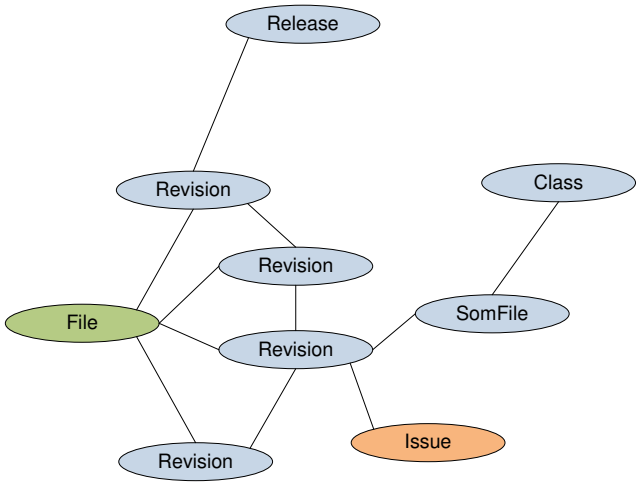
University of Zurich
Department of Informatics

Dynamic and Distributed
Information Systems

Mining Software Repositories with Relational Data Mining Methods

Diploma Thesis February 15, 2008



Sonja Näf
of Ittenthal AG, Switzerland

Student-ID: 03-704-194
snaef@dplanet.ch

Advisor: **Jonas Tappolet**

Prof. Abraham Bernstein, PhD
Department of Informatics
University of Zurich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to thank to Prof. Bernstein and Jonas Tappolet for their patient and helpful support. I appreciate that I got the opportunity to work on this interesting topic. Further, I thank André, Martin, Martina, Mattias, Matthias, Mike and Raphael for spending amusing coffee and lunch breaks with me and/or for proofreading my thesis. Only thanks to my parents who pushed me to do my homework when I was in primary school, it was possible to become a student one day.

Abstract

In complex software projects a lot of information about defect, release and source code history is gathered. Researchers figured out that mining these software repositories could provide valuable information about the software development. So far, software repositories were mined with traditional data mining methods which are suitable for propositional data. Propositional data is flat and homogeneous, held in a single-table-database. This thesis compares the traditional approach with relational data mining methods which are able to handle heterogeneous data. First, an introduction about relational data mining is given and then a few relational data mining tools are introduced. In a next step we present the data for our experiments and the necessary data preparations. Finally, we conduct several experiments which show the advantages as well as the weaknesses of the relational approach.

Zusammenfassung

In komplexen Software Projekten werden viele Daten über die Bug-, Release- und Source Code History gesammelt. Forscher haben herausgefunden, dass Software Repositories wertvolle Informationen zur Softwareentwicklung enthalten. Bis anhin wurde in Software Repositories mit traditionellen Data Mining Methoden nach Mustern gesucht. Traditionelle Data Mining Methoden sind für propositionale Daten geeignet, die in einer einzigen Tabelle gespeichert werden können und somit flach und homogen sind. Diese Diplomarbeit vergleicht traditionelle Vorgehensweisen mit relationalen Data Mining Methoden, welche mit heterogenen Daten umgehen können. Zu Beginn dieser Arbeit werden das relationale Data Mining sowie Tools vorgestellt. Nachher werden wir die zur Verfügung stehenden Daten beschreiben und die notwendigen Vorkehrungen für unsere Experimente erklären. Zum Schluss werden die durchgeführten Experimente, sowie die Stärken und Schwächen des relationalen Ansatzes diskutiert.

Table of Contents

Table of Contents ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Goals of this Work . . . 1
- 1.3 Area of Application . . . 2
- 1.4 Related Work . . . 3

2 Data Mining 5
- 2.1 Definitions . . . 5
- 2.2 Traditional versus Relational Data Mining . . . 6
- 2.3 Relational Data Mining Models . . . 7
  - 2.3.1 Inductive Logic Programming . . . 7
  - 2.3.2 Markov Logic Networks . . . 8
  - 2.3.3 Relational Bayesian Models . . . 8
  - 2.3.4 Relational Markov Networks . . . 9
  - 2.3.5 Relational Dependency Networks . . . 10

3 Analysis Tools 11
- 3.1 Proximity . . . 11
  - 3.1.1 Features . . . 11
  - 3.1.2 Algorithms . . . 12
- 3.2 Alchemy . . . 16
- 3.3 NetKit . . . 16
- 3.4 WEKA . . . 17

4 Data Preparation 19
- 4.1 Ontology Models . . . 19
  - 4.1.1 Software Ontology Model . . . 19
  - 4.1.2 Version Ontology Model . . . 21
  - 4.1.3 Bug Ontology Model . . . 21
  - 4.1.4 Interconnection of the Ontologies . . . 21
- 4.2 Data Preparation . . . 22
  - 4.2.1 Data Selection . . . 23

4.2.2 Data Preprocessing . . . 23
4.2.3 Data Transformation . . . 27

5 Evaluation **29**
5.1 Comparison Study . . . 29
5.1.1 Approach . . . 29
5.1.2 Results . . . 30
5.2 Data . . . 31
5.3 Evaluation Methods . . . 31
5.3.1 Accuracy . . . 32
5.3.2 ROC . . . 32
5.3.3 AUC . . . 33
5.4 Experiments . . . 33
5.4.1 Experiment 1 . . . 34
5.4.2 Experiment 2 . . . 37
5.4.3 Experiment 3 . . . 40
5.4.4 Experiment 4 . . . 43
5.4.5 Experiment 5 . . . 46
5.4.6 Experiment 6 . . . 48
5.4.7 Experiment 7 . . . 50
5.4.8 Experiment 8 . . . 53
5.4.9 Experiment 9 . . . 56
5.4.10 Experiment 10 . . . 59
5.4.11 Experiment 11 . . . 62
5.4.12 Experiment 12 . . . 64
5.4.13 Experiment 13 . . . 67
5.4.14 Summary . . . 70

6 Conclusions **71**
6.1 Limitations . . . 71
6.2 Future Work . . . 72

A Algorithms **73**
A.1 Chi Square Test of Independence . . . 73
A.2 MCMC/Gibbs Sampling . . . 74

B Data Preparation Steps **75**

C Weka Outputs **77**
C.1 Output Weka Ranker Experiment 7 . . . 77
C.2 Output Weka Ranker Experiment 9 . . . 78

D CD **81**

List of Figures **83**

List of Tables **85**

List of Listings 87

Bibliography 89

1

Introduction

1.1 Motivation

To withstand the fierce competition, companies need to be supported by highly available and reliable software. In mid-sized and large companies most of the transactions are accomplished by computers. This makes clear that defects cost a lot of money, not only for the company that misses profits while waiting for the fixed release, but also for the software producer that loses a lot of time and reputation. Unfortunately, avoiding defects is a difficult task to undertake. Software projects are very complex, consist of several ten thousand lines of code and many people work on it. For the project leaders it is indispensable to have tools that allow keeping track of the project and that indicate where problems are expected. One way to support the software engineers is to find patterns in software repositories that give information about the project status. Software repositories consist for example of data gathered by source control, defect tracking or versioning systems. Quite a lot of work was done mining this data with traditional data mining methods and it was also shown that it is worth doing it. So far, no attempts have been made mining software repositories with relational data mining methods. In contrast to the traditional approach, these newer methods can handle relations and unhomogenous data. As the different entities in software data are inhomogeneous and highly interconnected, we think, it is a promising approach using relational data mining methods.

1.2 Goals of this Work

The aim of this thesis is to investigate whether relational data mining techniques are suitable for mining software repositories. With the help of bug, versioning and source code data we want to predict the location of bugs and compare the findings with the good results achieved by Bernstein [Bernstein et al., 2007]. This earlier work was a propositional approach where they achieved the best results with temporal features. As in this thesis the data have a relational nature and in addition to temporal features also source code features are available, we assume to get even a better performance.

This thesis is composed of the following chapters: Chapter 2 provides a brief introduction to the field of data mining and shows what the advantages of relational data mining are in contrast

to traditional data mining. At the end, some common relational models are presented. In chapter 3 different analysis tools are described. Chapter 4 covers the data models and the necessary transformation steps to run mining methods. The evaluation in chapter 5 illustrates the mining experiments and its results. Finally, in chapter 6 conclusions and future work can be found.

1.3 Area of Application

Parnas [Parnas, 1994] states that due to source code modifications and missed adjustments to meet new needs software ages during its life cycle. The consequences of aging are [Parnas, 1994]:

1. customers find other, newer and better products
2. Buggy software due to changes
3. Performance loss because of savaged structure

To detect those problems software repositories are analyzed. There are two main approaches: Software evolution and complexity analysis. The complexity analysis should give insight about the structure of the source code [Lee et al., 1994]. Typical metrics for the complexity are lines of code, McCabes Cyclomatic Number [McCabe, 1976], Halstead's Metrics [Halstead, 1977], etc. The evolution analysis investigates the release history of a software system to get knowledge about the entire system, common change behavior, growth rates of modules or logical couplings between classes [1]. Various combinations of software and complexity analysis were conducted. However, relational data mining methods were never used so far.

Since in many domains data has a relational nature - software repositories as well - relational data mining methods are used in different areas. Where relational data mining (RDM) is not yet applied in the field of mining software repositories, biologists or chemists make widely use of it. In those fields researchers cope with large databases consisting of multiple, interacting relational tables (as an example see [Page and Craven, 2003]). Perlich et al [Perlic and Huang,] stated that RDM methods could be very useful for the customer relationship management (CRM). The goal of CRM is to get to know the customers preferences and behaviors and applying this information for marketing. This requires relational databases that include all customer information available, such as demographics, purchases, linkages to products and other customers etc.. Another area of application is fraud detection. Bernstein et al. [2] try to detect internal fraud in collaboration with a bank. In fraud cases usually several people and transactions are involved, which makes it useful to discover this data with relational data mining methods. As web sites are highly interconnected, in the field of web mining RDM is very common. Goals of web mining tools are for example to give insight about customer profiles, where to place commercials or creating personalized web sites.

Since in other fields relational data mining methods are successfully applied, we will try it out, too.

[1] http://seal.ifi.uzh.ch/fileadmin/User_Filemount/Vorlesungs_Folien/Evolution/SS07/SWEvol-4.pdf

[2] http://www.ifi.unizh.ch/ddis/research/relational-machine-learning-etc/analytical-internal-fraud-detection/

1.4 Related Work

In this section we briefly summarize what studies are important or interesting for this thesis.

For mining software repositories, Kiefer et al. [Kiefer et al., 2007] implemented EvoOnt, a software repository data exchange format based on the Web Ontology Language (OWL). It includes data of source code, versioning and bug tracking systems. They created three ontology models (software, version and bug ontology model) that describe what information of the software repositories are stored. For this thesis this earlier work was very important as this thesis uses their data models. The models are described later in Chapter 4.1.

Knab et al. [Knab et al., 2006] applied a decision tree learner for predicting defect density. Interesting to know from this work is that they found that size metrics such as lines of code aren't very helpful for predicting defects. This in contrast to evolution data as for example the number of modification of reports.

Bernstein et al. [Bernstein et al., 2007] predicted the location and number of bugs with the help of a decision tree learner, respectively with a regression tree learner. They achieved promising results using these two non-linear models if they were based on temporal features. The results of this thesis are compared to their results.

Graves et al. [Graves et al., 2000] developed statistical models for exploring characteristics that indicate large numbers of faults generated in the future development. They used change management data from a large software system with a long history and found as well that process measurements are better predictors than product metrics.

Zimmermann et al. [Zimmermann et al., 2007] mapped defects of the Eclipse project to source code locations. On the basis of complexity measures they built logic regression models to predict whether a file will have defects. They could show that the more complex a class is the more defects it will have, but they didn't succeed in predicting defects reliably.

Neuhaus et al. [Neuhaus et al., 2007] implemented the tool "Vulture" that predicts vulnurable components by analyzing the import structure of software components. By this approach they achieved, analysing the Mozilla project, an average recall of 0.65 and precision of 0.45. In this thesis no data about the import structure is available.

Nagappan et al. [Nagappan et al., 2005] found that failure-prone software components are correlated to code metrics. They state that these code metrics aren't applicapable for every other project, only to the same or similar projects.

In "The Top Ten List" Hassan et al. [Hassan and Holt, 2005] present an approach that indicates managers the ten most susceptible subsystems to have faults. They used a combination of heuristics that consider the recency/frequency (of modifications, fault fixes), the size (of the modification or the subsystem), code metrics and co-modifications.

2

Data Mining

In this chapter a brief introduction about data mining is given, the differences between traditional and relational data mining are explained and some common relational mining models are introduced.

2.1 Definitions

To describe what data mining is, we first introduce Fayyad et al's [Fayyad et al., 1996] definition of Knowledge Discovery in Databases (KDD):

"KDD is the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data."

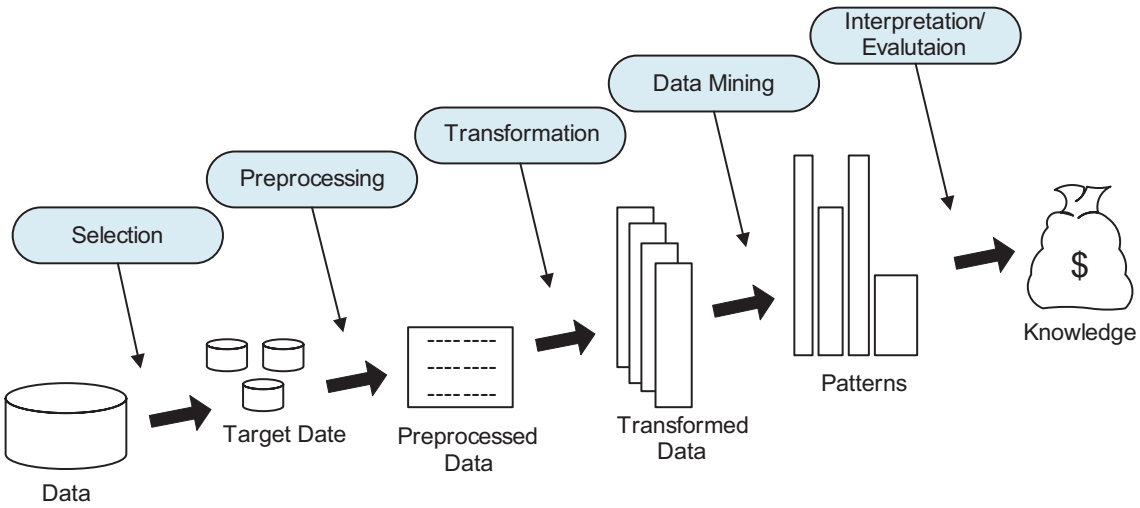


Figure 2.1: Steps in the KDD process [Fayyad et al., 1996]

The goal of this process is to help humans extracting useful information from large databases. It consists of several steps, one of it is data mining. Data mining is defined as the automatic or semiautomatic process of discovering patterns in data [Witten and Frank, 2005]. The identified patterns need to be meaningful and allow us to make predictions on new data. In order to get useful knowledge, before data mining methods can be applied, some other steps have to be fulfilled. Since those steps are important as well for this work, they are explained shortly (see Figure 2.1).

The first step is to understand the data and to know what kind of information should be extracted. In a second step the target data on which the discovery will be performed should be selected. In the third step the data is cleaned. This could, for example, be to decide what to do with missing data fields. Next, in the transformation step the amount of data is reduced by finding good features that represent the data depending on the task. After that, data mining methods are chosen and applied. Typical outputs are classification and regression trees and association, classification and regression rules [Džeroski and Lavrač, 2001]. Finally, the results should be interpreted and the whole process is possibly repeated.

2.2 Traditional versus Relational Data Mining

In traditional data mining the input data is usually stored in a single table where the table headers consist of attribute names and each row corresponds to an instance. Such data is called propositional data. Algorithms for propositional data assume that the instances are stored in a homogeneous structure where there exist for every object a fixed number of fields. Additionally, it is assumed that the data instances are independent and identically distributed [Neville et al., 2003b]. However, most data is stored in relational databases that consist of several tables with heterogeneous data records and relations. When relational databases should be mined with traditional data mining methods the information is truncated first, to be aggregated into one table. First of all this is costly and also information is lost.

Example A big shopping center with several selling points keeps account about their customers' purchases. In a relational database consisting of the two tables customer (see Table 2.1) and purchase (see Table 2.2) information about customers and their purchases is stored. Due to the aggravated competition they want to find out the characteristics of good customers with the goal to earn more money. As they use traditional mining methods they first aggregate the purchases by summing up the money the customers spent. This leads to a single table (see Table 2.3). From this they follow that Mr. Fischer is a much better customer than Mrs. Fischer as he spends more than three times more than her. However, they do not have any information about what was bought or how often someone went shopping. With relational mining methods this aggregation is not necessary. Hence, with the non-aggregated tables it is shown that Mrs. Mueller spends less in total, but she spends above-average for technical products. It is also seen the preferences when to go shopping.

Due to the aggregation of customers' spendings they did not have as much information about their customers as they would have had in a relational setting and probably miss profits. For this reason researchers developed relational data mining methods that can handle this kind of data structure. These methods are also called statistical relational learning methods and should

Table 2.1: Customer table

id	name	city	sex	nationality	age	profession
1	Mueller	Zurich	w	CH	28	pilot
2	Fischer	Zofingen	m	D	43	teacher
...						

Table 2.2: Purchase table

id	customer	date	article	amount	price	shop
1	1	2007-12-12	notebook	1	2000	Zurich
2	1	2007-12-12	PS 3	1	900	Olten
3	2	2007-01-06	sandwich	3	18	Basel
4	2	2007-01-07	salad	1	2	Basel
...						
n	2	2007-x-y	z	u	v	Basel

perform at least as well as traditional data mining methods [Neville et al., 2003b]. However, more computation is needed because of irregular structures and complex dependencies.

2.3 Relational Data Mining Models

As illustrated in the section above, a table with a fixed number of attributes is often not a good way to model data. Needed are objects described by any number of attributes where those objects can stand in relation to other objects. In real data uncertainty arises for example about the attributes of an object, the number of objects or the number of relations [Getoor and Taskar, 2007]. To handle with this uncertainty models based on combinations of graphical models, probabilistic grammars and logical formulas were built [Getoor and Taskar, 2007].

In this section the most common models are introduced.

2.3.1 Inductive Logic Programming

An example for a logic model is the Inductive Logic Programming (ILP) [Džeroski, 2007]. The advantage of ILP is that it can deal with data stored in multiple tables. It is possible to combine it with classification rules, decision trees or association rules to upgrade those propositional models to relational models. Decision trees combined with ILP are almost the same as propositional trees with the distinction that the upgraded relational trees have first-order logic queries in the node to classify the instances. However, finding patterns expressed in first-order logic needs more computational power than for conventional patterns. Another drawback is that it cannot handle uncertainty, either a statement is true or not. Since in reality usually it is not possible to distinguish clearly between true and false, other, more smoother models were built, such as Markov Logic Networks [Domingos et al., 2006].

