that some work also discussed the feasibility of *cross project* validation, training and testing models on data stemming from different projects [134, 317]; given the mixed results reported by these studies, see e.g. [178, 184, 317, 344], and the opportunities offered to researchers by the open source nature of the Android Open Source Project, such approach was however not considered in this chapter.

The remainder of this chapter is structured as follows.
Section 6.2 outlines the data collection process, providing a set of useful guidelines to other researchers.
Section 6.3 explains the empirical setup.
Section 6.4 elaborates on the results and situates them within the literature.
Section 6.5 summarizes our findings and offers a discussion on their ramifications.

6.2 Mining the Android platform

This section details the procedure by which the data were collected. This was in fact a two-step effort, in which an exploratory collection effort, detailed in the first paragraph, served as an input to the actual data acquisition process, explained in the second paragraph.

6.2.1 Exploratory collection effort

Data on the development of the Android platform was made available in the context of the Mining Software Repositories (MSR) challenge 2012 [287]. This included a Git repository extract dating from October 2008 to June 2011 as well as a dump of the public bug tracker, both accessible through the challenge website[1]. Remark that the Android platform is structured into several smaller projects, which is also reflected in the structure of the Git repository extract; each project is maintained as a separate repository.

As a first step, both the Git repository extract and the bug tracker dump were loaded into a database, providing us with a coarse overview of the Android platform. Each commit in a Git repository is uniquely identified by a SHA1 hash and identifies committer and files changed by the commit. Furthermore, Git offers the opportunity to specify the reason for each commit through a free text commit message, which allows to link commits to entries in a bug tracker [19, 332][2]. When assigning a (non) fault label to a source file on the basis of commit messages, the linkage with the bug tracker can serve as an instrument to instill additional confidence into the labeling. For instance, a commit referring to a bug which was closed several months before submission of the commit can be deemed questionable, and also the status of a bug (e.g. 'open', 'won't fix', 'resolved') can be taken into consideration [345]. However, since the bug IDs used during development typically refer to an inaccessible, private bug tracker, this approach was not possible.

Secondly, the head of the main trunk of the Android platform was checked out, which includes around 192,000 files or a total of ~4.15 GB, not including the kernel. As indicated earlier, the Android platform constitutes several smaller projects, such as externally developed applications, custom android versions, and the Linux kernel, to which a small number of modifications have been made[3]. As such, the history of specific parts of the Android platform was deemed uninteresting when investigating early warning mechanisms from the viewpoint of the Android developers. Moreover, as it has been recognized that considerable differences exist between the empirical distribution of static code features of different programming languages [329], it was decided to limit our study to java source code files, in line with e.g. [345].

[1] www.2012.msrconf.org

[2] Observe that there also exist content management systems with integrated bug tracker. Examples hereof are Gemini and Jira.

[3] Recently, the 'Android mainlining project' was founded to integrate the changes made in context of the Android platform back into the Linux kernel project. It was recognized that about 250 commits differentiate both versions. www.elinux.org/Android_Mainlining_Project

The following exclusion criteria were adopted to delineate the scope of this study.

- Exclude projects containing less than 1,000 java source files.

Excluded projects:

Abi	Development	Frameworks/ex	Hardware
Bionic	Device	Frameworks/media	ndk
Bootable	Docs	Frameworks/opt	Prebuilt
Build	Frameworks/compile	Frameworks/support	System

- Exclude externally developed projects.

Excluded projects:

Packages, Externals

- Exclude projects directly related to testing processes.

Excluded projects:

cts

Table 6.1 provides an overview of the retained projects. The Git repository extract made available for the MSR challenge contained data for a three year time frame; however, the data up to 26/3/2009 are not useable as commit messages and subject lines typically provide no information on the reason of the commit. A change in the logging procedure allowed for additional commit information to be captured from that point onwards. Remark also that part of the history of the Libcore project was not included in the Git repository extract, further motivating the final collection effort in which only data directly obtained from the Android Open Source Project was considered.

6.2.2 Final collection effort

Considering the 6 month development cycle, data on 4 major releases was collected and matched with 6 months of post release defect data. The chronological overview of Fig. 6.1 indicates that the first release on which sufficient data is

Project	**Description**
Dalvik	Dalvik is a virtual machine which allows to compile apps to byte code and executing them in this virtual machine; is based on the now defunct Harmony Apache project. Contains both Java and C source code files.
Frameworks base	A collection of smaller parts which has been developed or adapted to the Android project such as telephony and camera services. Contains both Java and C source code files.
Libcore	A library of files which are frequently used by other parts of the Android platform; was initially part of the Dalvik subproject until 30/04/2010. These files have mainly been written in Java.
SDK	The software development kit (SDK) which is offered to Android developers to assist in creating applications for the Android platform.

Table 6.1: Description of study scope

available is Eclair, and also 3 more recent releases were considered; i.e. Froyo, Gingerbread and Icecream sandwich. The source code of all projects listed in Table 6.1 was downloaded for each release, totalling ~1.3 GB of source code, documentation and support files. A set of static code features was derived from the Java source files, except for those related to testing procedures. An open source tool called 'Perst' was adopted hereto, which makes use of a code parser generated by JavaCC [311]. The most recent version was obtained and modified to extent the set features which can be mined by this tool. As such, the set of features includes line counts, McCabe and Halstead metrics, and their calculation is provided in Table 6.3.

LOC based metrics
Lines Of Code (LOC) has been used as an approximation of software size since the late sixties and more recently has been adopted as a proxy for software complexity in software fault prediction studies [94]. Being highly dependent on the selected programming language, a number of alternative measures were introduced in the 70s to quantify software complexity. Two such sets of metrics are McCabe complexity metrics and Halstead software science metrics.

Halstead metrics
Maurice Halstead defined a set of metrics based on the idea that a program or module could be regarded as a sequence of tokens, i.e. a sequence of operators and operands [122]. Based on the counts of these tokens, he specified a number of derivative measures which are sometimes referred to as 'software science' metrics. In constructing these metrics, Halstead drew upon insights from cognitive psychology research by taking the mental abilities of the human brain into account. His work has been criticized for several reasons, including for not defining a clear and consistent counting strategy of the number of tokens and for issues with the unit of measurement of several derived metrics [3]. Despite these drawbacks, Halstead metrics remain commonly used by software engineering practitioners [317].

McCabe metrics
Thomas McCabe regarded program complexity from a different perspective, relating program complexity to the number of linearly independent paths through a program. Hereto, a program or module is mapped to a flowgraph, where each node corresponds to a block of code where the flow is sequential and the arcs correspond to branches in the program. An often used McCabe metric is the well known cyclomatic complexity, $v(G)$, which is calculated as $v(G) = e - n + 1$. e represents the number of edges herein while n stands for the number of nodes in the flow chart.

Note that static code features are known to be correlated; previous work examining such features found that these could be grouped into four categories [202]. A first category related to metrics derived from flowgraphs (i.e. McCabe metrics) while a second category contained metrics related to the size and item

count of a program. The two other categories represented different types of Halstead metrics, motivating our selection of static code features.

Furthermore, an extract of the Git repository was directly obtained from the Android developers website in April 2012[4], which was parsed into a database, adopting a similar approach as followed by Fisher et al. [99]. The commit messages served as input to a text matching procedure to label source files as faulty or not. Note that Git utilizes a SHA1 hash to uniquely identify commits, motivating the addition of the last clause to remove incorrect bug referrals. The regular expressions we adopted, together with some examples in italics, are as follows:

- http://b/\d+
  http://b/123
- http://b/issue\?id=\d+
  http://b/issue?id=123
- http://code.google.com/p/android/detail\?id=\d+
  http://code.google.com/p/android/detail?id=123
- bugi?d?\s*\#?:?=?\s*\d+[^ a-zA-Z]\>
  bug: 123, bugid = 123
- fixe?d?\s*\#?\[?:?=?\s*\d+[^ a-zA-Z]\>
  fix: 123, fixed = 123
- issue\s*\#?:?=?\s*\d+[^ a-zA-Z]\>
  issue: 123, issue = 123

Table 6.4 provides a per project fault ratio in the top panel, and aggregate observation counts and fault ratios in the bottom panel. When compared to the collection effort of Zimmermann et al. [345], who reported fault ratios around 10 to 15%, and those reported as part of the NASA MDP programme (from 0.5% up to 32%, see e.g. [231]), it can be concluded that the reported fault ratios are in the same range.

6.2.3 Overview of the Android data

The left panel of Table 6.2 provides an overview of the number of commits and contributors in each of the releases, assuming a 6 months pre-release period; the right panel shows the number of contributors shared amongst projects. It can be seen that Frameworks/base is the project to which the most commits are made, totalling over 15,000 commits or more than 3,000 per release, while Dalvik, a virtual machine to compile apps to byte code prior to execution, is the most stable. This becomes even more apparent when considering only java source files, see Fig. 6.2, showing that a considerable fraction of source files was changed at least once in Frameworks/base, while, typically, between 10% and 30% of the source files were modified in all other projects. Note that the SDK only has become available after the Eclair release. Fig. 6.2 finally also indicates

[4] http://developer.android.com/index.html

	Eclair		Froyo		Gingerbread		Icecream sandwich		Number of shared contributors			
Projects	*Contr.*	*Commits*	*Contr.*	*Commits*	*Contr.*	*Commits*	*Contr.*	*Commits*	*Dalvik*	*Framew.*	*Libcore*	*SDK*
Dalvik	28	244	24	334	29	528	18	174	**73**	50	36	19
Frameworks	163	4,070	194	3,336	237	4,774	224	4,624	*9.5%*	**506**	73	50
Libcore	20	210	29	497	28	606	21	289	*29%*	*12.3%*	**87**	19
SDK	17	447	18	306	20	446	24	718	*14%*	*8.5%*	*11.2%*	**82**

Table 6.2: Android change activity information

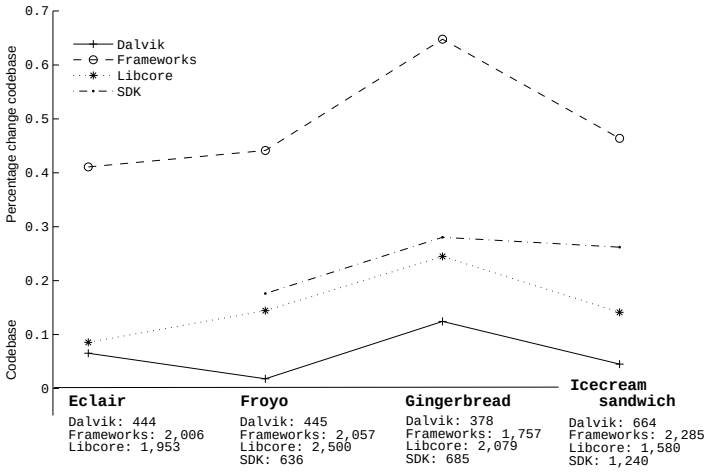


Figure 6.2: Android code base

the total number of source files in each project and for each release. Observe that a minority of source files could not be mined which was typically caused by files being empty or containing syntactic irregularities.

6.3 Empirical setup

6.3.1 Machine Learning techniques

As indicated in Chapter 4, a large number of classification techniques have been adopted in the field of software fault prediction, and also alternatives such as unsupervised and semi-supervised learning have been considered. This case study again considers the techniques presented in Chapter 5 with the exception of LS-SVM, as it includes a computationally very intensive simulated annealing parameter tuning step. On the other hand, the oblique classifier 1 (OC1) and rotation forest, which presents itself as an extension to random forest, are added. Fig. 6.3 provides an overview of the different learners; some exhibit adjustable hyperparameters which are tuned optimizing the AUC using a grid search procedure.

LOC based metrics	*Handle*	*Calculation*
Total number of lines	nl	
LOC_Blank	BLOC	
LOC_Comment	CLOC	
LOC_Excecutable	SLOC	nl-BLOC-CLOC
McCabe metrics		
Cyclomatic_Complexity	$v(G)$	
Cyclomatic_Density	$vd(G)$	
Decision_Density	$dd(G)$	$\frac{\text{Cond\_C}}{\text{Dec\_C}}$
Norm_Cyclomatic_Compl	$Norm\ v(G)$	$\frac{v(G)}{nl}$
Halstead metrics		
Num_Operators	N_1	
Num_Operands	N_2	
Num_Uniq_Operators	n_1	
Num_Uniq_Operands	n_2	
Vocabulary	C	$n_1 + n_2$
Length	N	$N_1 + N_2$
Difficulty	D	$\frac{n_1 \times N_2}{2 \times n_2}$
Level	L	$\frac{1}{\text{D}}$
Volume	V	$\text{N} \times \log_2(n_1 + n_2)$
Programming_Effort	E	$\text{D} \times \text{V}$
Programming_Time	T	$\frac{\text{E}}{18}$
Error_Estimate	B	$\frac{\text{V}}{3000}$
Content	I	$\frac{\text{V}}{\text{D}}$
Miscellaneous metrics		
Branch_Count	Branch_C	
Condition_Count	Cond_C	
Decision_Count	Dec_C	

Table 6.3: Overview of data set attributes

Projects	**Eclair**	**Froyo**	**Ginger-bread**	**Icecream sandwich**
Dalvik	0.2%	0.4%	3.4%	0%
Frameworks	18.5%	36.2%	24.1%	11.9%
Libcore	7.3%	7.5%	40%	2.7%
SDK	*N/A*	4.3%	6%	6%
Obs. count	4,388	5,622	4,883	5,753
Fault ratio	11.7%	17%	26.6%	6.1%

Table 6.4: Overview of fault ratios

Oblique classifier 1

Arguably, univariate decision tree learners are amongst the most popular techniques in software defect prediction thanks to their comprehensibility, see e.g. [173]. However, in Chapter 5, they were found to be outperformed at the 1% level on the MDP$^{\circ}$ data, a finding which was also echoed on the MDP* data, while multivariate linear models boosted better performance. As such, multivariate decision tree algorithms which generate a tree that considers multiple attributes at each split are believed to offer a valuable avenue to reconcile performance and comprehensibility [145]. One example of such multivariate (also referred to as *oblique*) decision tree learner is oblique classifier 1 (OC1). This technique was already introduced in Section 2.4.5, and further details can also be found in Murthy et al. [243].

Rotation forest

Key to the success of random forest is the randomness introduced by the bagging procedure and the repeated selection of a random attribute subset during the construction of the forest of univariate decision trees. Rotation forest combines the idea of pooling a large number of decision trees built on a subset of the attributes and data, with the application of principal component analysis prior to decision tree building, explaining its name. Rotating the axes prior to model building was found to enhance base classifier accuracy at the expense of losing the ability of ranking individual attributes by their importance [268]. Another dissimilarity to random forest lies in the selection of base classifiers, C 4.5 instead of CART, as the first was found to be the better performing of the two, see Table 5.5.

6.3.2 Classifier evaluation

Several metrics have been introduced in the domain of software fault prediction, including those which operate by imposing a specific threshold on the scores outputted by the learners. By adjustment of this threshold, alternative conclusions can be reached on the relative performance of learners, often rendering their use somewhat arbitrary. Table 2.3 already provided an exhaustive overview of classifier evaluation in this domain and hinted at the momentum which is being gained by threshold independent metrics such as the AUC, Alberg diagrams or the H-measure. Throughout this chapter, the AUC is adopted to accommodate this evolution; the AUC has been detailed in Section 2.4.5.

When comparing learners on a single data set, a randomized 10 fold cross validation setup is used, while in case of cross release validation, models are built on all available data of the previous release and validated on the data pertaining to the release under investigation.

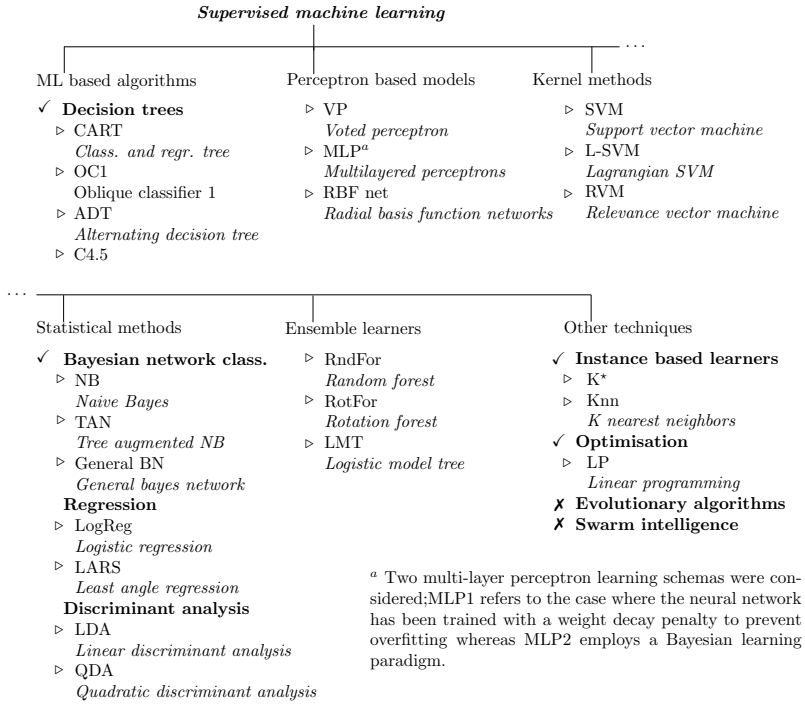


Figure 6.3: Overview of techniques used in this study

6.3.3 Statistical tests

The non parametric framework described in Chapter 5 is again adopted in this study; this framework consists of a Davenport-Iman test to assess whether there are any differences between treatments, paired with a post hoc Rom test comparing each treatment to a control treatment upon rejection of the null hypothesis of equal performance.

Two factors are of interest in this study: the type of classifier and the use of data from a different release. When investigating the type of classifier, the number of treatments k equals 24, while the number of test attempts P equals 4; otherwise, k is two while P equals 24×3. Note that Lehman advised the inequality $k \times P > 30$ to allow for meaningful statistical inferencing, a criterion which is satisfied in both cases [199].

6.4 Results

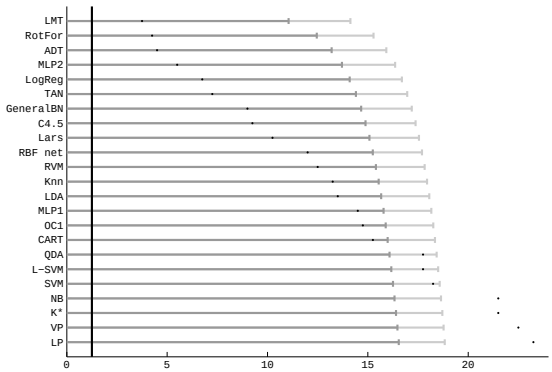


Figure 6.4: Bonferroni-Rom test

Random forest (RndFor) was found to be the best performing technique, closely followed by a number of other techniques, including LMT, RotFor, ADT and MLP2. In a first step, a Davenport-Iman test is performed, finding that the null hypothesis of equal performing techniques is strongly rejected ($p < 10^{-10}$). Thus, we can proceed with the post hoc Rom test which compares the single best performing technique with all other techniques, Fig. 6.4. The average ranks (AR) as calculated by the Davenport-Iman test are given on the horizontal axis while the full vertical line represents the AR of the best technique (RndFor). The AR of other techniques are represented by dots. If a dot is located in the darker (lighter) shaded areas, it is not found to be outperformed by RndFor at 5% (1%). To a large extent, these findings are in line to those presented in Chapter 5. This recurrence adds to our believe that some learners are more fit to serve as an underpinning to software fault prediction models. Note that techniques as

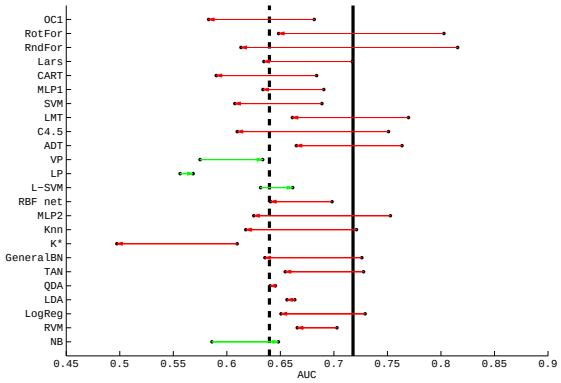


Figure 6.5: Cross release validation

RndFor can result in arbitrary and highly flexible decision boundaries, and as such can be regarded as an incarnation of the idea of local learning put forward by Menzies et al. who states that 'what appears to be useful in a global context is often irrelevant for particular local contexts.' [228].