Table 2.3: Aggregation table

id	name	city	sex	nationality	age	profession	total spendings
1	Mueller	Zurich	w	CH	28	pilot	2900
2	Fischer	Zofingen	m	D	43	teacher	10000
...							

2.3.2 Markov Logic Networks

A Markov Logic Network is a set of formulas in first-order logic with attached weights [Domingos et al., 2006]. Thus, logic is combined with probability and it is possible to have not only true and false but also more and less probable formulas. When the variables in these formulas are replaced by constants a Markov Network can be built. This is a graphical model consisting of undirected edges and is useful if the direction of the interconnected objects/variables is not obvious. As ILP MLN can be combined with other models to extend them to relational models. Methods in the data mining tool Alchemy are based on MLN (see Section 3.2).

2.3.3 Relational Bayesian Models

Another common example for graphical models are Relational Bayesian Models (RBM) [Getoor et al., 2007]. These extend Bayesian networks (BN) with objects, relations and their attributes in order to allow them to model relational data. BNs are directed acyclic models, representing probabilistic relationships among attributes.

A RBM consists of a schema of the domain and a probabilistic graphical model that describes the dependencies in the domain. The schema of the domain is a description of the defined objects, attributes and relations. It can be presented in an relational schema σ_r (see Figure 2.2). A relational skeleton is a partial instantiation of the schema and specifies for each class in the schema the objects and relations between them, without giving values to the attributes. When the Tables 2.1 and 2.2 are considered, customer Mueller with the id 1 is linked to two purchases. This is shown in the relational skeleton (see Figure 2.3). In the dependency structure S the dependencies of the attributes are specified. As shown in Figure 2.4 attributes of an object can depend on attributes of the same object or on attributes of related objects. If there is a one to many relation as in the example of Section 2.2 between the tables customer and purchase, the attribute depends on the aggregated value.

As in BN's, given the parents an attribute is independent of the other attributes. This results to the joint distribution in Equation 2.1. X corresponds to objects, A to attributes, I to the instantiation of the schema and Pa to parents.

$$P(I|\sigma_r, S, \theta_S) = \prod_{x \in \sigma_r} \prod_{A \in A(x)} P(I_{x.A}|I_{Pa_{(x.A)}}) \tag{2.1}$$

Here, only the most basic form of RBM was discussed, assuming that there is uncertainty about the attributes of the objects. However, it is possible to introduce uncertainty about reference or existence uncertainty. For further explanations see [Getoor et al., 2007].

In PROXIMITY a Relational Bayesian Classifier, based on the theory of BN's is available (see 3.1.2).

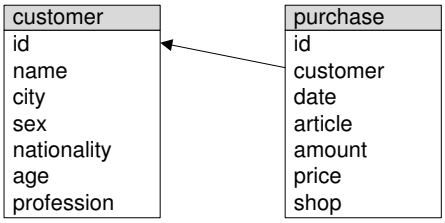


Figure 2.2: Schema RBM

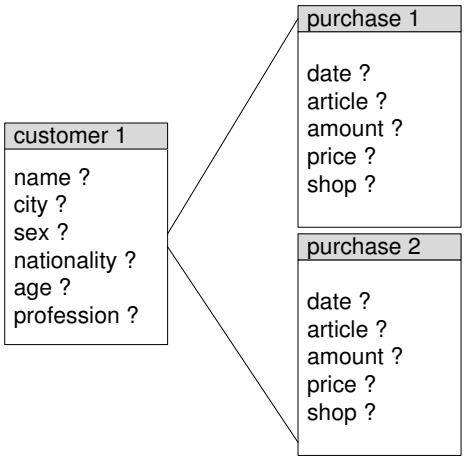


Figure 2.3: Relational Skeleton

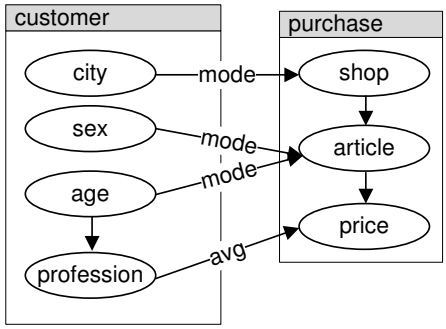


Figure 2.4: Dependency Structure

2.3.4 Relational Markov Networks

Relational Markov Networks (RMN) are an extension of Markov Networks. Contrary to RBN, RMN are undirected graph models where cycles are allowed [Taskar et al., 2007]. In contrast to directed models that classify each label separately, in RMN labels are collectively classified. The

idea of this approach is that similar entities are connected. If the shopping center mentioned in example of Section 2.2 knew the friends of customers, they could assume that these friends are more likely to buy similar articles. To define this correlation relational cliques are introduced. This can be done with a SQL query as in Listing 2.1.

```
SELECT customer1.Article, customer2.Article
FROM Customer customer1, Customer customer2, Link link
WHERE link.hasFriend = customer1.ID and link.isFriendOf = customer2.ID
```

Listing 2.1: SQL Query for Cliques

The conditional distribution of a RMN is then defined as in Equation 2.2. C corresponds to the cliques, I to the instantiations and Φ to the potential function. $Z(I.x, I.r)$ is a normalization function, called partition function (see Listing 2.3).

$$P(I.y|I.x, I.r) = \frac{1}{Z(I.x,I.r)} \prod_{c \in C} \prod_{c \in C(I)} \Phi_C(I.x_C, I.y_C) \tag{2.2}$$

$$Z(I.x, I.r) = \sum_{I.y'} \prod_{C \in C} \prod_{c \in C(I)} \Phi(I.x_C, I.y'_C) \tag{2.3}$$

2.3.5 Relational Dependency Networks

Relational Dependency Networks (RDN) are an extension of traditional dependency networks [Neville and Jensen, 2007]. RDN have characteristics of RBN and as well of RMNs. RDN is implemented in the data mining tool PROXIMITY and is further explained there (see Section 3.1.2).

3

Analysis Tools

In the evaluation in Chapter 5 several tools were used that we want to present. For mining the software repositories primarily PROXIMITY was used. For feature selection we used WEKA and for drawing diagrams Matlab [1]. Since Matlab is well-known, we will not further explain it.

3.1 Proximity

PROXIMITY is an open-source software system for mining relational data. It was designed and implemented by the Knowledge Discovery Laboratory (KDL) in the Department of Computer Science at the University of Massachusetts Amherst [2]. Proximity supports major research results of KDL, such as for example the three mining methods mentioned later in Section 3.1.2.

3.1.1 Features

MonetDB In PROXIMITY all the data are hold in MONETDB [3]. This is an open-source database system specially designed for high-performance applications in data mining. In contrast to, for example MySQL[4], the tables are fragmented vertically instead of horizontally [Ivanova et al., 2007]. Thus, the performance should be improved because most scientific applications access just a few columns in a table at a time. In this way only the relevant columns have to be fetched from disk.

Java Interface To interact with the database and visualize the data PROXIMITY provides a Java interface that connects to MonetDB.

QGraph PROXIMITY also provides QGraph [Blau et al., 2002], a visual query language. Users create a query by drawing a graph consisting of the objects and links that should be considered. Special for QGraph is that executing a query results in subgraphs which consist of the related objects and links. For defining the cardinality of objects and links annotations can be added. The

[1] http://www.mathworks.com/
[2] http://kdl.cs.umass.edu/proximity/
[3] http://monetdb.cwi.nl
[4] http://www.mysql.com/

query in Figure 3.1 would result in a container including all subgraphs with a customer that went shopping once or or more often. The result can be viewed in a textual mode (see Figure 3.2) or as a visualized graph (see Figure 3.3). Additionally, conditions on the object attributes can be set. Hence, it is possible to restrict this query to customers which are older than forty.

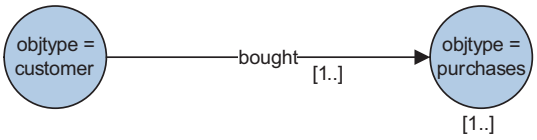


Figure 3.1: QGraph query

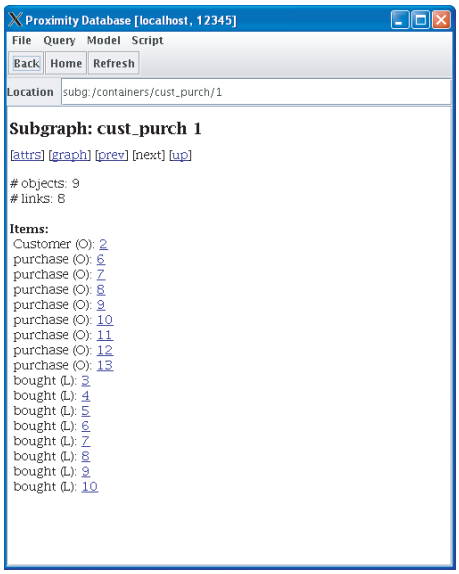


Figure 3.2: Contents of a Subgraph in Text Form

Python Scripts Another point worth mentioning is that PROXIMITY 's Java API can be invoked by Python[5] scripts or the interactive interpreter built in PROXIMITY . For example with the very short script in Listing 3.1 a new attribute for the object "customer" is created that counts for every customer the number of purchases.

3.1.2 Algorithms

In this section three relational data mining methods implemented using the data mining tool Proximity are presented. All the three different algorithms have in common that they train and test on subgraphs which are created by QGraph (see Section 3.1.1). A subgraph is an instance of a data set including one or more objects with its links and attributes and exactly one class label to predict which has to be of an non-continuous value.

[5] `http://www.python.org/`

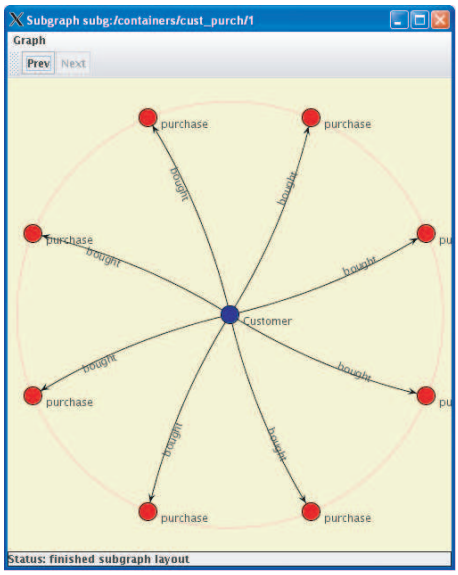


Figure 3.3: Contents of a Subgraph as a Graph

The Relational Bayesian Classifier (RBC) is a modification of the Simple Bayesian Classifier (SBC), adapted to relational contexts [Neville et al., 2003b]. The SBC is a typical traditional data mining method based on Baye's rule assuming that attributes are independent given the class. Considering the example in Table 2.3, this means that the probability that someone is a good customer given the attributes "city", "sex", "nationality", "age" and "profession" can be calculated as in Equation 3.1. Multiplying the probabilities of the attributes to get the conditioned probability is only possible because of the independence assumption [Witten and Frank, 2005].

$$P(C = YES|a_{1-5}) = \alpha \prod P(A_i = a_i|C = YES)P(C = YES) \tag{3.1}$$

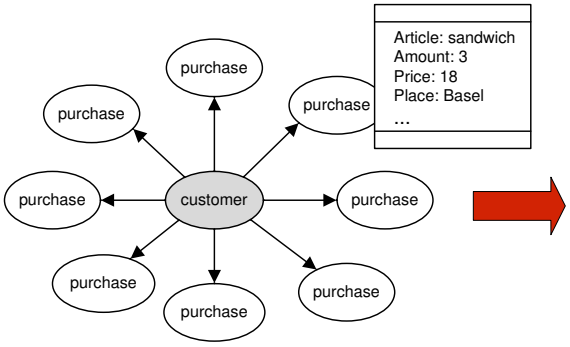
In order to be able to handle with relational data, the RBC flattens first the data into a homogeneous set of attributes (see Figure 3.4).

The challenge in this relational setting is to find a good estimator for the probabilities of multivalued attributes, for example for the attribute "articles". Neville et al. [Neville et al., 2003b] found that this works best when an independent value estimator is chosen. This estimator assumes that each value of a multivalued attribute is independently drawn from the same distribution. As an example the first instance of the attribute "article" is shown in Figure 3.4. To calculate

```
bought = DB.getLinks("linktype = 'bought'")
#group o2_id (purchases) by o1_id (customers) and count
numPurchases = bought.aggregate("count", "o1_id", "o2_id")
prox.objectAttrs.defineAttributeWithData("numPurchases",
       "int", numPurchases)
```

Listing 3.1: Python Script

the probability $P(YES|articles)$ the probabilities of the single values given the class 'YES' have to be multiplied (see Equation 3.2).



Good Customer	Customer Name	Purchase Article	Purchase Price	Purchase Place	...
NO	Berger	{wine,coke ,MP3}	{6, 2, 100}	{Zug, Zug, Bern}	...
YES	Mueller	{notebook, PS 3}	{2000, 900}	{Zurich, Olten}	...
YES	Fischer	{sandwich, salad, ...}	{18, 2, ...}	{Basel, Basel, ...}	...
...	...	...	...	...	...

Figure 3.4: Data Flattened

$$P(YES|articles) = P(wine|YES)P(coke|YES)P(MP3|YES)P(YES) \tag{3.2}$$

Once the probabilities of the attributes are estimated, classification is done by choosing the label value with the highest probability.

The Relational Probability Tree is a classification tree for relational data [Neville et al., 2003a]. It considers attributes of the target object, related objects and links. Classification trees are easily interpretable, which makes them to a very popular data mining method. As shown above, the heterogeneous data has to be flattened somehow. The RPT is doing this by using aggregation functions. Following aggregations functions are implemented: `MODE/AVERAGE, MINIMUM, MAXIMUM, EXISTS, COUNT, PROPORTION, DEGREE`. The RPT is constructing trees by recursively partitioning subgraphs. In each splitting step it chooses one of the seven features with the best score. This score is based on chi-square to calculate how much the feature depends on the class label. If a feature is not significant it is dropped. For further explanation about how this score comes up see Appendix A.1.

In Figure 3.5 an example output tree is shown. The data is the same as for the RBC where it is predicted whether a customer is a good customer or not. In every node of the tree a feature was selected. In the first step the data is divided by the feature `average`. If a customer spends in average more or equals 100 the data instance traverses the left branch. "nop" means "no operation" and is used if no aggregation is needed. In this example every customer has just one age that is why "nop" is used. The thicker a branch is the more instances pass this branch down. The red bars are equivalent to the Yes instances proportional to the No instances (blue bars). In the leaf nodes the number of instances that arrived at this leaf is depicted. If, for some reasons, for example due to missing values, an instance could not be properly predicted, an appropriate percentage of the instance to each path is assigned, this leads to numbers with decimal places.

Relational Dependency Networks As already mentioned before in Chapter 2, Relational Dependency Networks (RDN) [Neville and Jensen, 2007] are an extension of Dependency Networks,

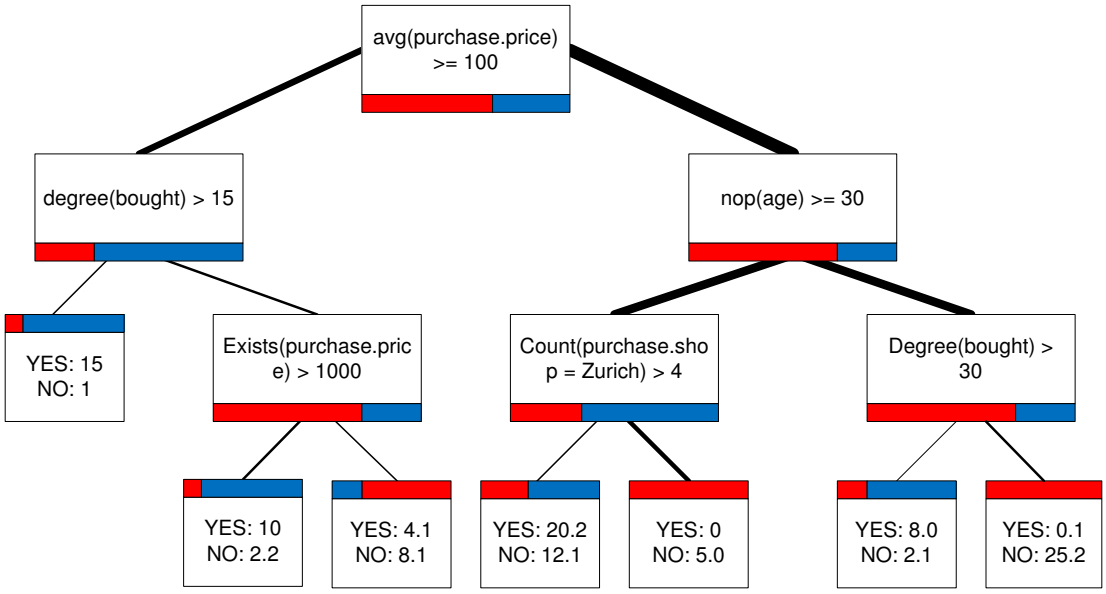


Figure 3.5: RPT output

used in traditional data mining. RDN have characteristics of RBNs and also of RMNs. RDNs are bidirected models. The most important strength of RDN is that it can learn and reason with cyclic relational dependencies, this in contrast to RBNs. This means that, as in the case of RMN (see Section 2.3.4), inference can be done collectively what makes results more precisely. RMNs are also able to handle with cyclic relational dependencies, but Neville et al. state that learning in RDNs is more efficient. RDN learns conditional probabilities independently with for example the RBC or RPT implemented in PROXIMITY instead of learning it jointly. For inferencing Gibbs Sampling is used that approximates the joint distribution (see Appendix A.2).

Results of RDNs can be seen in a dependency network (see Figure 3.6). It is shown that "profession" depends on "GoodCustomer". The loop for "GoodCustomer" indicates autocorrelation. This means, it is easier to predict whether someone is a good customer or not when related customers (for example friends) are considered, too.

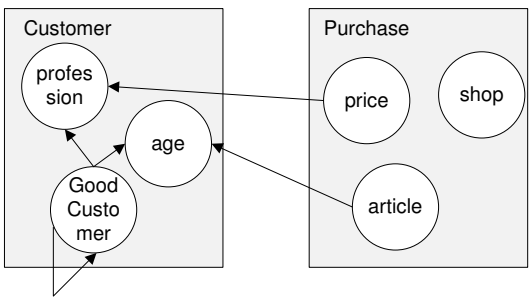


Figure 3.6: Dependency Graph

3.2 Alchemy

ALCHEMY is an open software package for statistical relational learning based on Markov Logic [Domingos and Richardson, 2007]. It is designed to run on Linux platforms. As mentioned in Chapter 2 Markov Logic consists of first-order logic formulas with attached weights. Predicting whether someone is a good or bad customer could be expressed by the following formula (see Equation 3.3). It says that pilots, older than thirty, are good customers.

$$\begin{aligned} \forall x Customer(x) \wedge HasProfession(x, y) \wedge y = Pilot \wedge HasAge(x, z) \wedge z > 30 \\ \Rightarrow GoodCustomer(x) \end{aligned} \tag{3.3}$$

The three main features of ALCHEMY are:

- weight learning
- structure learning
- inference

As it says, weight learning learns the weights for formulas. Since the statement in Equation 3.3 is not true in every case it can be softened by a weight. When structure learning is used, not only weights are learnt, but it also searches for new clauses. ALCHEMY supports the two basic types MCMC[6] and MAP[7]/MPE[8] for inferencing. Inferencing yields either to the probability or to the most likely state of the query atoms. ALCHEMY is a command line tool and can be run with the commands `learnweight`, `learnstruct` or `infer` plus several options.

In this thesis PROXIMITY was preferred for three main reasons. First, PROXIMITY provides an user-friendly interface for exploring and visualizing the data. In ALCHEMY all the data are stored in files. Second, it is easier to understand the results if they are presented as a decision tree or dependency network as it is provided by PROXIMITY . The other point is that in our domain it is very difficult to construct formulas. We do not know exactly what causes an issue. One evidence for an issue in the actual file could be if there were several issues in the last two, maybe also three or five, months. However, such kind of statements are cumbersome to express in first-order formulas. Nevertheless, in a future work, once good features already exist, it would be very interesting to know how this tool performs.

3.3 NetKit

NETKIT is a command line toolkit for statistical relational learning, written in Java [Macskassy and Provost, 2005]. This tool provides three main modules:

- Local classifier
- Relational classifier

[6] Markov chain Monte Carlo
[7] Maximum a Posteriori
[8] Most Probable Explanation

- Collective inferencing

The local classifier module is a non-relational classifier, using only the attributes on the target object. It can be applied to get to know the prior probabilities. It is possible to use WEKA classifiers [9]as local classifiers. The following relational algorithms are implemented: weighted-vote Relational Neighbor (wvRN), class-distributional Relational Neighbor (cdRN) and a network-only Multinomial Bayes Classifier with Markov Random. Similar to PROXIMITY , NETKIT provides several aggregators. With the combination of those aggregators non-relational WEKA classifiers can be used as relational classifiers. Currently, for inferencing Relaxation Labeling, Iterative Classification and Gibbs Sampling are available.

All the data are held in CSV-files and the graph has to be described in an ARFF[10] file. NETKIT can be run with any combination of local/relational classifiers and collective inferencing. An example run could look as the following:

```
java -jar ../NetKit.jar -rclassifier wvrn -showAUC schema.arff
```

There is a bunch of further useful command line options, all described in the tutorial, found on the main web page [11].

Important to know is that the relational classifiers in NETKIT are univariate models. Univariate models have the characteristic that they consider only class labels and not local attributes [Macskassy, 2007]. The idea is that for example friends of a good customer are good customers, too. However, in our domain it is essential considering as well local attributes. Macksassy states that univariate relational classifier with local classifier were combined but with limited success. He suggests to create links between similar entities to eliminate the drawback of ignoring local attributes. This is an interesting approach but it goes beyond this thesis. Another drawback of NETKIT is as in the case of Alchemy that there are not any tools for visualizing the data provided. For this reasons we decided to use PROXIMITY as the main tool. Though, it would be interesting to know how the aggregators implemented in NETKIT perform together with the WEKA classifiers.

3.4 WEKA

WEKA is a data mining tool of the university of Waikato in New Zealand [Witten and Frank, 2005]. It does not only provide a collection of the standard machine learning algorithms, but it also supports users for data preparation and analysis. In contrast to PROXIMITY that only provides classification algorithms, in WEKA classifiers, regression, clustering, association rules and attribute selection methods are available.

However, WEKA has one big drawback. It is, different to the tools mentioned above, only applicable to propositional data. Nevertheless, due to the lack of some useful features in PROXIMITY such as feature selection algorithms in this thesis as well WEKA was used.

[9] http://www.cs.waikato.ac.nz/ml/weka/
[10] http://www.cs.waikato.ac.nz/~ml/weka/arff.html
[11] http://www.research.rutgers.edu/~sofmac/NetKit.html

4

Data Preparation

In this chapter the data used is presented and the necessary steps of the KDD process (see Section 2.1) are shown, before data mining methods can be applied. The goal of this procedure is to be able to use PROXIMITY as the mining tool. An overview of the whole data preparation process is shown in Figure 4.1.

4.1 Ontology Models

In this thesis history data of the Eclipse project's compare plugin was used. This data was gathered in an earlier diploma thesis [Tappolet, 2007]. In that diploma thesis a plugin was implemented, which extracts data from source code, versioning and bug tracking systems and which stores this data in the OWL[1]/RDF[2] format. The data is structured in three models: `Software Ontology Model` (SOM), `Bug Ontology Model` (BOM) and `Version Ontology Model` (VOM). So, for each of the three repositories an ontology model was created. In the ontology model the structure of the data is defined, that is the entities, attributes and linkages. Since during the development of a software project the three repositories arise together, the three different models are interconnected. As for our purpose not all information stored in the models are relevant, in the pictures below only the data used by this approach are shown. The rectangles correspond to objects, the ellipses to attributes and the arrows to the linkages between the objects.

4.1.1 Software Ontology Model

The purpose of the `SOM` is to represent object-oriented software source code. As in Figure 4.2 depicted, this model consists of a `somFile`, a class that includes zero or more methods, attributes, local variables and formal parameters. The name `somFile` is chosen instead of `file` because the `VOM` already has an entity called `file`. The difference between the two files is that the file in the `VOM` exists only once and has no information about the source code. Whenever there is a new revision of this file it receives an additional link to the corresponding revision. The file in the `SOM` appears as many times as there were revisions done on this file.

[1] `http://www.w3.org/TR/owl-features/`
[2] `http://www.w3.org/RDF/`

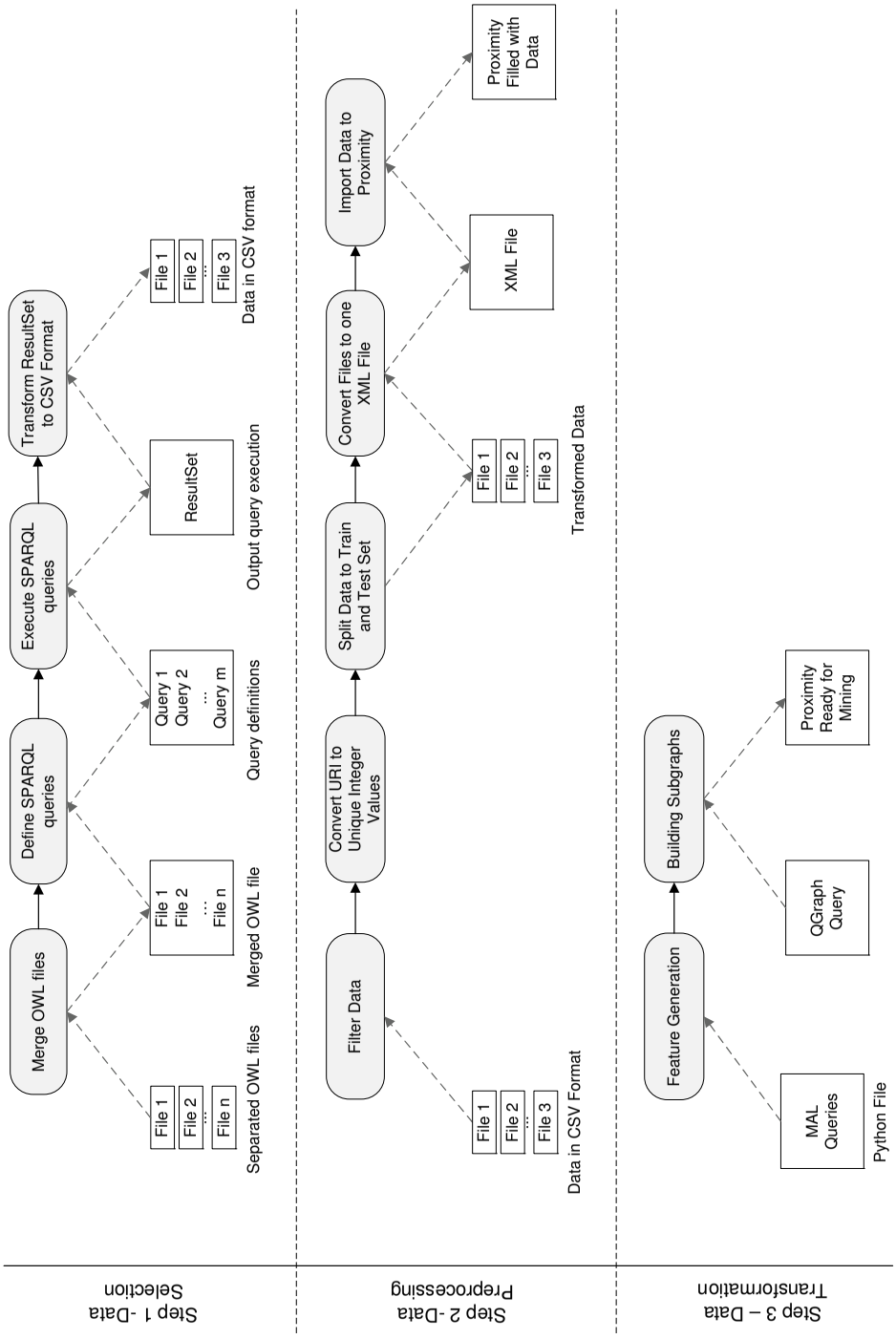


Figure 4.1: Data Preparation

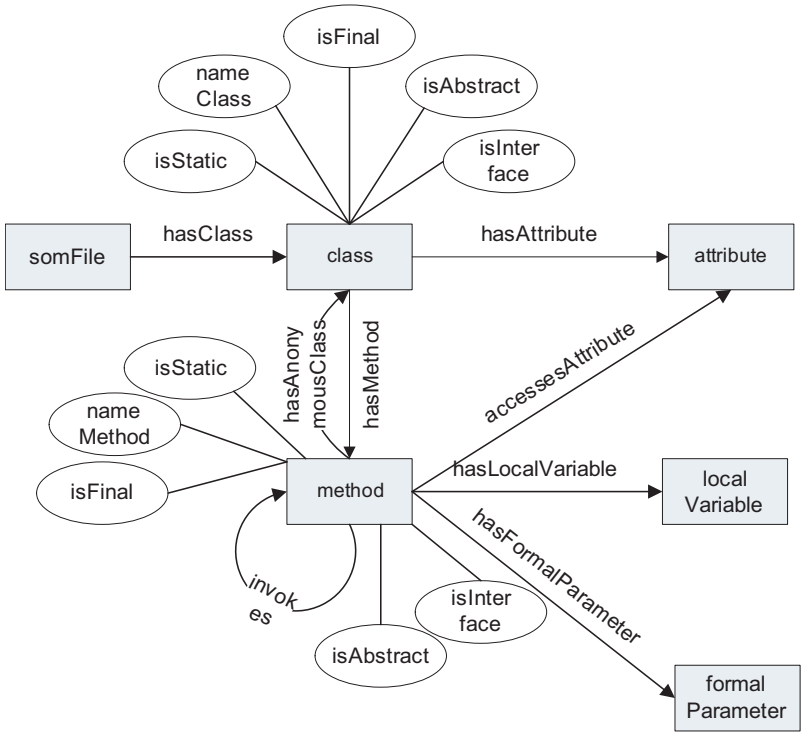


Figure 4.2: Software Ontology Model

4.1.2 Version Ontology Model

The data in the `VOM` was extracted from CVS[3]. This model includes information about the entities `revision`, `release` and `file` (see Figure 4.3). For every file the past revisions are stored. In the attributes of the revision information about the revision number, author, creation time and how many lines were added/removed is stored. In versioning systems it is possible to add several different revisions to one release. That is why for every revision it is stored to which release it belongs to.

4.1.3 Bug Ontology Model

The data in the `BOM` was gathered from Bugzilla[4]. In reality the `BOM` composes of more entities. However, for our evaluation the information in Figure 4.4 is enough.

4.1.4 Interconnection of the Ontologies

With the interconnection of the ontologies more information about the software development process is added. Hence, there is for example information about which file was changed at which time and whether in this revision a bug was fixed. For this connection the three following links

[3] `http://www.nongnu.org/cvs/`
[4] `http://www.bugzilla.org/`

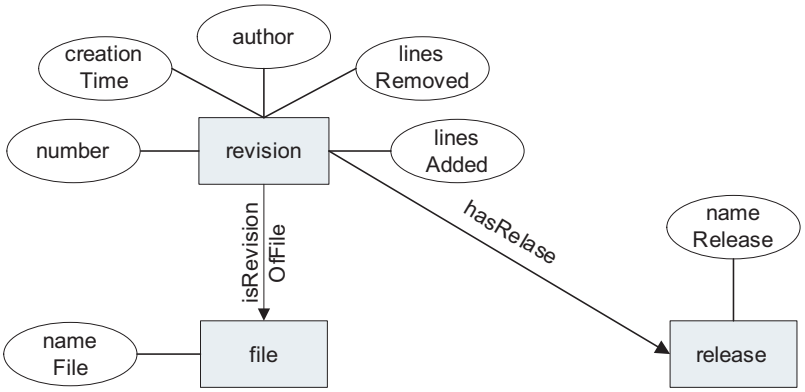


Figure 4.3: Version Ontology Model

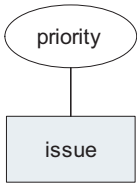


Figure 4.4: Bug Ontology Model

were needed: `isRevisionOfSomFile` between `revision` and `somFile`, `isResolutionOf` between `revision` and `issue` and `hasSomRelease` between `somFile` and `revision` (see Figure 4.5).

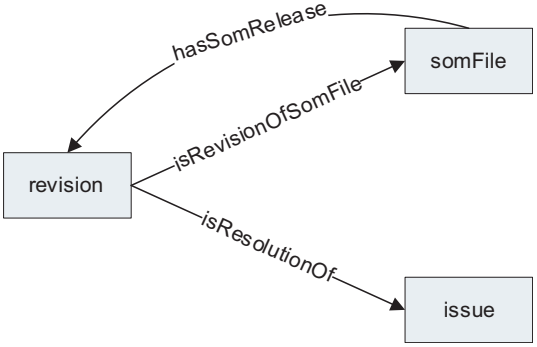


Figure 4.5: Interconnection of the Three Ontology Models

4.2 Data Preparation

In this section the necessary steps in order to run mining methods over the data are presented. Doing this, we followed the steps in the KDD process (see Section 2.1).

4.2.1 Data Selection

As a result in the mentioned diploma thesis above, the needed data is available in separated OWL files where there is one file for the revision data, one for the bug data and as many files as there are releases for the software code data.

As not all of this data is needed, the data we are interested in was selected. How this was done, is explained in this section.

Merge Data First of all we merged all OWL files into a single file to be able to easily query the data needed afterwards. For this step all the necessary tools were already prepared in the diploma thesis mentioned before (see Section 4). For every file a unique namespace was defined and stored in a XML file. They implemented a Java class "DataModel" with which it is possible to merge the files, given the path where the files are and a string array including the namespaces of the files that should be merged (see listing 4.1).

```
DataModel dm = new DataModel("path/to/datafolder");
OntModel model = dm.getOntology(OntModelSpec.OWL_DL_MEM, namespaces);
```

Listing 4.1: Merge Data

Define SPARQL Queries To import data into PROXIMITY the data has to be in the XML-format. In this XML file all the objects, links and attributes that should be imported have to be defined. For this reason SPARQL [5] queries for every object including its attributes and links we are interested in were created. Once the data is imported, it is not possible to import additional objects at a later time. Therefore, it is important to extract all the necessary objects right from the beginning. SPARQL is a query language for RDF/OWL data with a similar syntax to SQL with the main difference that SPARQL operates with namespaces. An example query is shown in Listing 4.2 that queries all entities of the type revision with its revision number, creation time, author and if available the state of the revision.

Execute SPARQL Queries The query defined before is just a string and is not executed yet. For the execution the class "QueryExecuter" was constructed that returns the result in a ResultSet (see Listing 4.3).

Transform ResultSet to CSV Format For further processing steps this ResultSet was parsed and transformed to the CSV format. For every type of object and link a separated file is generated. This is necessary for a further step.

4.2.2 Data Preprocessing

Several preprocessing steps were necessary. As it is relational data and the entities are interconnected, the easiest way to fulfill the different cleaning tasks is to do it all in the same step. However, for better understanding, the different steps are explained separately.

[5] http://www.w3.org/TR/rdf-sparql-query/

```
QueryExecuter qe = new QueryExecuter();

String revisions =
        "PREFIX vom: <" + Namespace.VOM + "> " +
        "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
        "SELECT ?URI ?number ?creationTime ?author ?state " +
        "WHERE { " +
                "?URI rdf:type vom:Revision . " +
                "?URI vom:number ?number . " +
                "?URI vom:author ?author . " +
                "?URI vom:creationTime ?creationTime . " +
                "OPTIONAL { " +
                        ?URI vom:state ?state . " +
                " }. " +
        "} ";

qe.executeQuery(revisions, "revision");
```

Listing 4.2: Define SPARQL Queries

```
public void executeQuery(String query, String queryName) {
        OntModel model = ModelFactory.createOnotlogyModel();
        Query q = QueryFactory.create(query);
        QueryExecution exec = QueryExecutionFactory.create(q, model);

        ResultSet result = exec.execSelect();

        ResultParser parser = new ResultParser();
        ArrayList data = parser.parseToProximityData(result);
}
```

Listing 4.3: Execute SPARQL Queries

Data Filtering First, the interesting data was selected and stored in CSV files. However, there is still undesired data or data in the wrong format. The three main filtering tasks were:

- Filtering Files: We are only interested in Java files. However, among the files there are as well other types of files, such as XML or other configuration files. As those files are linked as well to revisions, releases, bugs and source code, filtering those files does not only mean to ignore the unwanted files but also its linked entities. Optionally, a file needs to have at least one revision in the last six months, otherwise it is ignored.
- Filtering Revisions: When a file is deleted it is connected to a revision with the remark "dead". As we are not interested in files that do not exist anymore, those files with all it connected entities are deleted.
- Filtering somFiles: A revision can be linked to several releases. When there is a new release, in the current data model the revision is linked to a new source code file, even though it is the same. By filtering doubled source code files we made sure that every revision is connected only to one source code file.