Fig. 6.5 provides the cross release results; each technique is represented by two dots, giving its median performance when applying randomized 10 fold cross validation and cross release validation. A red (green) arrow is drawn between these points if a drop (increase) in performance is observed with respect to the ordinary cross validation setup. The median across all techniques is given by the full vertical line in case of the cross validation and the dashed line otherwise. The Davenport-Iman test is found to reject the null hypothesis of equal performance at 5% ($p = 2.2 \times 10^{-5}$). While most techniques perform worse, e.g., NB and L-SVM are found to be better performing in an out-of-release setting. Due to the limited number of observations (data sets) per technique, no statistical validation can however be provided on a per technique level. Note that also other research has found that using e.g., cross company (out of universe) and out-of-time models are able provide upfront defect predictions [251,317]; Section 2.3.2 recounts further details hereon.

6.5 Conclusion

In this study we first detailed the extraction process by which we obtained the data drawn upon in the rest of this chapter. Given the similarities in structure across software repositories, we believe the guidelines set forth in this part can serve as an underpinning to other researchers as well.

Next, the Android data was used to investigate the possibility to adopt a cross release validation schema. It is found that, while a significant performance penalty is incurred, such approach is feasible in case of insufficient data; when sufficient data on the current release is available, other approaches are to be preferred. These promising results open up avenues for further research, extending the scope to other software applications.

Intellectuals solve problems, geniuses prevent them.
Albert Einstein, 1879–1955

7 Conclusion

Software has become evermore prevalent over the past 60 years and also its development changed tremendously; it suffices to read Boehm's paper on the evolution of software engineering practices to appreciate this fact [36]. Despite the shift towards more formal development processes observed during the 70s exemplified by the introduction of the waterfall development methodology and formal code checkers, the productivity enhancing changes of the 80s, including 4^{th} generation programming languages and software factories, and the more recent proposals such as agile development methodologies and the service oriented programming paradigm, it is a recurring finding that software development remains troubled by cost overruns and schedule slippage. The bi-annual Chaos reports which provide a current state of affairs of the domain of software development underpin this conjecture and researchers and practitioners joined efforts in addressing these issues. In Belgium, the quintessential failed software project is 'Feniks', which would bring the judicial apparatus to 21^{st} century standards, but a plethora of other failed or challenged projects can be found worldwide. This dissertation discussed two key topics in the field of empirical software engineering, software effort and software fault estimation, in an attempt to persuade the reader of their criticality, offering valuable contributions to the current state of the art in both domains.

7.1 Thesis contributions

Software effort estimation

In a first part, we look into the enigma of software project planning. Despite our ever growing experience in, and the many approaches proposed to estimate total development effort, including expert opinion, formal models, and machine learning, it remained unclear which should be preferred. Academic papers and industry reports alike typically involve only a limited set of techniques on one or a few, sometimes proprietary, data sets and, as is indicated by Kitchenham, 'one of the main problems with evaluating techniques on one or two data sets is that no one can be sure that the specific data sets were not selected because they are the ones that favor the new technique' [177]. Also the usage of non standard

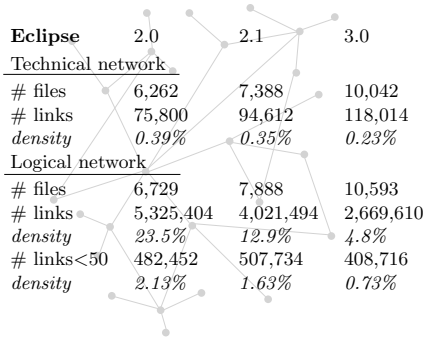
evaluation procedures hampers the generalizability of these studies. In Chapter 3, we try to close this loophole in the literature by reporting our findings of a large scale benchmarking study on 9 data sets collected from public archives and industry partners. We investigated the software effort estimation and machine learning literature, and made an extensive selection of machine learning techniques corresponding to different types of learning behavior. Secondly, we contemplated that comprehensibility could be of key importance when deploying machine learning models in a software production environment. As such, more succinct models were investigated by adopting a generic backward wrapper procedure which effectively removed correlated, noisy or irrelevant attributes.

Software fault estimation

As motivation to our second part, we point out that software verification and validation (V&V) procedures can take up to 60% of the total development budget. Machine learning models can be introduced to pinpoint fault prone regions in the code base, significantly cutting down V&V expenses. Chapter 4 discusses the current state of the literature, and identifies bayesian models, and the naive bayes learner in particular, as one of the most commonplace learners. Other bayesian learners however boost additional advantages and thus deserve our attention. Furthermore, it is conjectured that the development environment should be considered during model selection. One way is to favor a specific score threshold, trading sensitivity to specificity. However, drawing upon the recently introduced H-measure, this can be done more elegantly by specifying a cost distribution.

Software engineering data sets can be difficult and time consuming to collect; e.g. the Cocnasa data set presented in chapter 3 is the result of a million dollar attempt of the NASA to come up with an in-house variant of the cocomo model, but can be summarized in a csv file weighting only kilobytes. As such, many researchers opt to draw upon public data repositories. The commonplace usage of these public data sources comes however at the risk of repository overfitting, where reported improvements are only valid for these specific data sets. Moreover, recent iconoclastic work pointed out several potential shortcomings in these public data sources, questioning prior work. In Chapter 5, we answer the call of Gray et al. [117] concerning the data quality issues found in the NASA MDP data sets by revisiting the influential study of Lessmann et al. [200].

In a final chapter on software fault prediction, Chapter 6, we look into the issue of cross release validation, utilizing data from consecutive releases to learn and validate fault prediction models. Hereto, the historic development archives of the Android platform are mined and analyzed, drawing upon the updated evaluation framework laid out in previous chapters. Additionally, several new and promising techniques are pitched against each other.



Eclipse	2.0	2.1	3.0
Technical network			
# files	6,262	7,388	10,042
# links	75,800	94,612	118,014
density	*0.39%*	*0.35%*	*0.23%*
Logical network			
# files	6,729	7,888	10,593
# links	5,325,404	4,021,494	2,669,610
density	*23.5%*	*12.9%*	*4.8%*
# links<50	482,452	507,734	408,716
density	*2.13%*	*1.63%*	*0.73%*

Figure 7.1: Mining the Eclipse code base for network based learning

7.2 Issues for future research

While several topics were discussed throughout this text, the domain of empirical software engineering can hardly be called quiescent and a multitude of other questions remain to be asked. Providing an exhaustive list of outstanding issues is a difficult, if not impossible, task as one always underestimates the creativity of others and the rapid evolution iconic to the domain of computers[1]. Yet, the following paragraphs discuss a number of topics deemed promising avenues for future research.

A first aspect requiring further inquiry is the aspect of comprehensibility. While addressed to a certain extent in this text, comprehensibility is believed to assuage any preoccupations against the introduction of machine learning in a software development environment. More specifically, given the performance and black box nature of the random forest ensemble learner, methods to counter the opaqueness of such models deserve our attention. A pedagogical rule extraction approach such as ALPA (Active Learning-based Pedagogical Rule Extraction) presents itself as a perfect candidate hereto [167]. Note that while random forest has, to the best of our knowledge, not been adopted to software effort estimation, recent work did discuss ensemble learning with promising results [182]. As such, we foresee also possible applications in this domain.

A second line of thought concerns the adoption of network based learning in the domain of software fault prediction. Traditional machine learning techniques treat observations as being independent, inferring class membership on an observation-by-observation basis. By contrast, network based learning accounts for the possibility that the class membership of one observation depends on the class membership of neighboring observations. Prior research already highlighted the benefits offered by propositionalisation, and adopting a network learning approach such as outlined by Macskassy et al. [213] is a logical next step as higher order effects can be incorporated. During this dissertation, net-

[1]For example, in the 80's, Bill Gates allegedly said that 640 kilobytes of memory ought to be enough for anybody; currently, we count in gigabytes, if not more.

worked data have already been collected using Structure101[2], and can readily be paired with traditional, 'local' data, rendering this research topic more concrete. Specifically, the Eclipse code base, see Fig. 7.1, and the Android platform have already been mined. *Technical networks* refer to networks constructed by tracing calls between classes while *logical networks* are networks in which the presence of edges indicate whether files have been jointly modified during a 6 month time span prior to release. The jointly modification of an exceedingly large number of files might indicate e.g. a code dump, and not refer to the actual development process. Thus, *links<50* refers to the situation in which commits changing more than 50 files are not considered. Network densities are also presented, and it can be seen that our networks are densely connected, especially compared to e.g. networks of telco operators [324].

A final aspect deemed well worth visiting is that of more end-user driven evaluation metrics; statistical evaluation metrics such as the AUC are commonly used, but fail to take the specifics of the fault prediction domain into account. Aspects such as project environment, discussed in Chapter 4, and size of individual software modules should also be taken into consideration, and a first step has been made by Mende et al. [226]. Academia should strive to quantify the value of fault prediction models in business terms, which was also advocated by Briand et al. [47]; examples of such approach in the domain of e.g. telco churn can be found in Verbeke et al. [323].

[2]www.structure101.com

A

Details data selection

Chapter 3 discussed a large scale study revolving around the question of how to accurately estimate software development effort. More specifically, Section 3.4.2 detailed a set of data preprocessing rules; the application hereof on the largest software effort estimation data set, the ISBSG R11 data set, is illustrated below.