Convert URI to Unique Integer Values Important to note is that every object and link must have a unique identifier. In PROXIMITY this identifier needs to be an integer equal or greater than zero. In the OWL files, objects and links already have an unique identifier. However, it is in form of an URI as it is common in the OWL format. For converting the URIs to unique integer values, all the URIs of the objects are stored in a hash map where the URI is a key and the unique integer is the value of the corresponding key. The unique integer is created with a counter, which is increased for every new object added in the hash map. As a result we receive unique integer values for every object. Every link needs also a unique identifier, however, only among the links not among the objects. Again, there is a counter whose value is assigned to a link. A link consists of a source and a destination URI which correspond to objects. These URIs have as well to be converted to an integer value according to the URI of the objects stored in the hash map.

Specifying Type of Revisions Later, for mining we need to now how the current source code of a file looks like. That is why the latest revision of a file has to be determined. In the data for the revisions there is a link `hasNextRevision`. When a revision does not have a next revision, we know that this is the latest revision and it contains the information about the actual file. For the revisions the attribute `targetRevision` is added with the value one, if it is the latest revision of a file and zero otherwise.

Convert Files to one XML File It is only possible to import one XML file into PROXIMITY . This means that all the data that is currently separated on several files have to be merged into one single file. For this step we made use of a perl file that is provided by PROXIMITY . Thanks to the steps before, the data is already in the suitable format. We only have to write a specification file for the objects and one for the links (see Figure 4.4, 4.5). As the objects and links always have identifiers they do not need to be mentioned in the specification file. Only the attribute names and their data types are needed. Possible data types are: `BIGINT, DATE, DATETIME, DOUBLE, INTEGER, VARCHAR`. If this is done, the perl script can be executed with this command:

```
perl text2xml.pl new-db objspec linkspec tab
```

The four command line parameters are:
1. name of database to be created
2. name of the object specification file
3. name of the link specification file
4. separator character that separates fields in the text files (either comma, tab or space)

As a result of this execution one XML file that includes all the objects, links and attributes is generated.

```
revision<eol>
<tab>objecttype<whitespace>VARCHAR<eol>
<tab>number<whitespace>VARCHAR<eol>
<tab>creationtime<whitespace>DATE<eol>
<tab>author<whitespace>VARCHAR<eol>
<tab>removed<whitespace>INTEGER<eol>
<tab>added<whitespace>INTEGER<eol>

filename<eol>
<tab>attributename<whitespace>datatype<eol>
...
```

Listing 4.4: Object Specification File

```
hasClass<eol>
<tab>linktype<whitespace>VARCHAR<eol>

hasMethod<eol>
<tab>linktype<whitespace>VARCHAR<eol>

filename<eol>
<tab>attributename<whitespace>datatype<eol>
...
```

Listing 4.5: Link Specification File

Importing XML File Finally, the XML file can be imported with a script as well provided by PROXIMITY . To import this file of course the Monet database has to be started first and then the script, named `import-xml.bat`, has to be executed. How this works exactly, is nicely explained in the tutorial of PROXIMITY [6].

[6] `http://kdl.cs.umass.edu/proximity/documentation/Tutorial.pdf/`

4.2.3 Data Transformation

In Section 4.2.2 the necessary steps to import the data in PROXIMITY was shown. However, some further steps are necessary to run the mining algorithms.

Creating Subgraphs In section 3.1.2 we stated that the mining methods of PROXIMITY need a container consisting of several subgraphs as an input. As mentioned in section 3.1 these subgraphs and containers can be created with QGraph. Therefore, the query shown in Figure 4.6 is executed. It results in a container including all subgraphs with a file, its latest revision and if available its source code. How QGraph queries are defined is explained in a tutorial provided by the community of PROXIMITY [7].

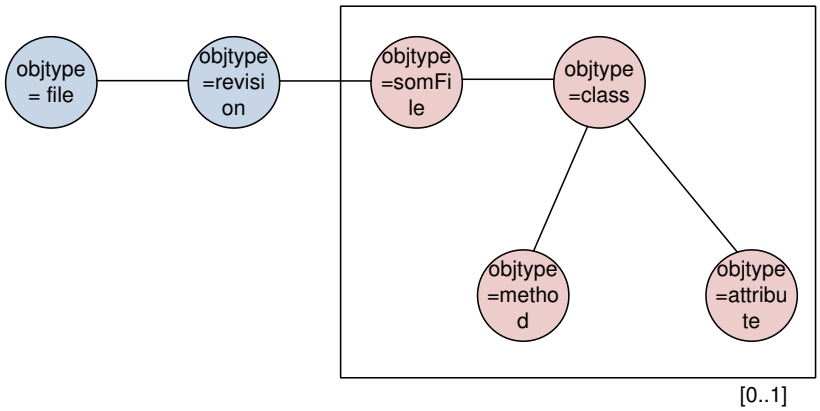


Figure 4.6: QGraph Query

Creating Features PROXIMITY provides an interface with which it is possible to query the MONETDB similar to SQL. The query language of MONETDB is called MAL and stands for "MonetDB Assembly Language" [Ivanova et al., 2007]. Once a query is executed, its result can be stored in a new attribute. All MAL queries and attribute generations are defined in a python script and then executed. As an example Listing 4.6 is shown. In that example the goal is to create an attribute that stores for every file how many revisions it had in the last month. That is why all revisions with a creationtime later than 2006-12-30 are joint with the link hasRevision. The result is then aggregated to count the according links between file and revisions.

As those files without any revisions in the last month are not included, the result is stored as a intermediate step in the variable `temp`. In Listing 4.7 an attribute function is defined that creates the attribute `revision1month`. If the variable created before (`temp`) is greater than zero then it gets this value, else zero. At the end the temporal variable `temp` is deleted.

Finally, all necessary steps are accomplished and we are now ready to run the data mining methods.

[7] `http://kdl.cs.umass.edu/proximity/documentation/QGraphGuide.pdf`

```
rev = DB.getObjects("creationtime >= ’2006-12-30’")
hasRevision = DB.getObjects("linktype = ’hasRevision’")
nst = rev.join(hasRevision, "id = o2_id")
nst = nst.project("id, o1_id")
numRev = nst.aggregate("count", "o1_id", "id")
numRev = numRev.renameColumns("id, value")
prox.objectAttrs.defineAttributeWithData("temp", "int", numRev)
```

Listing 4.6: MAL Query

```
currentAttrs = prox.objectAttrs
newAttrName = "revision1month"
newAttrFunction = "objtype = \"file\" AND temp > 0 ==> temp, objtype =
   \"file\" ==> 0"

prox.addAttribute(currentAttrs, newAttrName, newAttrFunction)
currentAttrs.delete("temp")
```

Listing 4.7: Attribute Function

5

Evaluation

Citation of Nagappan et al [Nagappan et al., 2005]:

"The key idea of using software repositories is that one can map problems (in the bug database) to fixes (in the version database) and thus to those locations in the code that caused the problem".

This is exactly the setting in this thesis. The goal of this evaluation is to find out if this relational combination of data is useful for predicting the location of bugs.

First, the paper of Bernstein [Bernstein et al., 2007] is introduced since we use this work for comparison. Then the chosen data and evaluating methods are presented. Finally, the conducted experiments and its results are described and arrived to a conclusion.

5.1 Comparison Study

To know whether the approach with relational data for predicting bugs is a good approach or not, a traditional approach for comparison is needed. Since in [Bernstein et al., 2007] good results were achieved, we decided to compare with them. This paper is introduced here briefly.

5.1.1 Approach

Bernstein et al. [Bernstein et al., 2007] argue that using temporal features and non-linear models is essential for predicting the location of defects and the number of reported bugs. The created features listed in Table 5.1 are mainly temporal features. For predicting the location of defects they used WEKA 's decision tree learner J48 and for predicting the number of defects WEKA 's tree regression learner M5P.

CVS and Bugzilla data provided by the MSR Mining Challenge 2007 [1] was used. This consists of the plugins `updateui, updatecore, search, dpeui, pdebuild` and `compare` of the Eclipse project. For learning they extracted features for releases until December 31 2006 and for testing for releases until January 31 2007.

[1] `http://msr.uwaterloo.ca/`

Table 5.1: Features

	Name	Description
1	LOC	Number of lines of code
2	LineAddedIRLAdd	Number of lines added to fix a bug relative to total number of lines added
3	LineAddedIRLDel	Number of lines deleted to fix a bug relative to total number of lines deleted
4	AlterType	Amount of modification done relative to LOC
5	AgeMonths	Age of a file in months
6	RevisionAge	Number of revisions relative to the age of a file
7	DefectReleases	Number of releases of a file from Dec 1 to 31 of 2006
8	Revision1Month	Number of revisions of a file from Dec 1 to 31 of 2006
9	DefectAppearance1Month	Number of releases of a file with defects from Dec 1 to 31 of 2006
10	ReportedI1Month	Number of revisions of a file with defects from Dec 1 to 31 of 2006
11	Revision2Month	Number of revisions of a file from Nov 1 to 31 of 2006
12	DefectAppearance2Month	Number of releases of a file with defects from Nov 1 to 31 of 2006
13	ReportedI2Month	Number of revisions of a file with defects from Nov 1 to 31 of 2006
14	Revision3Month	Number of revisions of a file from Oct 1 to 31 of 2006
15	DefectAppearance3Month	Number of releases of a file with defects from Oct 1 to 31 of 2006
16	ReportedI3Month	Number of revisions of a file with defects from Oct 1 to 31 of 2006
17	Revision5Month	Number of revisions of a file from Aug 1 to 31 of 2006
18	DefectAppearance5Month	Number of releases of a file with defects from Aug 1 to 31 of 2006
19	ReportedI5Month	Number of revisions of a file with defects from Aug 1 to 31 of 2006
20	ReportedIssues	Total Number of revisions of reported problems
21	Releases	Total Number releases
22	RevisionsAuthor	Number of revisions per author

5.1.2 Results

For selecting the best features they used a feature selection algorithm provided by WEKA . The significant features for predicting the location of defects in all five plugins were 2, 3, 8, 11, 16, 19 of Table 5.1. With an accuracy (ACC) of 99.164% and the area under curve (AUC) of 0.9251% the location of defects was predicted.

5.2 Data

In this thesis only the `compare` plugin was used as the OWL format is very memory consuming. We used as well the phase until 2006-12-31 to train and predicted bugs to the end of January 2007. For more comparison possibilities the same time range, just eleven months later (2007-06-30 to 2007-12-30) was considered as well.

The main set up is the same as in the earlier work mentioned before, however there are some differences. First, instead of propositional data relational data was used. Second, in addition to the evolution data as well source code data was considered (see Ontology Models in Section 4.1).

To know whether a file has an issue or not and how the current source code looks like those revisions that do not have a link `hasNextRevision` (meaning that this is the last revision of a file) were marked as `target revisions` (see Figure 5.1). In a first step we only considered files that have at least one revision in the time range. Later, also files with no revision during the selected time range were considered.

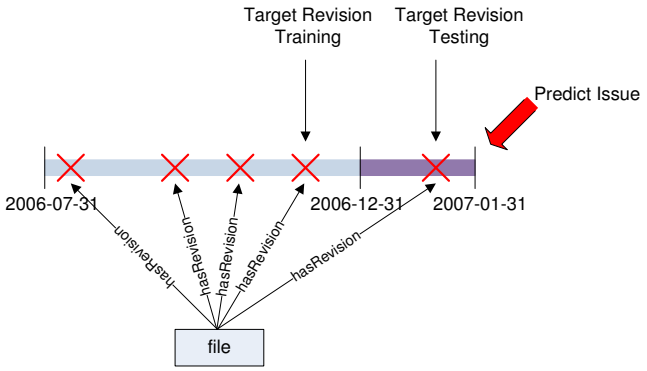


Figure 5.1: Time Range

In this thesis, in addition to the features of the comparison study, the features in Table 5.2 were available. In Table 5.3 some distributions of the data are shown. It shows the number of files considered in a time range. Note that for the phase from 2006-07-31 to 2007-01-31 only files were considered that had at least one revision during the time range. This in contrast to the phase in 2007 where all files are included. Further, the number of issues and the resulting expectation that a file will not have an issue are shown. In the last column the number of files with a target revision linked to a SOM file is listed. From there we know that in the time range 2006/07 quite a lot of target revisions have missing links. That is why we expect worse results for SOM features than in the time range 2007.

5.3 Evaluation Methods

For evaluating the experiments different measures were used which are briefly explained in this section.

Table 5.2: Source Code Features

	Name	Entity	Description
1	AccessControlQualifier	Method	public, private or protected
2	isAbstract	Method	true, false
3	isFinal	Method	true, false
4	isInit	Method	true, false
5	isInterface	Method	true, false
6	isStatic	Method	true, false
7	numForeignMethInv	Method	num of methods of another class a method invokes
8	numFormalParam	Method	num of parameters
9	numLocalVar	Method	num of local variables
10	numMethInv	Method	num of methods a method invokes
11	prevRevHasIssue	Revision	true, if revision before of that file had an issue
12	invClassHasIssue	Method	true, if class of invoked method has issue

Table 5.3: Distribution Data

Time range	Num files	Num files with issue	Prior no issue	Num target revisions with SOM file
train: 2006-12-31	88	13	85.2%	78
test: 2007-01-31	84	14	83.3%	60
train: 2007-11-30	161	16	90.06%	161
test: 2007-12-30	161	15	90.06%	161

5.3.1 Accuracy

The ACC measures how many instances were correctly classified in proportion to the total number of instances. When the values of the class label are not equally distributed, it might mistakenly seem that a good result was achieved. On 2007-11-30 in Table 5.3 only 16 of 161 files have an issue. When a mining method returns an accuracy of 90% for this data set, this is just what was expected and could have been achieved without data mining.

5.3.2 ROC

ROC stands for receiver operating characteristics and measures as well the performance of classifiers [Witten and Frank, 2005]. In comparison to the accuracy, beside the true positive rate also the false positive rate is considered. The false positive rate is the number of instances wrongly classified as positive divided by the total number of negative instances. In Figure 5.2 a ROC curve is depicted. The goal is to receive curves that are close to the point (0,1). The diagonal between the points (0,0), (1,1) is achieved when the instances are randomly classified.

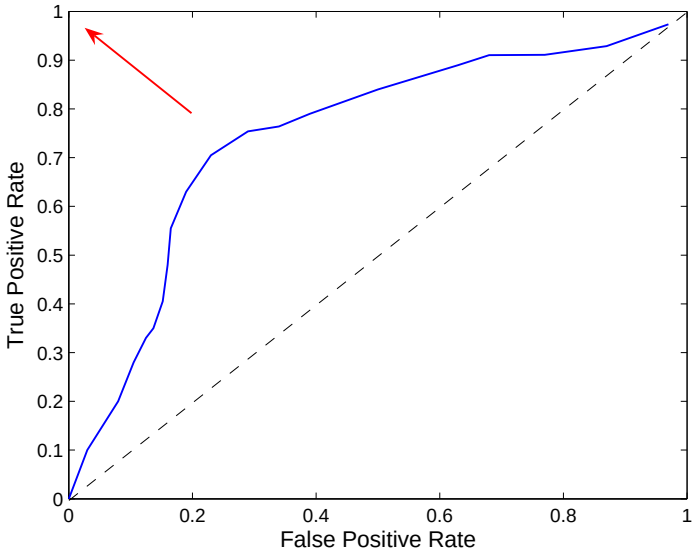


Figure 5.2: ROC Curve

5.3.3 AUC

AUC stands for area under curve and is a related measurement to ROC [Witten and Frank, 2005]. The larger the area under the ROC curve the larger is AUC and the better the result.

5.4 Experiments

In this section the experiments conducted with PROXIMITY are presented. For every experiment the RPT and RBC were tested ten times and then the ACC/AUC averaged. RPT was run with the default setting, this is with pValue = 0.05 (see Appendix A.1) and maxTreeHeight = 5. For time reasons RDN was just run once. The conditional probabilities, used by RDN, are learnt with RPT. For inference in RDN, Gibbs Sampling was 2000 times repeated with a burn in of 100.

The different experiment runs differ in five points: Features, time range, files, aggregators and revisions considered. Some key words are defined that are used later in the experiment descriptions:

1. Features
   (a) COM features: features of the comparison study in Table 5.1
   (b) SOM features: source code features in Table 5.2
2. Time range
   (a) 2006/07: training phase up to 2006-12-31 and the test phase on 2007-01-31
   (b) 2007: training phase up to 2007-11-30 and the test phase on 2007-12-31

3. Files
   (a) All files: all files, no matter they had a revision during the time range considered or not
   (b) Files with revision: only files that had at least one revision during the time range considered
4. Revisions
   (a) All target revisions: all files, no matter if they have a SOM file or not
   (b) Target revisions with SOM file: only target revisions that have a link to a SOM file

5.4.1 Experiment 1

The first experiment is run to see what happens and so all features available are considered.

Features	COM and SOM features
Time range	2006/07
Files	Files with revision
Revisions	All target revisions
Aggregators	All

Result In the decision tree created by RPT two COM features were selected to the two top features of the tree (see Figure 5.3). When a file did not have a defect in the last five months it is very likely not to be defect one month later. The tree shows that if a file had defects and not a lot of revisions were done, the file is very likely to be buggy in the future, too. This means that files are not debugged within a few revisions. In the rest of the nodes SOM features were chosen. The more complex a class is, that is the more entities a class has, the higher the probability that the file has defects. Methods with protected modifiers are less fault-prone than methods with public or private modifiers. The performance of all the three methods is worse than in the comparison study. Either the relational methods are not as good as WEKA 's J48 or the fact that all features were used, disturb the result. This has to be tested in another experiment (see experiment 4). RBC performed clearly better than the other two methods (see Table 5.4) what can be seen as well in the ROC curves (see Figures 5.4 and 5.5). This means that the splitting nodes created by RPT are not significant enough.