Details original data set	*Details preprocessed data set*
# Observations 5,052	# Observations 1,160
# Attributes 115	# Attributes 14

Attributes selection guidelines:

- Only attributes pertaining to software development are retained (i.e. attributes referring to software quality and software productivity are removed).
- Remove attributes counting towards a global attribute such as
  - value adjustment factor,
  - sizing attributes used in obtaining function point count.
- Remove attributes that are unknown at the moment of estimation such as project duration.
- Remove attributes with more than 25% missing values, such as
  - Software process CMMI: 97% missing,
  - 2^{nd} programming language: 96% missing,
  - Package customization: 44% missing.

Observation selection guidelines:

- Projects are retained with an overall data quality rating of A or B. B quality is defined as 'The submission appears to be fundamentally sound but there are some factors which could affect the integrity of the submitted data'.

- Retained projects with function point quality of A. B quality label indicates that 'the unadjusted function point count appears sound, but integrity cannot be assured since a single figure was provided'. This is potentially more problematic than a B rating for overall quality as size attributes are typically the most predictive attributes.
- Retained projects of which the function points are counted using the IFPUG 4 standard, as indicated in the ISBSG guidelines: 'You shouldn't mix pre-IFPUG V4 projects with V4 and post V4 (the sizing changed with that release)'.
- Retained projects of which effort refers to resource level 1 (i.e. only development team effort included).
- Retained projects with no missing value for categorical attribute 'primary programming language' as only 13% of the observations have a missing value for this attribute.

Missing value handling

- Median value imputation for variable 'Team size'. The imputed value is '6'.

Dummy encoding

- Dummy encoding nominal attributes
  - Development type
  - Organization type
  - Business area type
  - Application type
  - Architecture
  - Development technique
  - 1^{st} Database system
  - Platform
  - Language type
  - Primary programming language
  - 1^{st} Hardware
  - 1^{st} Operating system

List of Figures

List of Tables

Bibliography

[1] T. Abdel-Hamid, K. Sengupta, and C. Swett. The impact of goals on software project management: An experimental investigation. *MIS Quarterly*, 23(4):531–555, 1999.

[2] A. Abran and P. Robillard. Function points: A study on their measurement processes and scale transformations. *Journal of Systems and Software*, 25:171–184, 1994.

[3] R. Al Qutaish and A. Abran. An analysis of the design and definitions of Halsteads metrics. In *Proceedings of the 15th International Workshop on Software Measurement*, pages 337–352, 2005.

[4] O. Alan and C. Catal. Thresholds based outlier detection approach for mining class outliers: An empirical case study on software measurement datasets. *Expert Systems with Applications*, 38(4):3440–3445, 2011.

[5] A. Albrecht and J. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6):639–648, 1983.

[6] C. Aliferis, A. Statnikov, I. Tsamardinos, S. Mani, and X. Koutsoukos. Local causal and Markov blanket induction for causal discovery and feature selection for classification part I: Algorithms and empirical evaluation. *Journal of Machine Learning Research*, 11:171–234, 2010.

[7] C. Aliferis, I. Tsamardinos, and A. Statnikov. HITON: a novel Markov blanket algorithm for optimal variable selection. In *AMIA Annual Symposium Proceedings*, 2003.

[8] E. Allwein, R. Schapire, and Y. Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. *Journal of Machine Learning*, 1:113–141, 2000.

[9] E. Altman and H. Rijken. How rating agencies achieve rating stability. *Journal of Banking & Finance*, 28(11):2679–2714, 2004.

[10] R. Andrews, J. Diederich, and A. Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems*, 8(6):373–389, 1995.

[11] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[12] E. Arisholm and L. Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE International symposium on Empirical Software Engineering*, 2006.

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

[14] W. Arthur Jr., W. Bennet Jr., P. A. Edens, and S. T. Bell. Effectiveness of training in organizations: A meta-analysis of design and evaluation features. *Journal of Applied Psychology*, 88(2):627–635, 2003.

[15] I. Askira-Gelman. Knowledge discovery: comprehensibility of the results. In *Proceedings of the* 31^{th} *Annual Hawaii International Conference on System Sciences*, volume 5, pages 247–256, 1998.

[16] M. Auer, A. Trendowicz, B. Graser, E. Haunschmid, and S. Biffl. Optimal project feature selection weigths in analogy-based cost estimation: Improvement and limitations. *IEEE Transactions on Software Engineering*, 32(2):83–92, 2006.

[17] D. Azar and J. Vybihal. An ant colony optimization algorithm to improve software quality prediction models: Case of class stability. *Information and Software Technology*, 53:388–393, 2011.

[18] M. Azzeh, D. Neagu, and P. Cowling. Improving analogy software effort estimation using fuzzy feature subset selection algorithm. In *Proceedings of the* 4^{th} *International workshop on predictor models in software engineering*, pages 71–78, 2008.

[19] A. Bachmann and A. Bernstein. Data retrieval, processing and linking for softare process data analysis. Technical report, University of Zurich, 2009.

[20] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the* 18^{th} *International Symposium on Foundations of Software Engineering*, 2010.

[21] B. Baesens, C. Mues, D. Martens, and J. Vanthienen. 50 years of data mining and OR: upcoming trends and challenges. *Journal of the Operational Research Society*, 60:16–23, 2009.

[22] B. Baesens, R. Setiono, C. Mues, and J. Vanthienen. Using neural network rule extraction and decision tables for credit-risk evaluation. *Management Science*, 49(3):312–329, 2003.

[23] B. Baesens, T. Van Gestel, S. Viaene, M. Stepanova, and J. Vanthienen. Benchmarking state-of-the-art classification algorithms for credit scoring. *Journal of the Operational Research Society*, 54(6):627–635, 2003.

[24] E. Baisch and T.Liedtke. Comparison of conventional approaches and soft-computing approaches for software quality prediction. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 2, pages 1045–1049, 1997.

[25] M. Baojun, K. Dejaeger, J. Vanthienen, and B. Baesens. Software defect prediction based on association rule classification. In *International Conference on Electronic-Business Intelligence*, pages 396–402, 2010.

[26] N. Barakat and A. Bradley. Rule extraction from support vector machines: A review. *Neurocomputing*, 74:178–190, 2010.

[27] N. Barakat and J. Diederich. Learning-based rule-extraction from support vector machines. In *Proceedings of the* 14^{th} *International Conference on Computer Theory and Applications*, 2004.

[28] K. Beck. Extreme programming explained: embrace change. *Addison-Wesley*, 2001.

[29] B. Beizer. *Software testing techniques*. Dreamtech Press, 2002.

[30] I. Benbasat and R. N. Taylor. Behavioral aspects of information processing for the design of management information systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 12(4):439–450, 1982.

[31] J.-Y. Bergeron. Estimation of information systems development efforts: A pilot study. *Information & Management*, 22(4):239–254, 1992.

[32] N. Bettenburg, S. Just, S. Schröter, C. Weiß, R. Premraj, and T. Zimmermann. Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 21–25, 2007.

[33] R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[34] C. Bishop. *Neural networks for pattern recognition*. Oxford University Press, 1995.

[35] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[36] B. Boehm. A view of 20th and 21st century software engineering. In *Proceedings of the* 28^{th} *International Conference on Software Engineering*, pages 12–29, 2006.

[37] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approaches - A survey. *Annals of Software Engineering*, 10(4):177–205, 2000.

[38] B. Boehm and P. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988.

[39] B. Boehm, B. Steece, and R. Madachy. *Software cost estimation with Cocomo II*. Prentice Hall, 2000.

[40] G.E.P. Box and D.R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society, Series B*, 26(211–252), 1964.

[41] A. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

[42] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[43] L. Breiman, J. Friedman, R. Olsen, and C. J. Stone. *Classification And Regression Trees*. Wadsworth & Books/Cole Advanced Books & Software, 1984.

[44] L. Briand, V. Basili, and W. Thomas. A pattern recognition approach for software engineering data analysis. *IEEE Transactions on Software Engineering*, 18(11):931–942, 1992.

[45] L. Briand, K. El Emam, D. Surmann, and I. Wieczorek. An assessment and comparison of common software cost estimation modeling techniques. In *Proceedings of the 21st International Conference on Software Engineering*, pages 313–323, 1999.

[46] L. Briand, T. Langley, and I. Wieczorek. A replicated assessment and comparison of common software cost modeling techniques. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 377–386, 2000.

[47] L. Briand, W. Melo, and J. Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, 28(7):706–720, 2002.

[48] G. Brier. Verification of forecasts expressed in terms of probability. *Monthly Weather Review*, 78(1):1–3, 1950.

[49] C. Burgess and M. Lefley. Can genetic programming improve software effort estimation? A comparative evaluation. *Information and Software Technology*, 43:863–873, 2001.

[50] Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes. Ensemble selection from libraries of models. In *Proceedings of the* 21st *International Conference on Machine Learning*, 2004.

[51] E. Castillo, J. Gutiérrez, and A. Hadi. *Expert systems and probabilistic network models*. Springer Verlag, 1997.

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

[53] C. Catal and B. Diri. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. *Information Sciences*, 179(8):1040–1058, 2009.

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

[55] C. Catal, U. Sevim, and B. Diri. Practical development of an Eclipse-based software fault prediction tool using Naive Bayes algorithm. *Expert Systems with Applications*, 38:2347–2353, 2011.

[56] J. Catlett. Tailoring rulesets to misclassification costs. In *Proceedings of the 1995 Conference on AI and Statistics*, pages 88–94, 1995.

[57] Z. Chen, T. Menzies, D. Port, and B. Boehm. Feature subset selection can improve software cost estimation accuracy. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–6, 2005.

[58] J. Cheng, R. Greiner, J. Kelly, D. Bell, and W. Liu. Learning Bayesian networks from data: An information-theory based approach. *Artificial Intelligence*, 137:43–90, 2002.

[59] D. Chickering. Optimal structure identification with greedy search. *Journal of Machine Learning Research*, 3:507–554, 2002.

[60] D. Chickering, C. Meek, and D. Heckerman. Large-sample learning of Bayesian Networks is NP-Hard. *Journal of Machine Learning Research*, 5:1287–1330, 2004.