Table 5.4: Result Experiment 1

Model	ACC	AUC
RBC	0.8090	0.8446
RDN	0.8427	0.6577
RPT	0.8426	0.6086

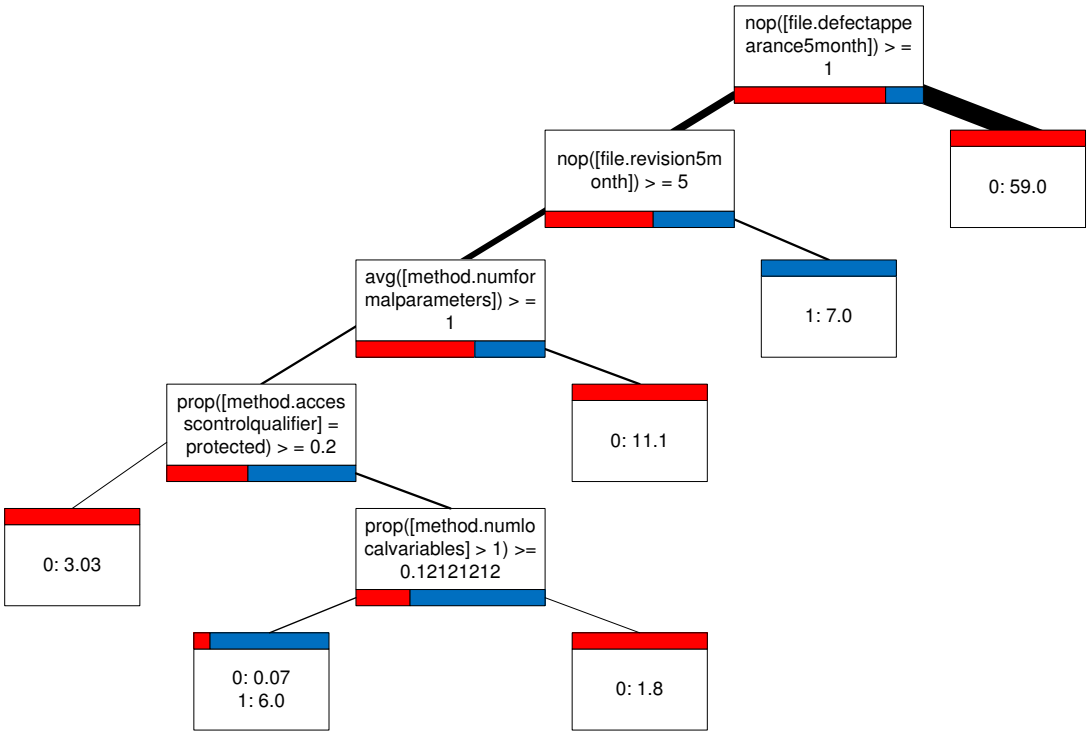


Figure 5.3: RPT Experiment 1

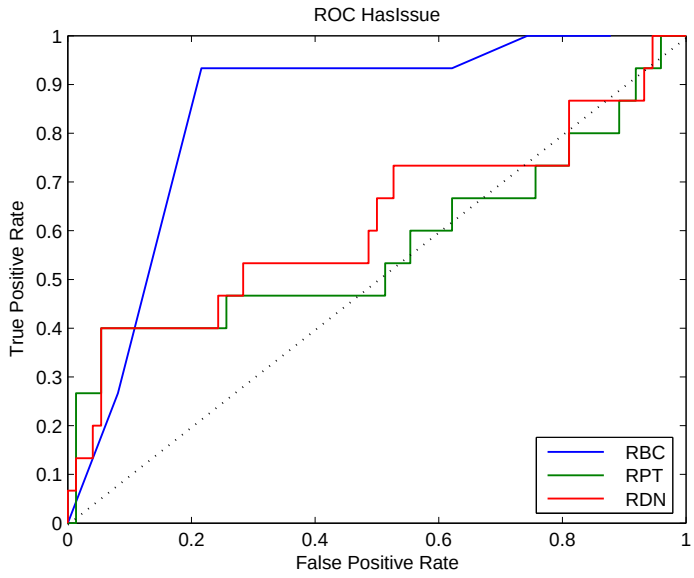


Figure 5.4: ROC HasIssue Experiment 1

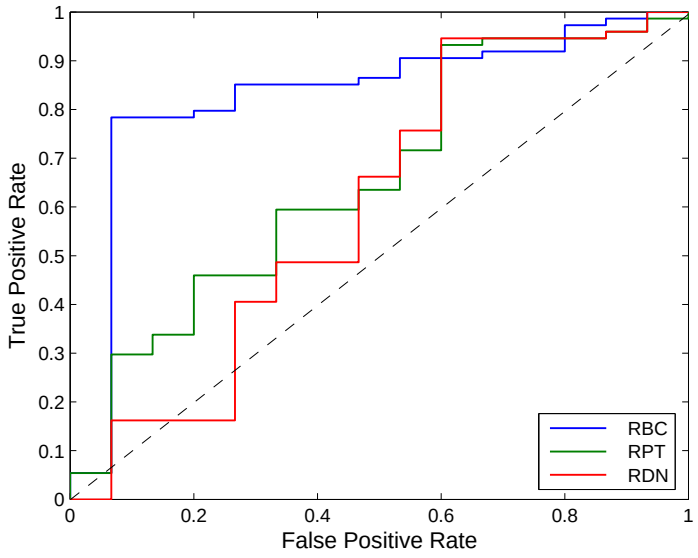


Figure 5.5: ROC HasNoIssue Experiment 1

5.4.2 Experiment 2

Experiment 2 should give insight what performance SOM features alone can achieve.

Features	SOM features
Time range	2006/07
Files	Files with revision
Revisions	All target revisions
Aggregators	All

Result In Table 5.5 and also in the ROC curves in Figures 5.6 and 5.7 it is shown that the classification in this experiment is pure random. The resulted tree in Figure 5.8 is larger than the one in experiment 1. There are a lot of small numbers of instances reached the leaves, indicating that there is not a real significant feature found. However, this is not really astonishing, as in the training data only about 78 of 88 files and in the test data only 60 of 84 files have a target revision connected to a SOM file.

Table 5.5: Result Experiment 2

Model	ACC	AUC
RBC	0.8539	0.4725
RDN	0.8427	0.4333
RPT	0.8427	0.4159

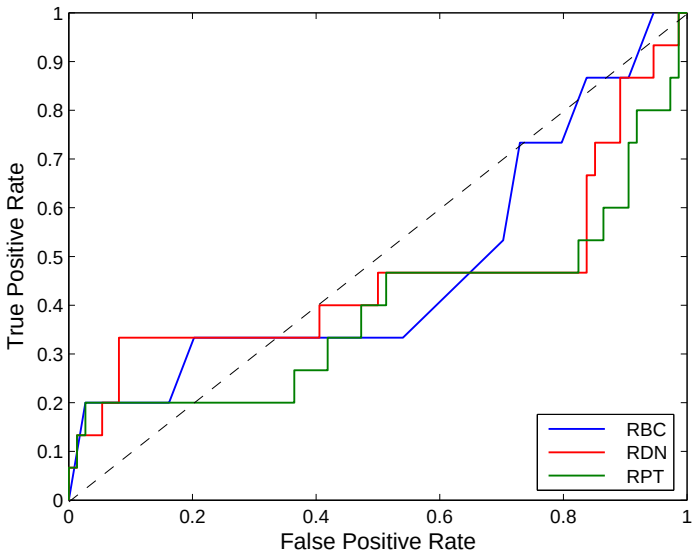


Figure 5.6: ROC HasIssue Experiment 2

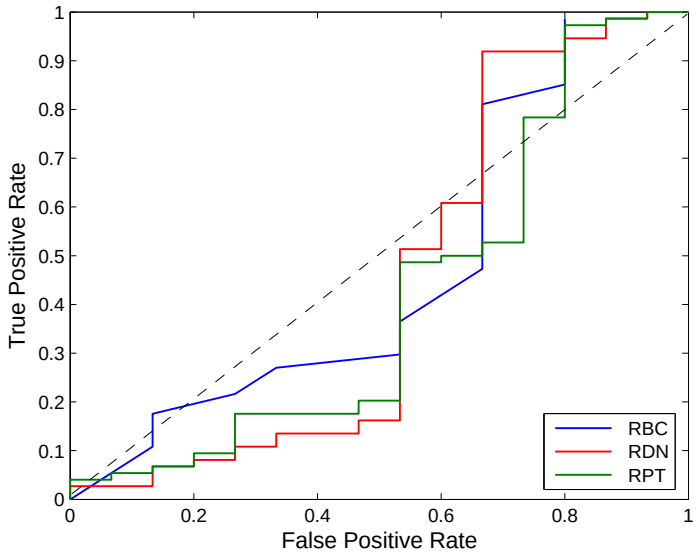


Figure 5.7: ROC HasNoIssue Experiment 2

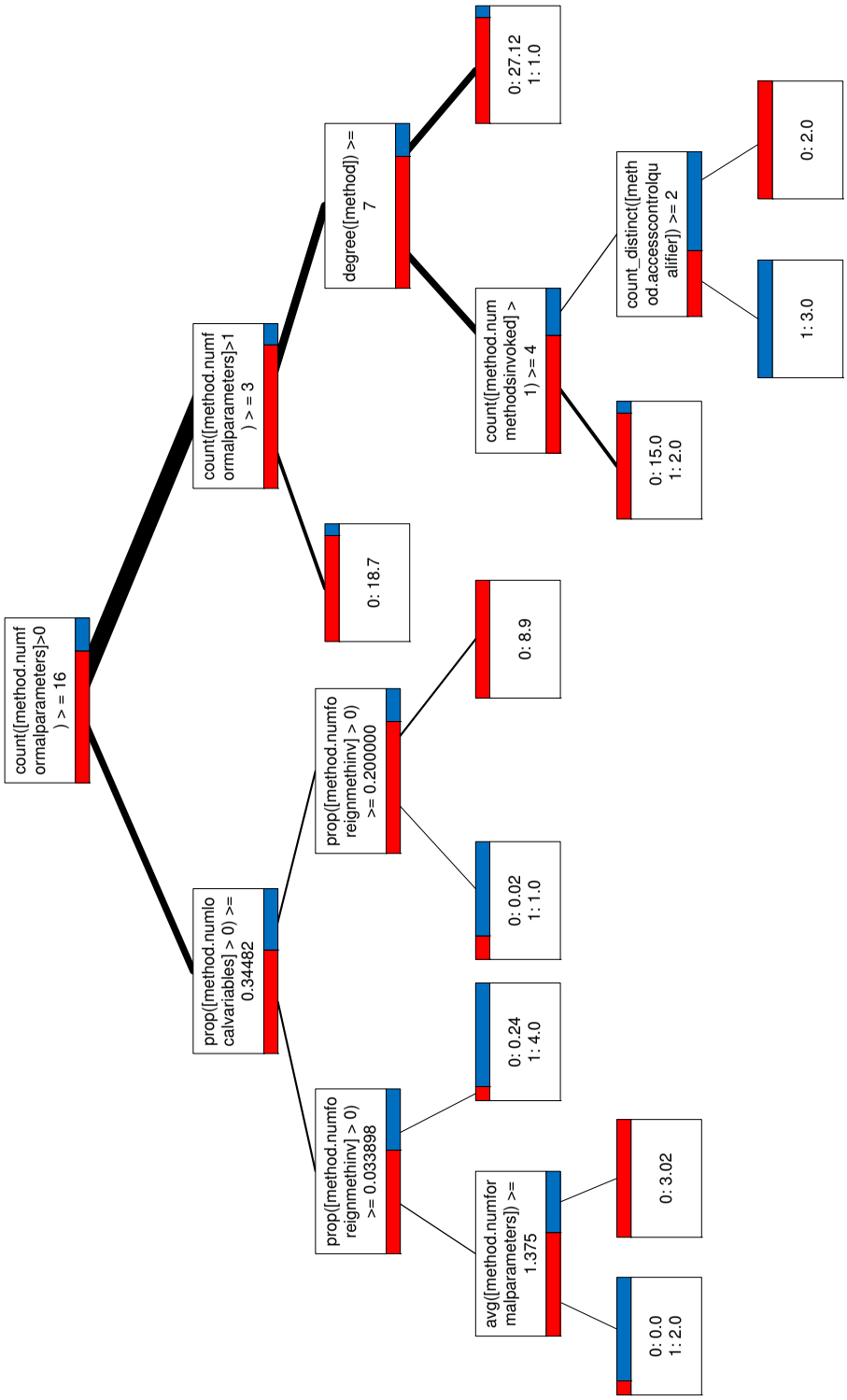


Figure 5.8: RPT Experiment 2

5.4.3 Experiment 3

The SOM features in the experiment before came off badly. It might be that this was because of missing links between target revisions and SOM files. That is why this experiment is repeated and this time only files with target revision linked to a SOM file are considered.

Features	SOM features
Time range	2006/07
Files	Files with revision
Revisions	Target revisions with SOM file
Aggregators	All

Result The results for RDN and RPT are this time even worse (see Table 5.6) than in the experiment before. Looking at the decision tree in Figure 5.11 shows that not very meaningful splitting nodes were found. As an example, the node stating that if there are more methods invoked the file is less likely to have an issue (see Figure 5.11). The outcome of this experiment implies that the missing links to source code files, mentioned in the experiment before, were in the case of RPT and RDN not the reason for the bad results. Instead, it indicates that the existing SOM features are not expressive enough to predict defects. However, RBC could improve its result compared to the experiment before, meaning that in this case the missing links had some influence.

Table 5.6: Result Experiment 3

Model	ACC	AUC
RBC	0.8166	0.7049
RDN	0.7666	0.4161
RPT	0.7833	0.4159

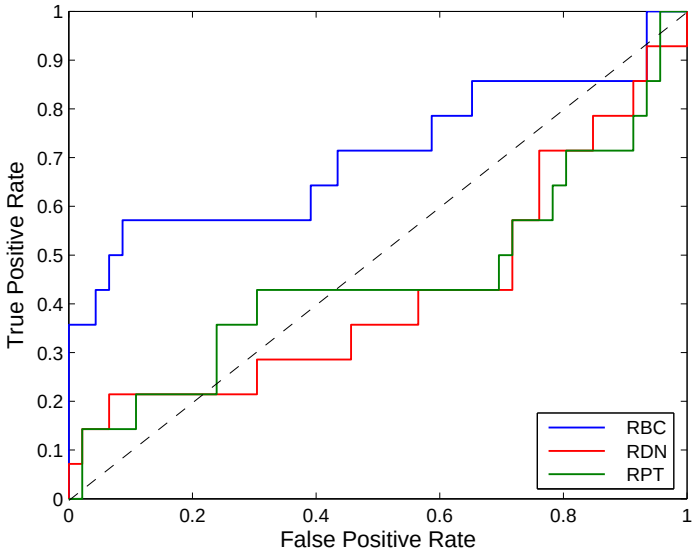


Figure 5.9: ROC HasIssue Experiment 3

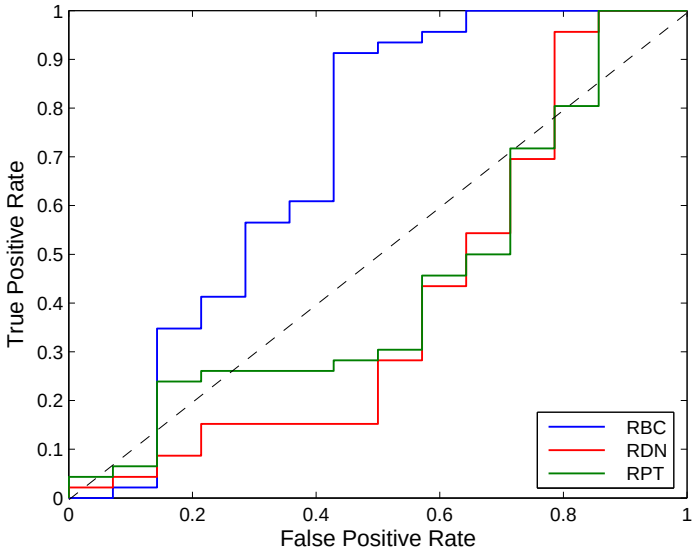


Figure 5.10: ROC HasNoIssue Experiment 3

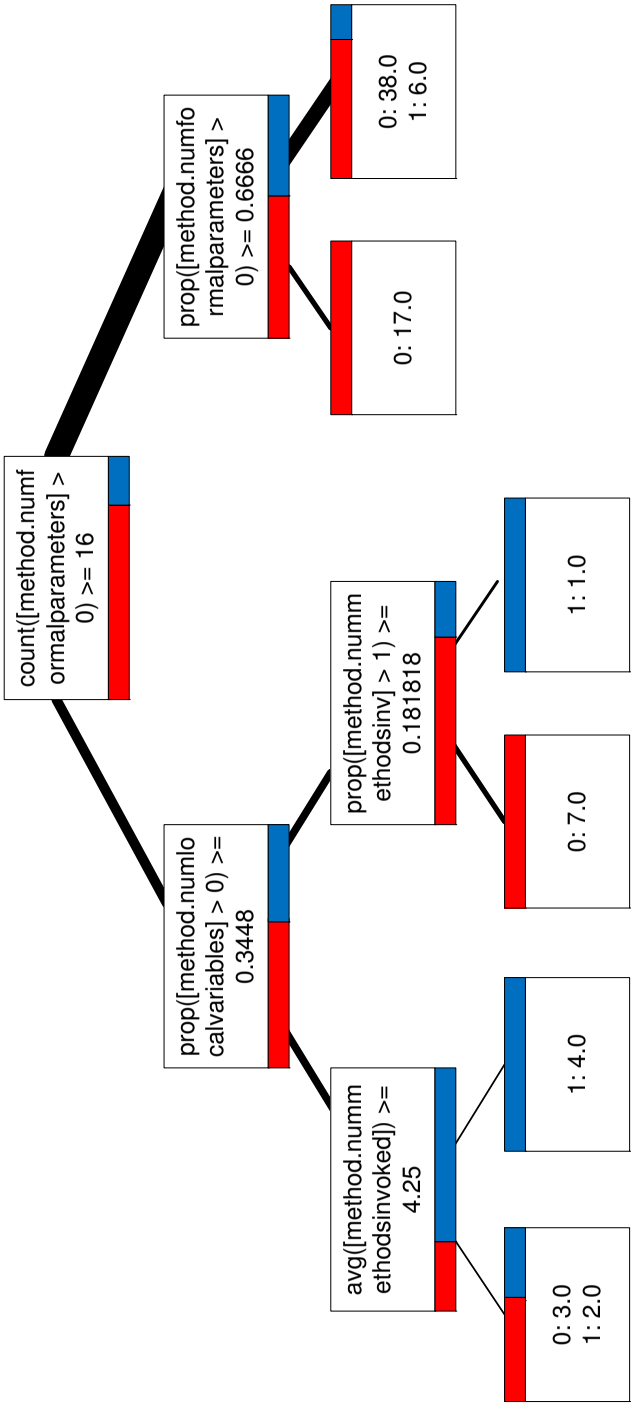


Figure 5.11: RPT Experiment 3

5.4.4 Experiment 4

As in experiment 1 already remarked, it should be tested if relational methods perform as good as WEKA 's J48. In this experiment only the significant COM features created in the comparison study are used.

Features	Significant COM features (2,3,8,11,16,19 in Table 5.1)
Time range	2006/07
Files	Files with revision
Revisions	All target revisions
Aggregators	All

Result The AUC (see Table 5.7) and the ROC curve (see Figures 5.13/5.14) in this experiment are comparable to the AUC/ROC curve in the comparison study. However, the ACC is clearly smaller. As in the comparison study the features `reportedI5months` and `revision2months` belong to the top features. Here, the most top feature `DefectAppearance1Month` was not chosen, but instead PROXIMITY selected the feature `degree[class]`. This feature does not make any sense, since every file should have one class, unless there is a missing link between revision and SOM file. This means this features was just chosen due to missing values. To solve this problem either the degree features are prohibited or only target revisions with a SOM file are considered. This is tried out in the next two experiments.

Table 5.7: Result Experiment 4

Model	ACC	AUC
RBC	0.7865	0.8525
RDN	0.8539	0.9234
RPT	0.8539	0.9300

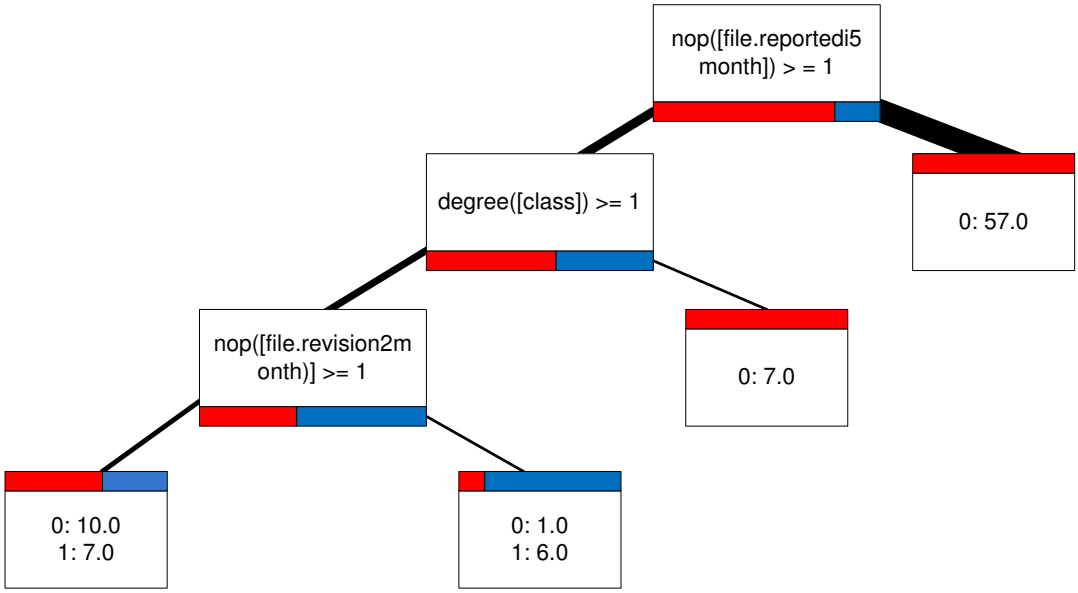


Figure 5.12: RPT Experiment 4

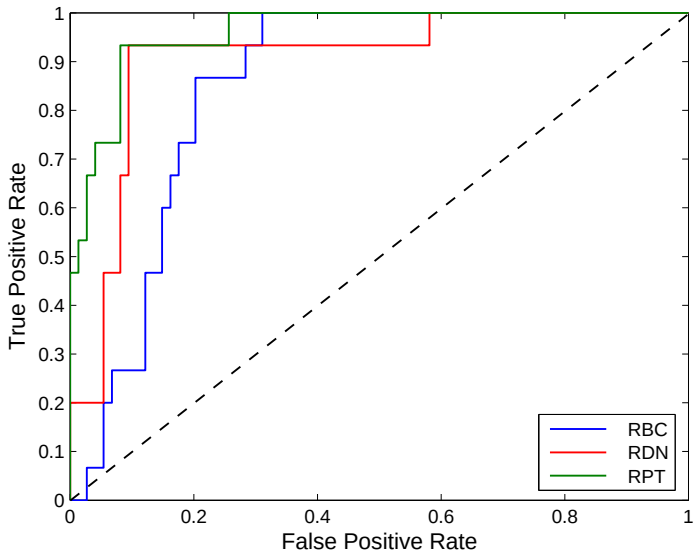


Figure 5.13: ROC HasIssue Experiment 4

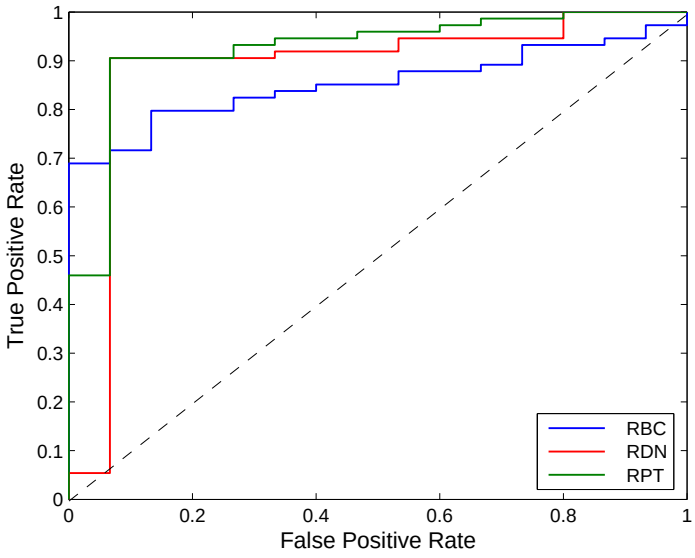


Figure 5.14: ROC HasNoIssue Experiment 4

5.4.5 Experiment 5

As in the last experiment explained, the generated feature `degree[class]` does not make sense. That is why the same experiment is done again, but this time we prohibit PROXIMITY to create features with the aggregator `degree`.

Features	Significant COM features (2,3,8,11,16,19 in Table 5.1)
Time range	2006/07
Files	Files with revision
Revisions	All target revisions
Aggregators	All, without `degree`

Result This results to the same decision tree as in experiment 4 without the feature `degree[class]` (see Figure 5.15). As RBC does not use the degree aggregator the result remains the same as in the experiment before (see Table 5.8). While RDN could keep its performance, the result of RPT is slightly worse. One strength of RPT is to build small trees. However, in this case the question arises whether this tree is not too small. This is tested later in experiment 10 and 11. It is not possible to prohibit the generation of the degree feature just on the entity "class". Therefore, we will not apply this solution to avoid the feature `degree[class]` in further experiments.

Table 5.8: Result Experiment 5

Model	ACC	AUC
RBC	0.7865	0.8524
RDN	0.8539	0.9225
RPT	0.8539	0.8952

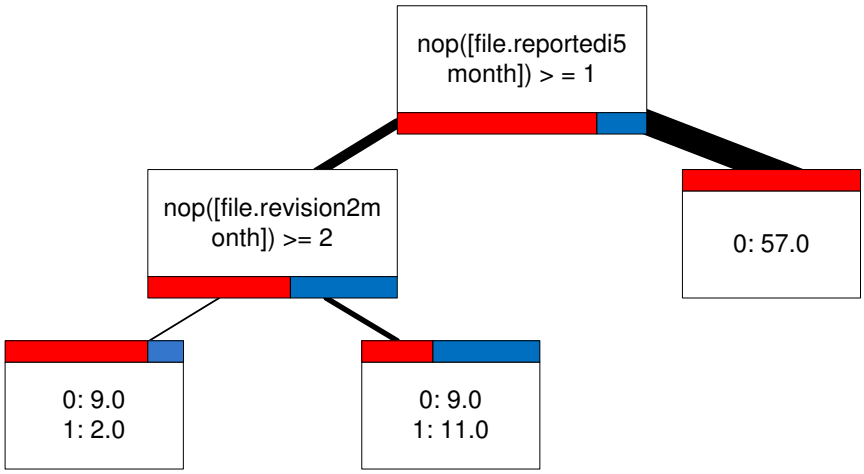


Figure 5.15: RPT Experiment 5

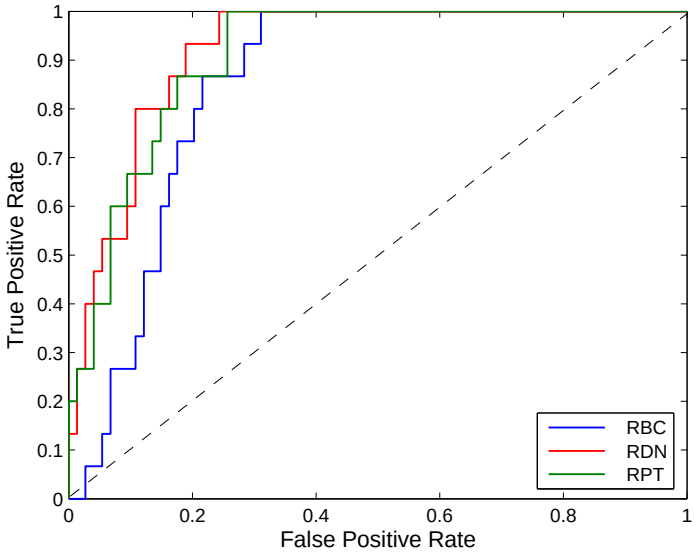


Figure 5.16: ROC HasIssue Experiment 5

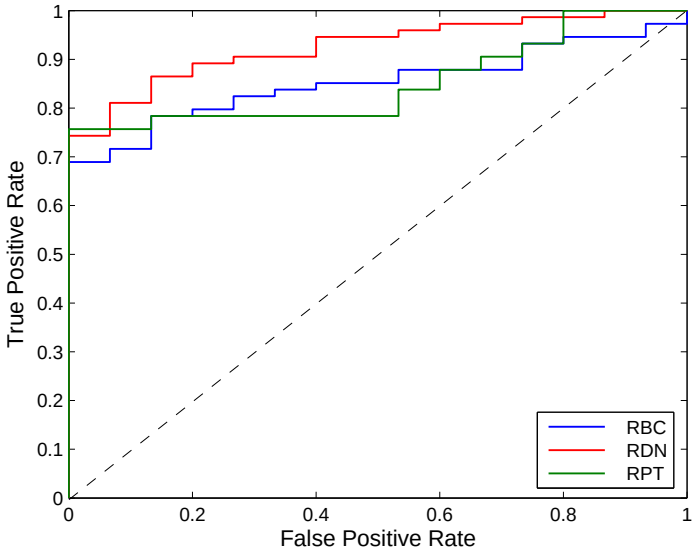


Figure 5.17: ROC HasNoIssue Experiment 5

5.4.6 Experiment 6

Experiment 4 is repeated again, this time only with target revisions that are linked to a SOM file in order to get rid of the feature `degree[class]`.

Features	Significant COM features (2,3,8,11,16,19 in Table 5.1)
Time range	2006/07
Files	Files with revision
Revisions	Revisions with SOM file
Aggregators	All

Result The same tree as in the experiment before was build. The AUC is slightly higher than in the comparison study but the ACC is smaller. As the results are similar to the results in experiment 4 we prefer this solution to the problem of the feature degree[class] as the solution in experiment 5 prohibits the degree feature for all entities.

Table 5.9: Result Experiment 6

Model	ACC	AUC
RBC	0.8333	0.9084
RDN	0.8000	0.9550
RPT	0.7999	0.9286

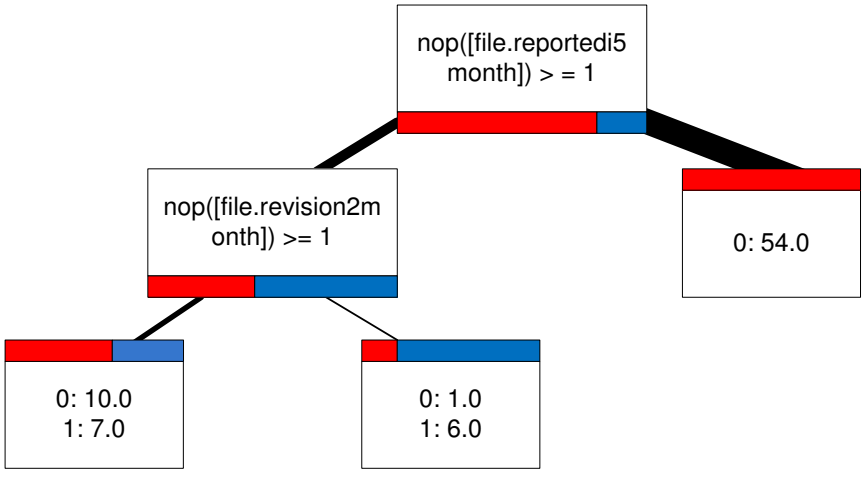


Figure 5.18: RPT Experiment 6

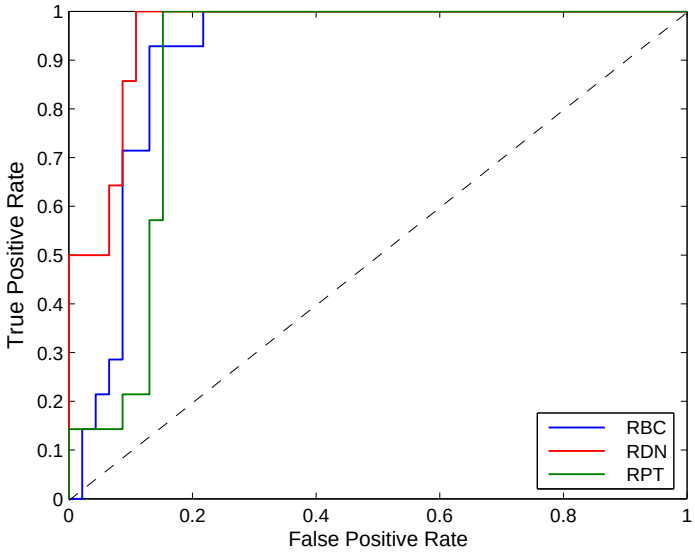


Figure 5.19: ROC HasIssue Experiment 6

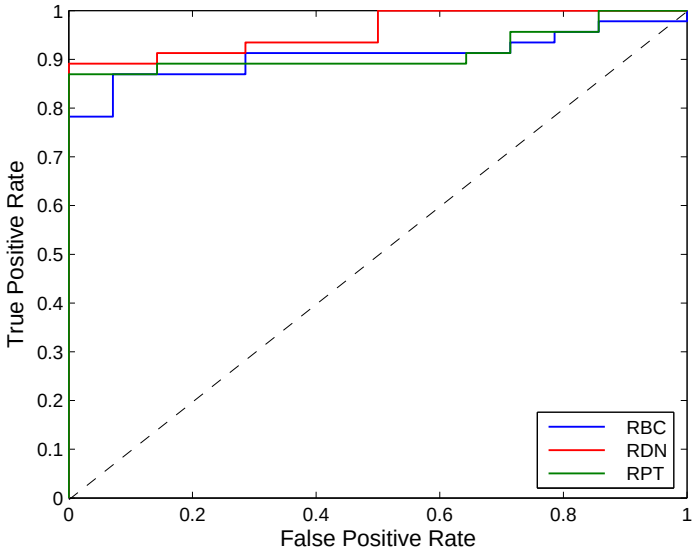


Figure 5.20: ROC HasNoIssue Experiment 6

5.4.7 Experiment 7

So far we have seen that using all COM features combined with all SOM features or only SOM features results in bad results. In this experiment it is tested whether there exists a combination of COM and SOM features that beat the results of experiments where only COM features are used. That is why, first all COM features and some aggregated SOM features were exported to WEKA to find the best features with the feature selection algorithm `Ranker` (Output of WEKA is shown in Appendix C). Using this selected features we started a new PROXIMITY experiment. As there are many target revisions with no SOM file we decided to use only files that have a target revision with a SOM file. Otherwise the SOM features have a smaller chance to be selected.

Features	2, 3, 7, 12, 13, 15, 16, 18, 19, 20 from Table 5.1 1, 8, 9, 11 from Table 5.2
Time range	2006/07
Files	Files with revision
Revisions	Target revisions with SOM file
Aggregators	All

Result This time the results of the AUC (see Table 5.10) exceed the results achieved in the comparison study. However, the ACC is still worse. Again a COM feature is chosen as the top node (see Figure 5.23). In comparison to experiment 6 where there were the same constraints like in this experiment but with the difference that only COM features were used, slightly better results due to the combination of COM and SOM features were achieved in this experiment. Especially, the ACC was improved. In this experiment it is shown that the right combination of both sorts of features is able to improve the results. However, the top node is still dominated by a temporal feature.

A totally different remark we want to add here: In different studies it is proved that collectively inferencing gives better results than predicting each instance separately. In our case the class label is ”hasissue” which we attached as an attribute to the revisions. In every subgraph created, the information whether the previous revision had an issue or not is available in an attribute. However, this feature was never selected in the experiments conducted so far. On one hand, this means that all methods whose strength is collectively inferencing (for example RDN and RMN) cannot improve the results. On the other hand, knowing whether a file had a defect in the revision before does not help at all predicting bugs.