[61] S. Chidamber and C. Kemerer. A metrics suite for Object-Oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[62] N.-H. Chiu and S.-J. Huang. The adjusted analogy-based software effort estimation based on similarity distances. *Journal of Systems and Software*, 80:628–640, 2007.

[63] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, 14(3):462–467, 1968.

[64] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software engineering metrics and models*. The Benjamin/Cummings Publishing Company, Inc, 1986.

[65] G. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.

[66] D. Čubranić. Automatic bug triage using text categorization. In *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering*, 2004.

[67] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 433–436, 2007.

[68] R. Dawes, D. Faust, and P. Meehl. Clinical versus actuarial judgement. *Science*, 243(4899):1668–1674, 1989.

[69] M. De Almeida and S. Matwin. Machine learning method for software quality model building. *Lecture notes in computer science*, pages 565–573, 1999.

[70] A. De Carvalho, A. Pozo, and S. Vergilio. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *Journal of Systems and Software*, 83(5):868–882, 2010.

[71] L. De Rore. *Measuring productivity and improving efficiency in software development environments*. PhD thesis, K.U.Leuven, 2009.

[72] R. De Veaux and D. Hand. How to lie with bad data. *Statistical Science*, 20(3):231–238, 2005.

[73] K. Dejaeger, F. Goethals, A. Giangreco, L. Mola, and B. Baesens. Gaining insight into student satisfaction using comprehensible data mining techniques. *European Journal of Operational Research*, 218(2):548–562, 2012.

[74] K. Dejaeger, S. Lessmann, M. Shepperd, and B. Baesens. Benchmarking classification models for software defect prediction: Revisiting earlier results. *IEEE Transactions on Software Engineering*, In submission, 2012.

[75] K. Dejaeger, W. Verbeke, D. Martens, and B. Baesens. Data mining techniques for software effort estimation: a comparative study. *IEEE Transactions on Software Engineering*, 38(2):375–397, 2012.

[76] K. Dejaeger, T. Verbraken, and B. Baesens. Towards comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering*, In press, 2012.

[77] J. Demšar. Statistical comparison of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[78] J. M. Desharnais. *Analyse statistique de la productivities des projets de developpement en informatique apartir de la techniques des points de fonction*. PhD thesis, University du Quebec, Quebec, Canada, 1988.

[79] T. Dietterich and G. Bakiri. Solving multiclass learning via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[80] P. Domingos. The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3(4):409–425, 1999.

[81] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29:103–130, 1997.

[82] M. Dorigo. Editorial on swarm intelligence. *Swarm intelligence*, 1(1):1–2, 2007.

[83] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley, New York, 1973.

[84] O. J. Dunn. Multiple comparisons among means. *Journal of American Statistical Association*, 56:52–64, 1961.

[85] C. Ebert and R. Dumke. *Software Measurement: Establish-Extract-Evaluate-Execute*. Springer, 2007.

[86] J. Egan. *Signal detection theory and ROC Analysis*. Academic Press New York, 1975.

[87] J. Ekanayake, J. Tappolet, H. Gall, and A. Bernstein. Tracking concept drift of software projects using defect prediction quality. In *6th IEEE International Working Conference on Mining Software Repositories*, pages 51–60, 2009.

[88] K. Elish and M. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[89] J. Elith and J. Leathwick. Predicting species distributions from museum and herbarium records using multiresponse models fitted with multivariate adaptive regression splines. *Diversity and Distributions*, 13(3):265–275, 2007.

[90] L. Eveleens and C. Verhoef. The rise and fall of the Chaos report figures. *IEEE Software*, 27(1):30–36, 2010.

[91] M. Evett, T. Khoshgoftaar, P. Chien, and E. Allen. GP-based software quality prediction. In *Proceedings of the 3rd Annual Conference on Genetic Programming*, pages 60–65, 1999.

[92] T. Fawcett. An introduction to ROC analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[93] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the International Joint Conference on Uncertainty in AI*, pages 1022–1027, 1993.

[94] N. Fenton and M. Neil. Software metrics: Successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999.

[95] N. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 2002.

[96] N. Fenton and S. Pfleeger. *Software metrics: A rigorous & practical approach*. PWS Publishing Company, 1998.

[97] C. Ferri, J. Hernández-Orallo, and M. Salido. Volume under the ROC surface for multi-class problems. In *Proceedings of the 14th European Conference on Machine Learning*, pages 108–120, 2003.

[98] G. Finnie, G. Wittig, and J-M. Desharnais. A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models. *Journal of Systems and Software*, 39:281–289, 1997.

[99] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, 2003.

[100] P. Flach, J. Hernández-Orallo, and C. Ferri. A coherent interpretation of AUC as a measure of aggregated classification performance. In *Proceedings of the 28th International Conference on Machine Learning*, 2011.

[101] T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit. A simulation study of the model evaluation criterion MMRE. *IEEE Transactions on Software Engineering*, 29(11):985–995, 2003.

[102] Y. Freund and R. Schapire. A decision-theoretic generalisation of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[103] M. Friedl and C. Brodley. Decision tree classification of land cover from remotely sensed data. *Remote Sensing of Environment*, 61:399–409, 1997.

[104] J. Friedman. Multivariate adaptive regression splines. *Annals of Statistics*, 19(1):1–67, 1991.

[105] M. Friedman. A comparison of alternative tests of significance for the problem of m rankings. *Annals of Mathematical Statistics*, 11:86–92, 1940.

[106] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.

[107] G. Fung, S. Sandilya, and R. Rao. Rule extraction from linear support vector machines. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 32–40, 2005.

[108] S. García, A. Fernández, J. Luengo, and F. Herrera. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information Sciences*, 180(10):2044–2064, 2010.

[109] A. Giangreco, A. Carugati, and A. Sebastiano. Are we doing the right thing? Food for thoughts on training evaluation and its context. *Personnel Review*, 39(2):162–177, 2010.

[110] A. Giangreco, A. Sebastiano, and R. Peccei. Trainees' reactions to training: An analysis of the factors affecting overall satisfaction with training. *The International Journal of Human Resources Management*, 20(1):96–111, 2009.

[111] S. Goedertier, J. De Weerdt, D. Martens, J. Vanthienen, and B. Baesens. Process discovery in event logs: An application in the telecom industry. *Applied Soft Computing*, 11(2):1697–1710, 2011.

[112] S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on Dependable and Secure Computing*, 4(1):32–40, 2001.

[113] I. Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81:186–195, 2008.

[114] J. González-Barahona and G. Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, In press, 2012.

[115] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.

[116] C. Graham, J. Correia, F. Biscotti, and M. Cheung. Forecast: enterprize software markets, 2Q11 update. Technical report, Gartner, 2010.

[117] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson. The misuse of the NASA metrics data program data sets for automated software defect prediction. In *15^{th} Annual Conference on Evaluation & Assessment in Software Engineering*, pages 96–103, 2011.

[118] L. Guo, Y. Ma, B. Cukic, and H. Singh. Robust prediction of fault-proness by random forests. In *Proceedings of the 15^{th} International Symposium on Software Reliability Engineering*, 2004.

[119] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2006.

[120] M. Hagan and M. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6):989–993, 1994.

[121] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, In press, 2012.

[122] M. Halstead. *Elements of software science*. Elsevier, 1977.

[123] F. Hampel. A general qualitative definition of robustness. *Annals of Mathematical Statistics*, 42:1887–1896, 1971.

[124] H. Han, L. Giles, H. Zha, C. Li, and K. Tsioutsiouliklis. Two supervised learning approaches for name disambiguation in author citations. In *Proceedings of JCDL*, 2004.

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

[126] D. Hand. Evaluating diagnostic tests: The area under the ROC curve and the balance of errors. *Statistics in Medicine*, 29(14):1502–1510, 2010.

[127] D. Hand and R. Till. A simple generalisation of the Area Under the ROC Curve for multiple class classification problems. *Machine Learning*, 45:171–186, 2001.

[128] D. Hand and K. Yu. Idiot's Bayes–Not so stupid after all? *International Statistical Review*, 69(3):385–398, 2001.

[129] J. Hanley and B. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143:29–36, 1982.

[130] M. Harrold. Testing: a roadmap. In *Proceedings of the conference on the future of software engineering*, pages 61–72, 2000.

[131] A. Hars and S. Ou. Working for free? - Motivations of participating in open source projects. In *Proceedings of the 34^{th} Hawaii International Conference on System Sciences*, 2001.

[132] T. Hastie and R. Tibshirani. Classification by pairwise coupling. *The Annals of Statistics*, 26(2):451–471, 1998.

[133] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction (2^{nd} edition)*. Springer Science, 2009.

[134] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19:167–199, 2012.

[135] D. Hecker. Occupational employment projections to 2014. *Monthly Laber Review*, 128:70–101, 2005.

[136] D. Heckerman, D. Geiger, and D. Chickering. Learning bayesian networks: The combination of knowledge and statitical data. *Machine Learning*, 20:194–243, 1995.

[137] F. J. Heemstra and R. J. Kusters. Function point analysis: evaluation of a software cost estimation model. *European Journal of Information Systems*, 1(4):223–237, 1991.

[138] A. Heiat. Comparison of artificial neural networks and regression models for estimating software development effort. *Information and Software Technology*, 44(15):911–922, 2002.

[139] J. Hernández-Orallo, P. Flach, and C. Ferri. Threshold choice methods: the missing link. *Arxiv preprint arXiv:1112.2640*, 2012.

[140] A. E. Hoerl. Application of ridge analysis to regression problems. *Chemical Engineering Progress*, 58(5):54–59, 1962.

[141] P. W. Holland. Robust regression using iteratively reweighted least squares. *Communications in Statistics: Theory and Methods*, 6(9):813–827, 1977.

[142] R. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine learning*, 11(1):63–90, 1993.

[143] S.-J. Huang, N.-H. Chiu, and L.-W. Chen. Integration of the grey relational analysis with genetic algorithm for software estimation. *European Journal of Operational Research*, 188:898–909, 2008.