Table 5.10: Result Experiment 7

Model	ACC	AUC
RBC	0.9000	0.9549
RDN	0.9166	0.9534
RPT	0.9166	0.9442

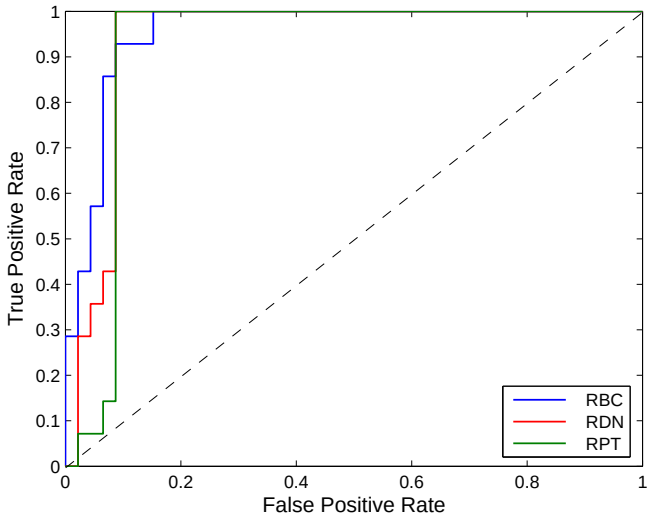


Figure 5.21: ROC HasIssue Experiment 7

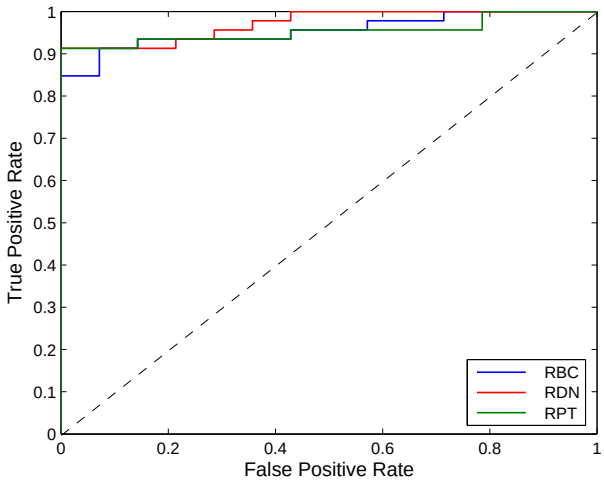


Figure 5.22: ROC HasNoIssue Experiment 7

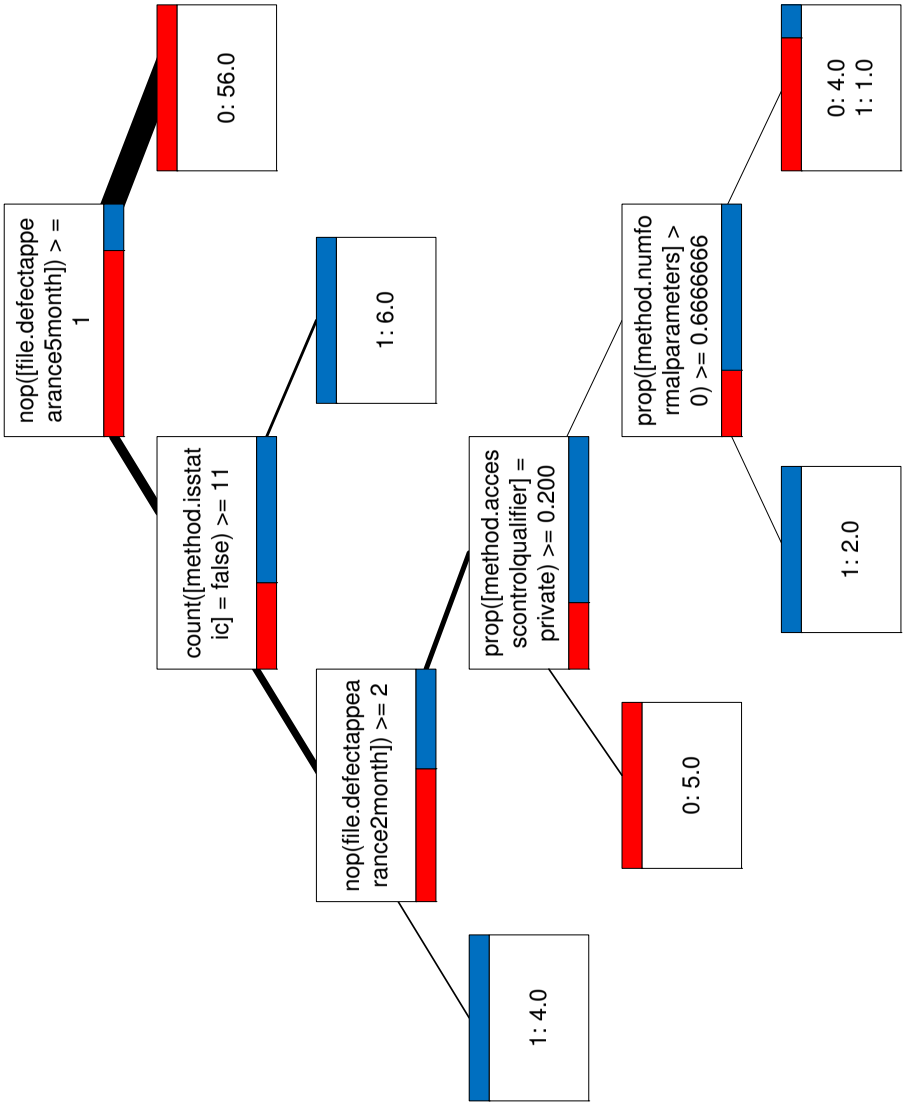


Figure 5.23: RPT Experiment 7

5.4.8 Experiment 8

The same experiment as in experiment 7 is conducted again with the difference that some more SOM features are added to know if this worsens the result.

Features	2, 3, 7, 12, 13, 15, 16, 18, 19, 20 from Table 5.1 1, 2, 5, 6, 8, 11, 12 from Table 5.2
Time range	2006/07
Files	Files with revision
Revisions	Target revisions with SOM file
Aggregators	All

Result The results of AUC and ACC are about the same as in experiment 7 although not the same SOM features were selected (see Figure 5.26). This indicates that it does not exist a that important SOM feature that is chosen in every case. The COM feature `nop(file.defectappearance2month]) >= 2` was dropped. Although, so far no very significant SOM features was found, it points that a combination of COM and SOM features perform better than COM features alone, since the results in the comparable experiment 6 are worse.

Table 5.11: Result Experiment 8

Model	ACC	AUC
RBC	0.9000	0.9549
RDN	0.9333	0.9611
RPT	0.9333	0.9437

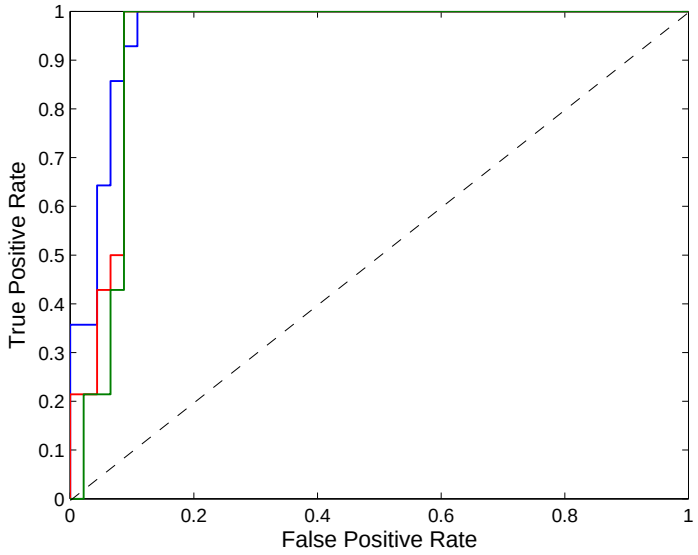


Figure 5.24: ROC HasIssue Experiment 8

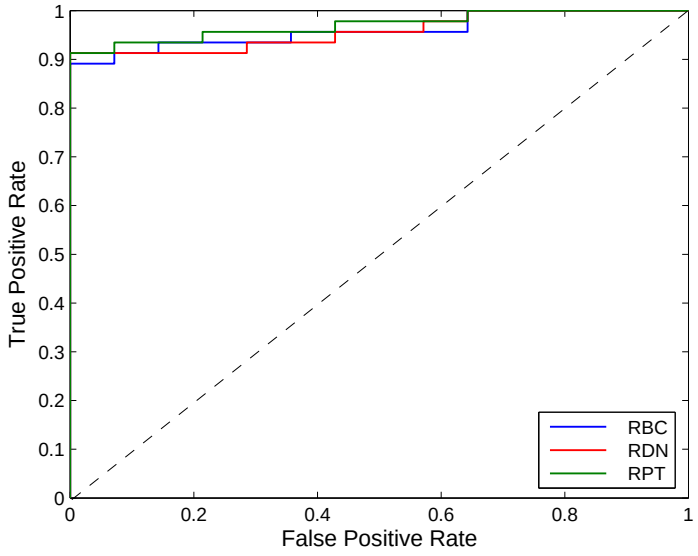


Figure 5.25: ROC HasNoIssue Experiment 8

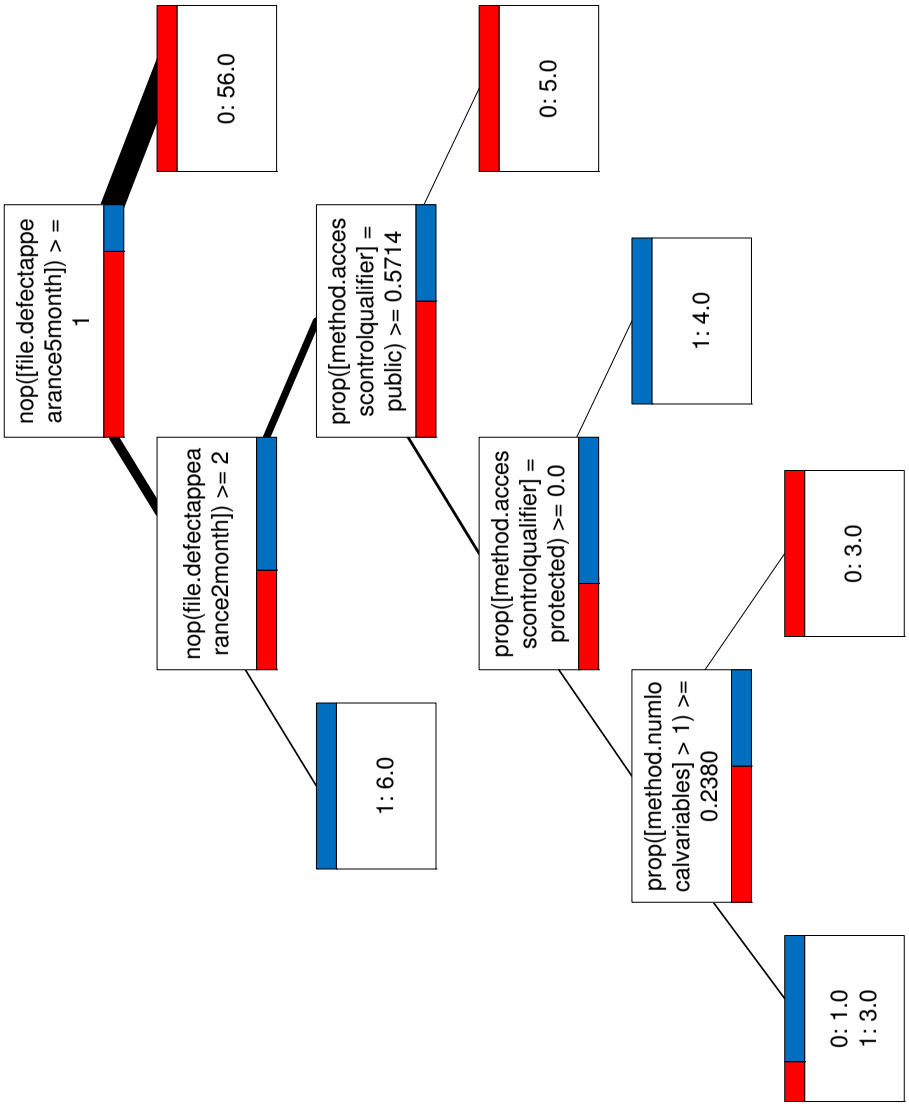


Figure 5.26: RPT Experiment 8

5.4.9 Experiment 9

In this experiment the time range 2007 was tested. As in experiment 7 some features were, if necessary, aggregated and exported to WEKA to rank the features (for the output see Appendix C.2). As only 24 of 161 files have at least one revision in the time range, we decided also to consider those files without a revision in that time range. All target revisions are connected to a SOM file.

Features	2, 3, 7, 12, 13, 14, 15, 16, 17, 18, 19, 20 of Table 5.1 1, 10, 11 of Table 5.2
Time range	2007
Files	All target revisions
Revisions	All files
Aggregators	All

Result The result of AUC and ACC in the case of RBC is very high (see Table 5.12). The decision tree in Figure 5.29 is again quite small and no SOM features were chosen at all. This means that all other features were dropped by RPT as they were not significant enough. This is not the case for RBC as it does not calculate any dependency levels, but only conditional probabilities. In this experiment it is shown that the method of RBC is better than the one of RDN and RPT. It might be that the pValue, which is the threshold to decide whether a feature is significant or not, is too restrictive. As Neville et al. (see [Neville et al., 2003a]) state that the current implementation of the pValue might need some more exploration, in the next experiment this value is slightly changed.

Table 5.12: Result Experiment 9

Model	ACC	AUC
RBC	0.9627	0.9771
RDN	0.9316	0.8735
RPT	0.9316	0.9009

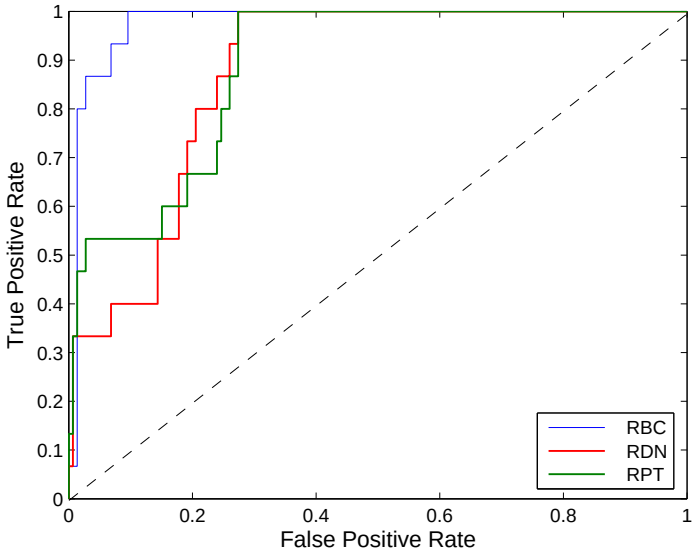


Figure 5.27: ROC HasIssue Experiment 9

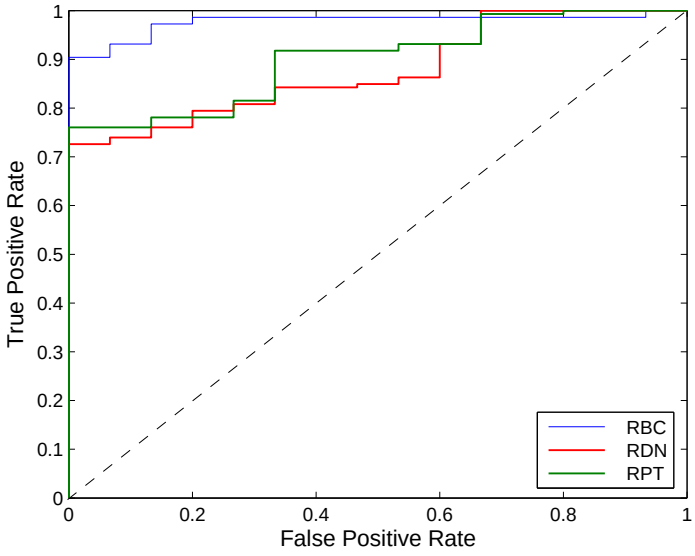


Figure 5.28: ROC HasNoIssue Experiment 9

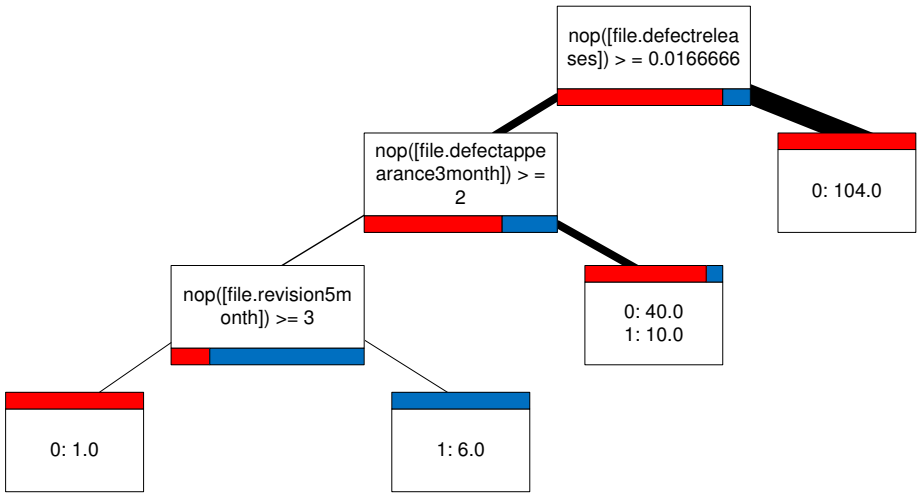


Figure 5.29: RPT Experiment 9

5.4.10 Experiment 10

As the tree of the experiment before is quite small and no SOM features at all were chosen, we decided to increase the pValue to 0.06 (see Appendix A.1). This means, it is easier to depend on the class label and the tree should become larger.

Features	2, 3, 7, 12, 13, 14, 15, 16, 17, 18, 19, 20 of Table 5.1
	1, 10, 11 of Table 5.2
Time range	2007
Files	All target revisions
Revisions	All files
Aggregators	All

Result As expected, the tree became larger and the performance of RPT and RDN is better (see Table 5.13). However, it seems that the resulted decision tree is overfitted. As an example see Figure 5.32 where the node `prop([method.numformalparameters] >1) >= 0.4` is found. If this condition is true no issue is expected. Only one instance passes the left (true) branch. This experiment showed that increasing the pValue was not useful and not more significant features can be found this data set.