[144] J. Hudepohl, S. Aud, T. Khoshgoftaar, E. Allen, and J. Mayrand. EMERALD: Software metrics on the desktop. *IEEE Software*, 13(5):56–60, 1996.

[145] J. Huysmans, K. Dejaeger, C. Mues, J. Vanthienen, and B. Baesens. An empirical evaluation of the comprehensibility of decision table, tree and rule based predictive models. *Decision Support Systems*, 51(1):141–154, 2011.

[146] A. Idri, T. M. Khoshgoftaar, and A. Abran. Can neural networks be easily interpreted in software cost estimation ? In *Proceedings of the IEEE International Conference on Fuzzy Systems*, pages 1162–1167, 2002.

[147] A. Idri, A. Zahi, E. Mendes, and A. Zakrani. Software Cost Estimation Models Using Radial Basis Function Neural Networks. In *Lecture Notes in Computer Science*, volume 4895, pages 21–31, 2008.

[148] International Software Benchmarking Group Limited. ISBSG data release R11. Technical report, www.ISBSG.org, 2010.

[149] R. Jeffery, M. Ruhe, and I. Wieczorek. A comparative study of two software development cost modeling techniques using multi-organizational and company-specific data. *Information and Software Technology*, 42(14):1009–1016, 2000.

[150] R. Jeffery, M. Ruhe, and I. Wieczorek. Using public domain metrics to estimate software development effort. In *Proceedings of the 7^{th} International Software Metrics Symposium*, pages 16–27, 2001.

[151] J. Jenkins and A. Milton. Empirical investigation of systems development practices and results. *Information & Management*, 7(2):73–82, 1984.

[152] L. Jensen and A. Bateman. The rise and fall of supervised machine learning techniques. *Bioinformatics*, 27(24):3331–3332, 2011.

[153] Y. Jiang and B. Cukic. Misclassification cost-sensitive fault prediction models. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, 2009.

[154] Y. Jiang, B. Cukic, and Y. Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13:561–595, 2008.

[155] Y. Jiang, B. Cukic, and T. Menzies. Fault prediction using early lifecycle data. In *The 18th IEEE International Symposium on Software Reliability*, pages 237–246, 2007.

[156] Y. Jiang, B. Cukic, and T. Menzies. Cost curve evaluation of fault prediction models. In *The 19th IEEE International Symposium on Software Reliability*, pages 197–206, 2008.

[157] Y. Jiang, B. Cukic, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the International Conference on Software Engineering, Promise workshop*, 2008.

[158] G. John and P. Langley. Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 338–345, Montreal, Québec, Canada, 1995. Morgan Kaufmann.

[159] C. Jones. *Programming productivity*. McGraw-Hill New York, 1986.

[160] M. Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70:37–60, 2004.

[161] M. Jørgensen. Forecasting of software development work effort: Evidence on expert judgement and formal models. *International Journal of Forecasting*, 23:449–462, 2007.

[162] M. Jørgensen and K. Moløkken-Østvold. How large are software cost overruns? A review of the 1994 CHAOS report. *Information and Software Technology*, 48:297–301, 2006.

[163] M. Jørgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007.

[164] M. Jørgensen and D. I. K. Søberg. Impact of experience on maintenance skills. *Journal of Software Maintenance and Evolution: Research and practice*, 14(2):123–146, 2002.

[165] M. Jørgensen and S. Wallace. Improving project cost estimation by taking into account managerial flexibility. *European Journal of Operational Research*, 127:239–251, 2000.

[166] N. Jørgensen. Putting it all in the trunk: incremental software development in the FreeBSD open source project. *Information Systems Journal*, 11(4):321–336, 2001.

[167] E. Junqué de Fortuny and D. Martens. Active learning-based pedagogical rule extraction. *IEEE Transactions on Data and Knowledge*, In submission, 2012.

[168] T. Kamiya, S. Kusumoto, and K. Inoue. Prediction of fault-proneness at early phase in Object-Oriented development. In *Proceedings of the* 2^{nd} *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1999.

[169] S. Kanmani, V. Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object-oriented software fault prediction using neural networks. *Information and Software Technology*, 49:483–492, 2007.

[170] G. Kass. An exploratory technique for investigating large quantities of categorical data. *Applied statistics*, pages 119–127, 1980.

[171] R. Kaufman, J. Keller, and R. Watkins. What works and what doesn't: Evaluation beyond Kirkpatrick. *Performance & Instructions*, 35(2):8–12, 1995.

[172] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the International Conference on Neural Networks*, pages 1942–1948, 1995.

[173] T. Khoshgoftaar, E. Allen, and J. Deng. Using regression trees to classify fault-prone software modules. *IEEE Transactions on Reliability*, 51(4):455–462, 2002.

[174] T. Khoshgoftaar and N. Seliya. Tree-based software quality estimation models for fault prediction. In *Proceedings of the* 8^{th} *IEEE Symposium on Software Metrics*, pages 203–214, 2002.

[175] C. Kirsopp and M. Shepperd. Making inferences with small numbers of training sets. In *IEE Proceedings Software*, volume 149, pages 123–130, 2002.

[176] B. Kitchenham. A procedure for analyzing unbalanced data sets. *IEEE Transactions on Software Engineering*, 24(4):278–301, 1998.

[177] B. Kitchenham and E. Mendes. Why comparative effort prediction studies may be invalid. In *Proceedings of the* 5^{th} *International Conference on Predictor Models in Software Engineering*, 2009.

[178] B. Kitchenham, E. Mendes, and G. Travasoss. Cross versus within company cost estitmation studies: A systematic review. *IEEE Transactions on Software Engineering*, 33(5):316–329, 2007.

[179] B. Kitchenham, L.M. Pickard, S.G. MacDonell, and M. Shepperd. What accuracy statistics really measure. *IEE Proceedings Software*, 148(3):81–85, 2001.

[180] E. Kocaguneli, T. Menzies, A. Bener, and J. Keung. Exploiting the essential assumptions of analogy-based effort estimation. *IEEE Transactions on Software Engineering*, In press, 2012.

[181] E. Kocaguneli, T. Menzies, and J. Keung. Kernel methods for software effort estimation. *Empirical Software Engineering*, In press, 2012.

[182] E. Kocaguneli, T. Menzies, and J. Keung. On the value of ensemble effort estimation. *IEEE Transactions on Software Engineering*, In press, 2012.

[183] E. Kocaguneli, A. Tosun, B. Turhan, and B. Caglayan. Prest: an intelligent software metrics extraction, analysis and defect prediction tool. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, 2009.

[184] E. Kocaguneli, Y. Yang, and J. Keung. When to use data from other projects for effort estimation. In *Proceedings of the 25^{th} International Conference on Automated Software Engineering*, 2010.

[185] S. Koch and J. Mitlöhner. Software project effort estimation with voting rules. *Decision Support Systems*, 46:895–901, 2009.

[186] R. Kohavi. A study on cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the* 14^{th} *International Joint Conference on Artificial Intelligence*, pages 1137–1145, 1995.

[187] A. Kołcz, A. Chowdhury, and J. Alspector. Data duplication: An imbalance problem? In *Workshop on Learning form Imbalanced Datasets II*, 2003.

[188] I. Kononenko. Semi-naive bayesian classifier. In *Proceedings 6^{th} European Working Session on Learning*, pages 206–219. Berlin: Springer Verlag, 1991.

[189] S. Kotsiantis, S. Zaharakis, and P. Pintelas. Supervised machine learning: A review of classification techniques. *Informatica*, 31:249–268, 2007.

[190] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[191] P. Kruchten. *The Rational Unified Process - An Introduction (2^{nd} ed.)*. Addison-Wesley, 2000.

[192] V. Kumar, V. Ravi, M. Carr, and R. Kiran. Software development cost estimation using wavelet neural networks. *Journal of Systems and Software*, 81:1853–1867, 2008.

[193] P. Langley and S. Sage. Induction of selective bayesian classifiers. In *Proceedings of the 10^{th} Conference on Uncertainty in Artificial Intelligence*, pages 399–406, 1994.

[194] D. Lapedes. *McGraw-Hill dictionary of scientific and technical terms*. McGraw-Hill Book Company, 1978.

[195] P. Larranaga, C. Kuijpers, R. Murga, and Y. Yurramendi. Learning bayesian network structures by searching for the best ordering with genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 26(4):487–493, 1996.

[196] S. Lee and J. Pershing. Dimensions and design criteria for developing training reactions evaluations. *Human Resources Development International*, 5(2):175–197, 2002.

[197] T.-S. Lee and I-F. Chen. A two-stage hybrid credit scoring model using artificial neural networks and multivariate adaptive regression splines. *Expert Systems with Applications*, 28:743–752, 2005.

[198] M. Lefley and M. Shepperd. Using genetic programming to improve software effort estimation based on general data sets. In *Lecture Notes in Computer Science*, volume 2724, pages 2477–2487, 2003.

[199] E. Lehman and H. D'Abrera. *Nonparametrics-statistical methods based on ranks*. Holden-Day San Francisco, 1975.

[200] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[201] S. Lessmann and S. Voß. Customer-centric decision support. *Business & Information Systems Engineering*, 2(2):79–93, 2010.

[202] H. Li and W. Cheung. An empirical study of software metrics. *IEEE Transactions on Software Engineering*, 13(6):697–708, 1987.

[203] J. Li and G. Ruhe. A comparative study of attribute weighting heuristics for effort estimation by analogy. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering (ISESE'06)*, 2006.

[204] J. Li, G. Ruhe, A Ak-Emran, and M Richter. A flexible method for software effort estimation by analogy. *Empirical Software Engineering*, 12:65–107, 2007.

[205] Y. Li, M. Xie, and T. Goh. A study of mutual information based feature selection for case based reasoning in software cost estimation. *Expert Systems with Applications*, 36:5921–5931, 2009.

[206] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315, 2005.

[207] W. Liqun, L. Jincheng, X. Wei, and L. Hao. China statistical yearbook. Technical report, National Bureau of Statistics of China, 2010.

[208] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity a review. *ACM Computing Surveys*, 33(2):177–208, 2001.

[209] Y. Liu, T. Khoshgoftaar, and N. Seliya. Evolutionary optimization of software quality modeling with multiple repositories. *IEEE Transactions on Software Engineering*, 36(6):852–864, 2010.

[210] L. Long, C. Dubois, and R. Faley. Online training: the value of capturing trainee reactions. *Journal of Workplace Learning*, 20(1):21–37, 2008.

[211] P. Louis, E. Van Laere, and B. Baesens. Motivating and predicting bank rating transitions using optimal survival analysis models. In *Proceedings of the 24^{th} Australasian Finance & Banking Conference*, 2011.

[212] S. MacDonell and M. Shepperd. Combining techniques to optimize effort predictions in software project management. *Journal of Systems and Software*, 66:91–98, 2003.

[213] S. Macskassy and F. Provost. Classification in networked data: A toolkit and a univariate case study. *Journal of Machine Learning Research*, 8:935–983, 2007.

[214] Jureczko Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6^{th} International Conference on Predictive Models in Software Engineering*, 2010.

[215] S. Mahmood, R. Lai, Y. Soo Kim, J. Hong Kim, S. Cheon Park, and H. Suk Oh. A survey of component based system quality assurance and assessment. *Information and Software Technology*, 47(10):693–707, 2005.

[216] C. Mair, M. Shepperd, and M. Jørgensen. An analysis of datasets used to train and validate cost prediction systems. *ACM SIGSOFT Software Engineering Notes*, 4:1–6, 2005.

[217] J. Maletic and A. Marcus. Data cleansing: Beyond integrity analysis. In *Proceedings of the 5^{th} International Conference on Information Quality*, pages 200–209, 2000.

[218] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[219] D. Martens, B. Baesens, and T. Fawcett. Editorial survey: swarm intelligence for data mining. *Machine learning*, 82(1):1–42, 2011.

[220] D. Martens, B. Baesens, T. Van Gestel, and J. Vanthienen. Comprehensible credit scoring models using rule extraction from support vector machines. *European Journal of Operational Research*, 183(3):1466–1476, 2007.

[221] D. Martens, M. De Backer, R. Haesen, B. Baesens, C. Mues, and J. Vanthienen. Ant based approach to the knowledge fusion problem. In *Ant Colony Optimisation and Swarm Intelligence, 5^{th} International Workshop, ANTS 2006*, volume 4150, pages 84–95, 2006.

[222] D. Martens, M. De Backer, R. Haesen, M. Snoeck, J. Vanthienen, and B. Baesens. Classification with ant colony optimization. *IEEE Transactions on Evolutionary Computing*, 11(5):651–665, 2007.

[223] K. Maxwell. *Applied statistics for software managers*. Prentice-Hall, New York, 2000.

[224] K. Maxwell, L. Van Wassenhove, and S. Dutta. Software development productivity of European space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22(10):706–718, 1996.

[225] K. Maxwell, L. Van Wassenhove, and S. Dutta. Performance evaluation of general and company specific models in software development effort estimation. *Management Science*, 45:787–803, 1999.

[226] T. Mende and R. Koschke. Effort-aware defect prediction models. In *Proceedings of the 14^{th} European Conference on Software Maintenance and Reengineering*, 2010.

[227] E. Mendes, I. Watson, C. Triggs, N. Mosley, and S. Counsell. A comparative study of cost estimation models for web hypermedia applications. *Empirical Software Engineering*, 8:163–196, 2003.

[228] T. Menzies, A. Butcher, A. Marcus, and T. Zimmermann D. Cok. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 26^{th} International Conference on Automated Software Engineering*, pages 343–351, 2011.

[229] T. Menzies, Z. Chen, J. Hihn, and K. Lum. Selecting best practices for effort estimation. *IEEE Transactions on Software Engineering*, 32(11):883–895, 2006.

[230] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 32(11):2–13, 2007.

[231] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[232] T. Menzies and M. Shepperd. Special issue on repeatable results in software engineering prediction, editorial. *Empirical Software Engineering*, 17:1–17, 2012.

[233] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the International Conference on Software Engineering, Promise workshop*, 2008.

[234] R. Michalski. A theory and methodology of inductive learning. *Artificial intelligence*, 20(2):111–161, 1983.

[235] H.-T. Moges, K. Dejaeger, W. Lemahieu, and B. Baesens. A multidimensional analysis of data quality for credit risk management: New insights and challenges. *Information and Management*, Under review, 2012.

[236] H.-T. Moges, K. Dejaeger, W. Lemahieu, and B. Baesens. A total data quality management for credit risk: new insights and challenges. *International Journal of Information Quality*, In press, 2012.

[237] C. Moler. *Numerical Computing with Matlab*, chapter 5, pages 1–27. Society for Industrial and Applied Mathematics, 2004.

[238] K. Moløkken-Østvold, M. Jørgensen, S. Tanilkan, H. Gallis, A. Lien, and W. Hove. A survey on software estimation in the Norwegian industry. In *Proceedings of the 10^{th} International Symposium on Software Metrics*, pages 208–219, 2004.

[239] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10^{th} International Conference on Artificial Intelligence: Methodology, Systems, Applications*, pages 41–50, 2002.

[240] J. Moody and C. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computing*, 1:281–294, 1989.

[241] D. Mossman. Three-way ROCs. *Medical Decision Making*, 19:78–89, 1999.

[242] T. Mukhopadhyay, S. S. Vicinanza, and M. J. Prietula. Examining the feasibility of a case-based reasoning model for software effort estimation. *MIS Quarterly*, 16(2):155–171, 1992.

[243] S. Murthy, S. Kasif, and S. Salzberg. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2:1–32, 1994.

[244] I. Myrtveit and E. Stensrud. A controlled experiment to assess the benefits of estimation with analogy and regression models. *IEEE Transactions on Software Engineering*, 25(4):510–525, 1999.

[245] I. Myrtveit, E. Stensrud, and M. Shepperd. Reliability and validity in comparative studies of software prediction models. *IEEE Transactions on Software Engineering*, 31(5):380–391, 2005.

[246] P. Nemenyi. *Distribution-free multiple comparisons*. PhD thesis, Princeton University, 1963.

[247] OECD. Education at a glance 2010: OECD indicators. Technical report, www.OECD.org, 2010.

[248] S. Olejnik, J. Li, S. Supattathum, and C. Huberty. Multiple testing and statistical power with modified bonferroni procedures. *Journal of Educational and Behavioral Statistics*, 22(4):389–406, 1997.

[249] G. Ooi and C. Soh. Advances in information systems. *Developing an activity-based costing approach for system development and implementation*, 34(3):54–71, 2003.

[250] T. Ostrand, E. Weyuker, and R. Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96, 2004.

[251] T. Ostrand, E. Weyuker, and R. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(5):340–355, 2005.

[252] G. Pai and J. Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Transactions on Software Engineering*, 33(10):675–686, 2007.

[253] H. Park and S. Baek. An empirical validation of a neural network model for software effort estimation. *Expert Systems with Applications*, 35:929–937, 2008.

[254] J. Pearl. *Probabilistic reasoning in Intelligent Systems: networks for plausible inference*. Morgan Kaufmann, 1988.

[255] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8):1226–1238, 2005.

[256] D. Phan, D. Vogel, and J. Nunamaker. The search for perfect project management. *Computerworld*, 22(39):95–100, 1988.

[257] R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization: an overview. *Swarm intelligence*, 1(1):33–57, 2007.

[258] D. Port and M. Korte. Comparative Studies of the Model Evaluation Criterions MMRE and PRED in Software Cost Estimation Research. In *Proceedings of the 2^{nd} ACM-IEEE International symposium on Empirical software engineering and measurement*, pages 51–60, 2008.

[259] T. Prifti, S. Banerjee, and B. Cukic. Detecting bug duplicate reports through local references. In *Proceedings of the 7^{th} International Conference on Predictive Models in Software Engineering*, 2011.

[260] F. Provost, T. Fawcett, and R. Kohavi. The case against accuracy estimation for comparing induction algorithms. In *Proceedings of the 15th International Conference on Machine Learning*, 1998.

[261] L. Putnam. A general empirical solution to the macro software sizing and estimation problem. *IEEE Transactions on Software Engineering*, 4(4):345–381, 1978.

[262] T.-S. Quah and M. Thwin. Application of neural networks for software quality prediction using object-oriented metrics. In *Proceedings of the International Conference on Software Maintenance*, 2003.

[263] J. Quinlan. Learning with continuous classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992.

[264] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 2003.

[265] C. Rasmussen. Gaussian processes in machine learning. *Advanced Lectures on Machine Learning*, pages 63–71, 2004.

[266] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[267] J. Roberts, I. Hann, and S. Slaughter. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the Apache projects. *Management science*, 52(7):984–999, 2006.

[268] J. Rodríguez and L. Kuncheva. Rotation forest: A new classifier ensemble method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10):1619–1630, 2006.

[269] P. Rousseeuw. Least median of squares regression. *Journal of the American Statistical Association*, 79(388):871–880, 1984.

[270] W. Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, pages 1–9, 1970.

[271] J.P. Sacha. *New synthesis of Bayesian network classifiers and cardiac SPECT image interpretation*. PhD thesis, University of Toledo, 1999.

[272] R. Sakia. The box-cox transformation technique: A review. *Journal of the Royal Statistical Society. Series D*, 41(2):169–178, 1992.

[273] L. Sargent, B. Allen, J. Frahm, and G. Morris. Enhancing the experience of student teams in large classes. *Journal of Management Education*, 33(5):526–552, 2009.

[274] N. Sebe, M. Lew, Y. Sun, I. Cohen, T. Gevers, and T. Huang. Authentic facial expression analysis. *Image and Vision Computing*, 25(12):1856–1863, 2007.

[275] P. Sentas, L. Angelis, I. Stamelos, and G. Bleris. Software productivity and effort prediction with ordinal regression. *Information and Software Technology*, 47:17–29, 2005.

[276] A. Seret, T. Verbraken, S. Versailles, and Bart Baesens. A new SOM-based method for profile generation: Theory and an application in direct marketing. *European Journal of Operational Research*, 220(1):199–209, 2012.

[277] V. Sessions and M. Valtorta. Towards a method for data accuracy assessment utilizing a Bayesian network learning algorithm. *Journal of Data and Information Quality*, 1(3):1–34, 2009.

[278] R. Setiono, K. Dejaeger, W. Verbeke, D. Martens, and B. Baesens. Software effort prediction using regression rule extraction from neural networks. In *Proceedings of the 22nd International Conference on Tools with Artificial Intelligence*, pages 45–52, 2010.

[279] C. Shannon and W. Weaver. *The mathematical theory of communication*. Urbana, University of Illinois Press, 1949.

[280] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[281] M. Shepperd and D. Ince. A critique of three metrics. *Journal of Systems and Software*, 26(3):197–210, 1994.

[282] M. Shepperd and G. Kadoda. Using simulation to evaluate prediction techniques. In *Proceedings of the 7th International Software Metrics Symposium*, pages 349–359, 2002.

[283] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(12):736–743, 1997.

[284] M. Shepperd, C. Schofield, and B. Kitchenham. Effort estimation using analogies. In *Proceedings of the 18th International conference on Software Engineering*, 1996.

[285] M. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the nasa software defect data sets. *IEEE Transactions on Software Engineering*, Under review, 2012.

[286] S. Sherer. Software fault prediction. *Journal of Systems and Software*, 29(2):97–105, 1995.

[287] E. Shihab, Y. Kamei, and P. Bhattacharya. The 9^{th} working conference on mining software repositories. In *Mining Challenge 2012: The Android Platform*, 2012.

[288] J. Shirabad and T. Menzies. *The PROMISE Repository of Software Engineering Databases*. University of Ottawa, 2005.

[289] F. Shull, V. Basili, B. Boehm A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Proceedings of the 8^{th} IEEE Symposium on Software Metrics*, pages 249–258, 2002.

[290] Y. Singh, A. Kaur, and R. Malhotra. Empirical validation of object-oriented metrics for predicting fault proneness models. *Software Quality Journal*, 18(1):3–35, 2010.

[291] A. Sinha and H. Zhao. Incorporating domain knowledge into data mining classifiers: An application in indirect lending. *Decision Support Systems*, 46(1):287–299, 2008.

[292] V. Sinha, S. Mani, and M. Gupta. Mince: Mining change history of android project. In *9^{th} IEEE Working Conference on Mining Software Repositories*, pages 132–135, 2012.

[293] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37(3):356–370, 2011.

[294] D. Specht. A general regression neural network. *IEEE Transactions on Neural Networks*, 2(6):568–576, 1991.

[295] P. Spirtes, C. Glymour, and R. Scheines. *Causation, prediction, and search*. The MIT Press, second edition edition, 2000.

[296] K. Srinivasan and D. Fisher. Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering*, 21(2):126–137, 1995.

[297] K. Strike, K. El Emam, and N. Madhavji. Software cost estimation with incomplete data. *IEEE Transactions on Software Engineering*, 27(10):890–908, 2001.

[298] T. Stützle and H. Hoos. $max - min$ ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[299] J. A. K. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural Processing Letters*, 9(3):293–300, 1999.

[300] C. Symons. Interview with Charles Symons. *Computer aid*, 2006.

[301] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson Addison Wesley Boston, 2006.

[302] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. Arnetminer: Extraction and mining of academic social networks. In *Proceeding of the 14^{th} ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, pages 990–998, 2008.

[303] J. Tang, J. Zhang, D. Zhang, L. Yao, C. Zhu, and J. Li. Arnetminer: An expertise oriented search system for web community. In *Proceedings of ISWC 2007*, 2007.

[304] The Standish group. CHAOS report. Technical report, Standish group, 1994.

[305] The Standish Group. CHAOS Report. Technical report, Standish group, 2009.

[306] A. Tickle, R. Andrews, M. Golea, and J. Diederich. The truth will come to light: Directions and challenges in extracting the knowledge embedded within trained artificial neural networks. *IEEE Transactions on Neural Networks*, 9(6):1057–1068, 1998.

[307] V. Tiwari. Software engineering issues in development models of open source software. *International Journal of Computer Science and Technology*, 2(2):38–44, 2011.

[308] B. Todd and R. Stamper. The relative accuracy of a variety of medical diagnostic programs. *Methods of information in medicine*, 33:402–416, 1994.

[309] P. Tomaszewski, J. Hakansson, H. Grahn, and L. Lundberg. Statistical models vs. expert estimation for fault prediction in modified code-an industrial case study. *Journal of Systems and Software*, 80(8):1227–1238, 2007.

[310] A. Tosun, A. Bener, and B. Turhan. An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536, 2011.

[311] A. Tosun, A. Bener, B. Turhan, and T. Menzies. Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. *Information and Software Technology*, 52(11):1242–1257, 2010.

[312] A. Tosun, B. Turhan, and A. B. Bener. Feature weighting heuristics for analogy-based effort estimation models. *Expert Systems with Applications*, 36:10325–10333, 2009.

[313] I. Tsamardinos, L. Brown, and C. Aliferis. The max-min hill-climbing Bayesian network structure learning algorithm. *Machine Learning*, 65(1):31–78, 2006.

[314] B. Turhan and A. Bener. Software defect prediction: Heuristics for weighted naive bayes. In *Proceedings of the 2nd International Conference in Software and Data Technologies*, pages 244–249, 2007.

[315] B. Turhan and A. Bener. Analysis of Naive Bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering*, 68(2):278–290, 2009.

[316] B. Turhan, G. Kocak, and A. Bener. Software defect prediction using call graph based ranking (CGBR) framework. In *34th Euromicro Conference Software Engineering and Advanced Applications*, 2008.

[317] B. Turhan, T. Menzies, A. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[318] T. Van Gestel, B. Baesens, P. Van Dijcke, J. Garcia, J. Suykens, and J. Vanthienen. A process model to develop an internal rating system: sovereign credit ratings. *Decision Support Systems*, 42(2):1131–1151, 2006.

[319] T. Van Gestel, J.A.K. Suykens, B. Baesens, S. Viaene, J. Vanthienen, G. Dedene, B. De Moor, and J. Vandewalle. Benchmarking least squares support vector machine classifiers. *Machine Learning*, 54:5–32, 2004.

[320] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and Software*, 81(5):823–839, 2008.

[321] J. Vanthienen and G. Wets. From decision tables to expert system shells. *Data and Knowledge Engineering*, 13(3):265–282, 1994.

[322] V. Vapnik. *Statistical Learning Theory*. John Wiley, 1998.

[323] W. Verbeke, K. Dejaeger, D. Martens, J. Hur, and B. Baesens. New insights into churn prediction in the telecommunication sector: a profit driven data mining approach. *European Journal of Operational Research*, 38(1):211–229, 2012.

[324] W. Verbeke, K. Dejaeger, T. Verbraken, D. Martens, and B. Baesens. Mining social networks for customer churn prediction. In *Workshop on Information and Decision in Social Networks*, 2011.

[325] W. Verbeke, D. Martens, C. Mues, and B. Baesens. Building comprehensible customer churn prediction models with advanced rule induction techniques. *Expert Systems with Applications*, 38(3):2354–2364, 2011.

[326] T. Verbraken, W. Verbeke, and B. Baesens. Profit optimizing customer churn prediction with Bayesian network classifiers. *Intelligent Data Analysis*, In press, 2011.

[327] Y. Wang and I. H. Wittig. Induction of model trees for predicting continuous classes. In *Poster papers of the 9^{th} European Conference on Machine Learning*, 1997.

[328] Wasserman. *Social network analysis: Methods and applications*. Cambridge university press, 1994.

[329] S. Watanabe, H. Kaiya, and K. Kaijiri. Adapting a fault prediction model to allow inter languagereuse. In *Proceedings of the 4^{th} International workshop on Predictor models in software engineering*, pages 19–24, 2008.

[330] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang. Systematic literature review of machine learning based software development effort estimation models. *Information and Software Technology*, 54:41–59, 2012.

[331] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1:80–83, 1945.

[332] C. Williams and J. Spacco. SZZ revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36, 2009.

[333] M. Williams. *A history of computing technology*. IEEE Computer Society Press, 1997.

[334] I. Witten and E. Frank. *Data mining: Practical machine learning tools and techniques*. Morgan Kaufmann Publishers, 2005.

[335] G. Wittig and G. Finnie. Estimating software development effort with connectionist models. *Information and Software Technology*, 39(7):469–476, 1997.

[336] X. Wu, V. Kumar, R. Quinlan, J. Gosh, Q. Yang, H. Motoda, G. McLachlan, A. Ng, B. Liu, P. Yu, Z.-H. Zhou, M. Steinbach, D. Hand, and D. Steinbach. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14:1–37, 2008.

[337] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28^{th} International Conference on Software Engineering*, pages 282–291, 2006.

[338] J. Yu, V. Smith, P. Wang, A. Hartemink, and E. Jarvis. Using Bayesian network inference algorithms to recover molecular genetic regulatory networks. In *Proceedings of the 3^{rd} International Conference on Systems Biology*, 2002.

[339] X. Yuan, T. Khoshgoftaar, E. Allen, and K. Ganesan. An application of fuzzy clustering to software quality prediction. In *Proceedings of the 3^{rd} IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, pages 85–90, 2000.

[340] H. Zhang. On the distribution of software faults. *IEEE Transactions on Software Engineering*, 34(2):301–302, 2008.

[341] Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32(10):771–789, 2006.

[342] Y. Zhou, B. Xu, and H. Leung. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software*, 83(4):660–674, 2010.

[343] T. Zimmermann and N. Nagappan. Predicting defect using network analysis on dependency graphs. In *Proceedings of the International Conference on Software Engineering*, pages 531–540, 2008.

[344] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction. In *Symposium on the Foundations of Software Engineering*, 2009.

[345] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the 3^{rd} International Workshop on Predictor Models in Software Engineering*, 2007.

Doctoral dissertations from the faculty of business and economics

See http://www.kuleuven.be/doctoraatsverdediging/archief.htm