Table 5.13: Result Experiment 10

Model	ACC	AUC
RBC	0.9440	0.9671
RDN	0.9440	0.9511
RPT	0.9565	0.9700

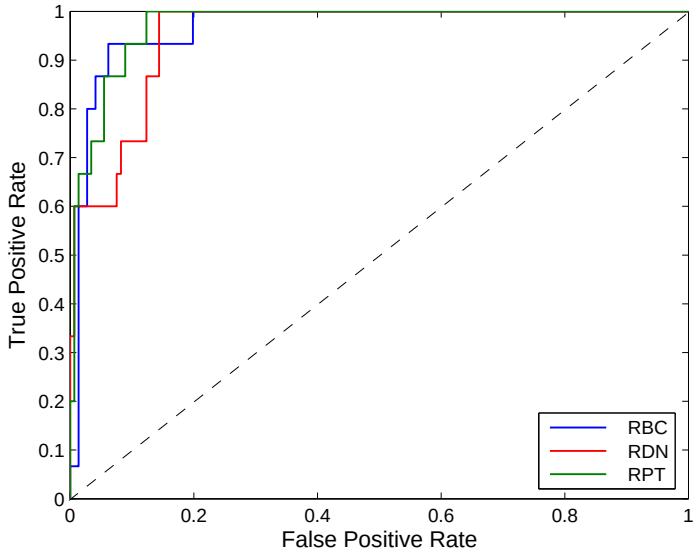


Figure 5.30: ROC HasIssue Experiment 10

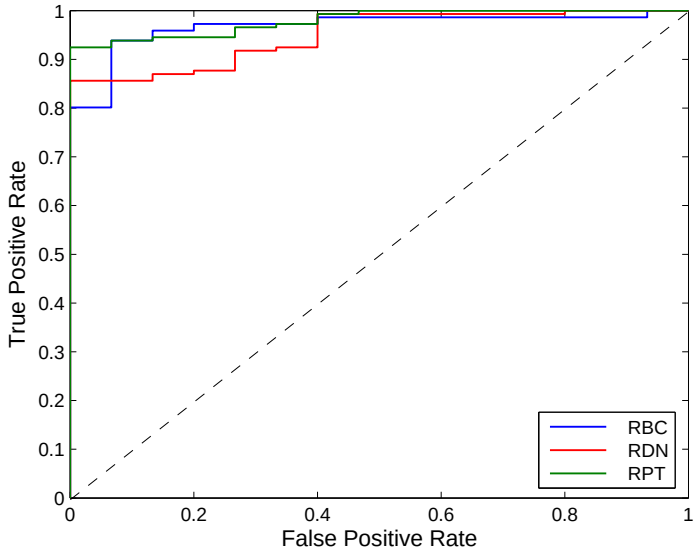


Figure 5.31: ROC HasNoIssue Experiment 10

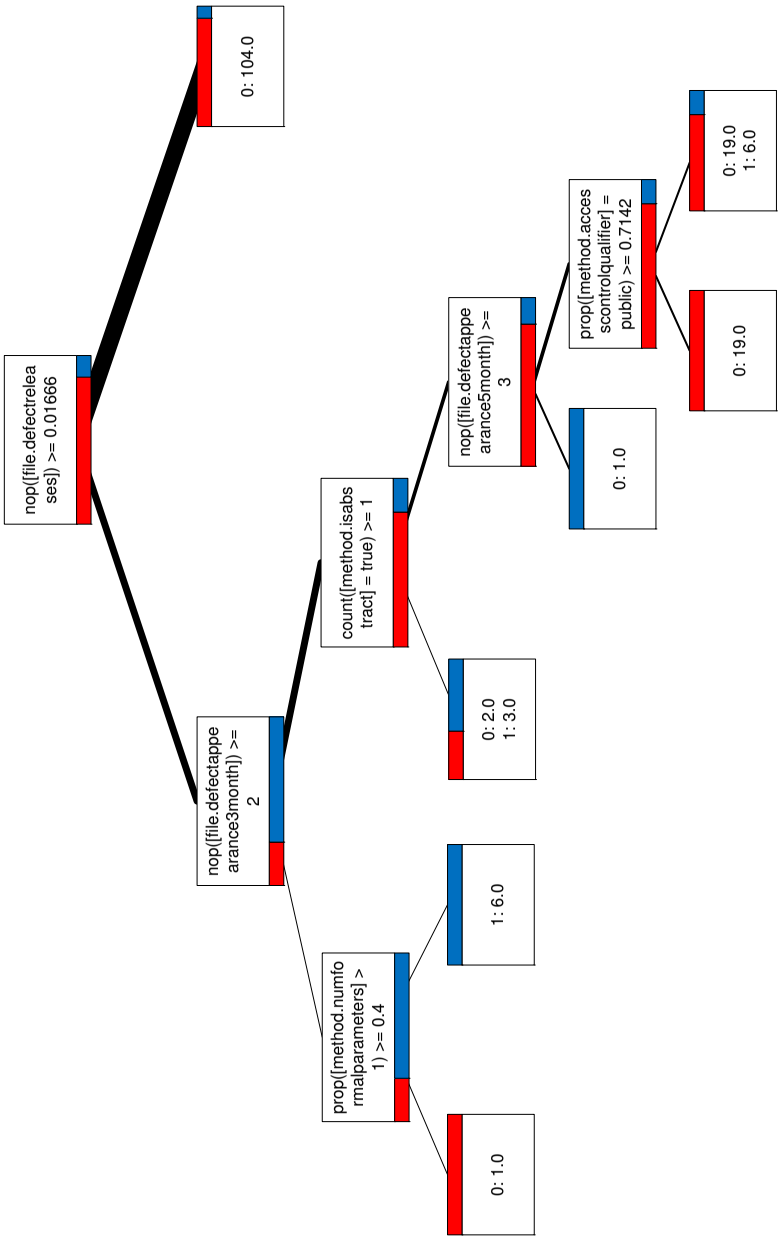


Figure 5.32: RPT Experiment 10

5.4.11 Experiment 11

In experiment 10, by increasing the pValue a higher ACC and AUC was achieved, but the tree was overfitted. Nevertheless, we want to test the same experiment on the data set 2006/07.

Features	2, 3, 7, 12, 13, 14, 15, 16, 17, 18, 19, 20 of Table 5.1 1, 2, 5, 6, 8, 11, 12 of Table 5.2
Time range	2006/07
Files	Files with revision
Revisions	Target revisions with SOM file
Aggregators	All

Result The decision tree (see Figure 5.35) is smaller than in the experiment before and the created nodes in the tree are also meaningful. However, compared to the achieved results in experiment 8, the performance is now worse (see Table 5.14) and increasing the pValue did not help to improve the performance as well here.

Table 5.14: Result Experiment 11

Model	ACC	AUC
RBC	0.9000	0.9503
RDN	0.9166	0.9347
RPT	0.9166	0.9436

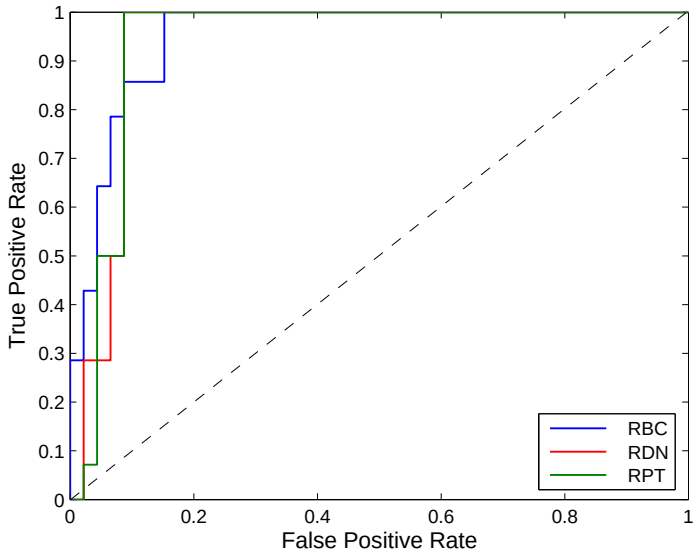


Figure 5.33: ROC HasIssue Experiment 11

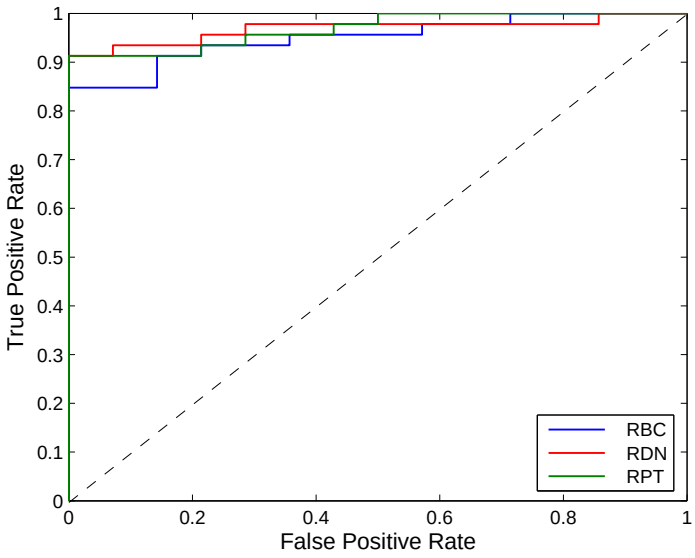


Figure 5.34: ROC HasNoIssue Experiment 11

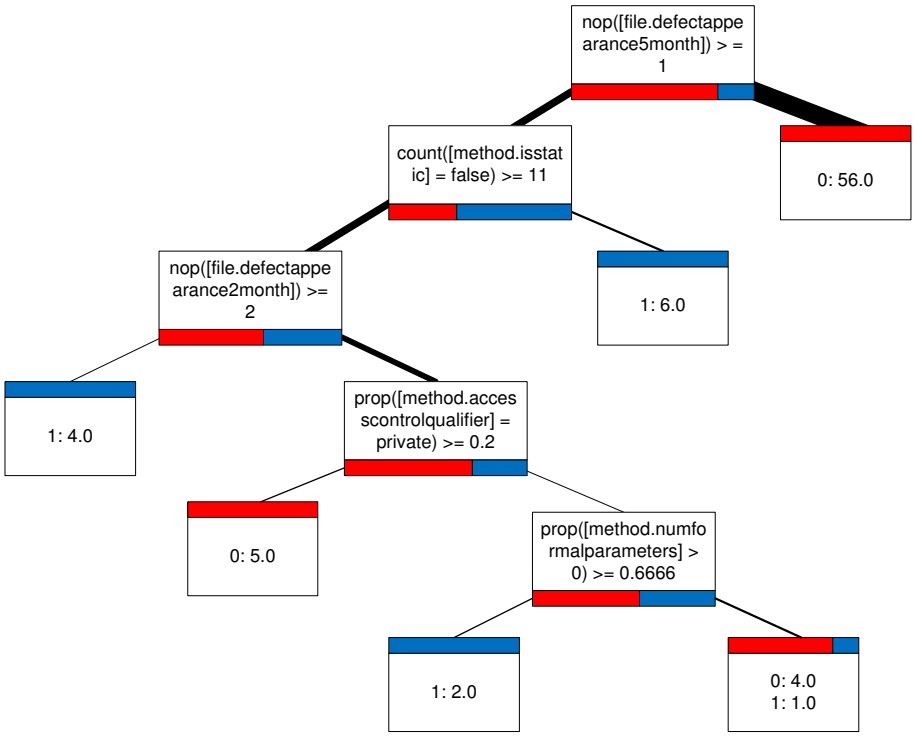


Figure 5.35: RPT Experiment 11

5.4.12 Experiment 12

As in various experiments so far conducted either the feature `reportedI5month` or `defectappearance5month` were the top node of the decision tree, in this experiment the time span is extended to seven, ten and twenty months back.

Features	2, 3, 7, 12, 13, 14, 15, 16, 17, 18, 19, 20 of Table 5.1
	1, 10, 11 of Table 5.2
	plus the new features: `revision7month`, `reprotedI7month`, `defectappearance7month`
	`revision10month`, `reprotedI10month`, `defectappearance10month`
	`revision20month`, `reprotedI20month`, `defectappearance20month`

Time range	2007
Files	All target revisions
Revisions	All files
Aggregators	All

Result This time, there is a new top node selected, namely `defectappearance10month`. The feature `defectappearance20month` is chosen as well, but it does not belong to the most important features. This means that looking even more than twenty months back will not improve the classification result. Again no SOM features were significant. The feature `defectAppearanceXmonth` is similar to the feature `reportedIXmonth`. The difference is that in the former the number of defect releases and in the latter the number of defect revisions is counted. It might be thought that this is not an important difference, however, in this experiment and also in the experiments before the number of defect releases was more significant. The frequent appearance of the feature `defectXappearance` implies that when a released file is bug free for some months it is very likely to be stable in the future.

Table 5.15: Result Experiment 12

Model	ACC	AUC
RBC	0.9503	0.9803
RDN	0.9006	0.9621
RPT	0.9006	0.9496

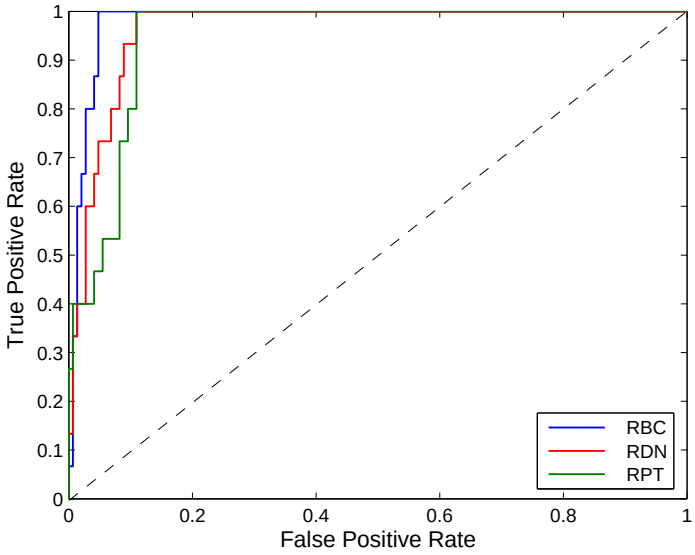


Figure 5.36: ROC HasIssue Experiment 12

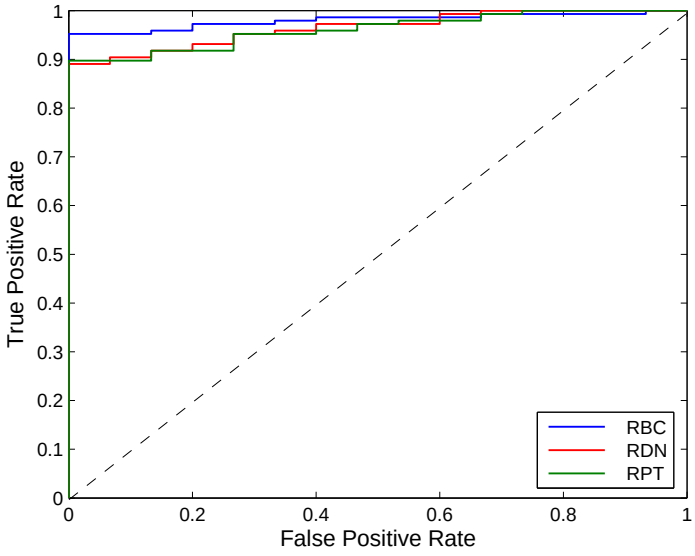


Figure 5.37: ROC HasNoIssue Experiment 12

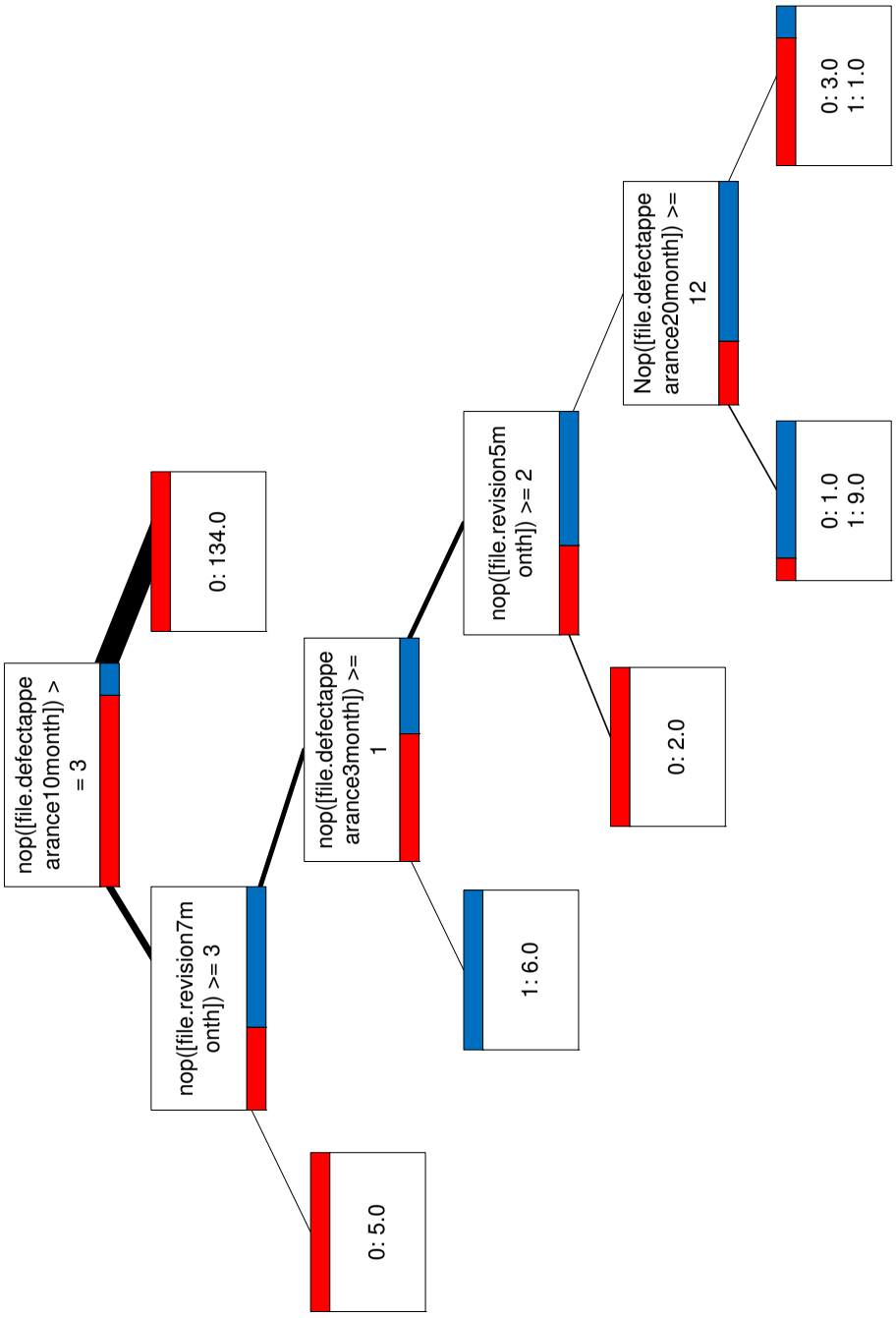


Figure 5.38: RPT Experiment 12

5.4.13 Experiment 13

In this experiment the same temporal features as in the experiment before are added, but applying it on the time range 2006/07.

Features	2, 3, 7, 12, 13, 14, 15, 16, 17, 18, 19, 20 of Table 5.1
	1, 8, 9, 11 of Table 5.2
	plus the new features: `revision7month`, `reprotedI7month`, `defectappearance7month`
	`revision10month`, `reprotedI10month`, `defectappearance10month`
	`revision20month`, `reprotedI20month`, `defectappearance20month`

Time range	2006/07
Files	Target revisions with SOM file
Revisions	File with revision
Aggregators	All

Result This time, the new features with the extended time period did not have any influence. Remember that the difference between the data set 2006/07 and 2007 is that in 2006/07 only files with at least one revision during the time range were considered. Thus, the reason why the new features did not influence the results in this experiment, might be that there is more current information available than it is in the phase 2007. This implies that older information gets pointless in case newer information exists. However, this contradicts the fact that in the experiment before better results were achieved. This shows that more studies have to be done on this issue.

Table 5.16: Result Experiment 13

Model	ACC	AUC
RBC	0.8666	0.9557
RDN	0.9333	0.9425
RPT	0.9333	0.9456

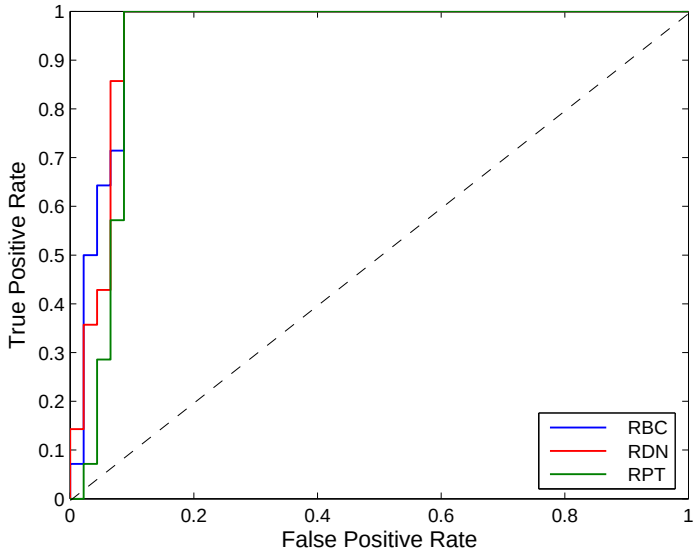


Figure 5.39: ROC HasIssue Experiment 13

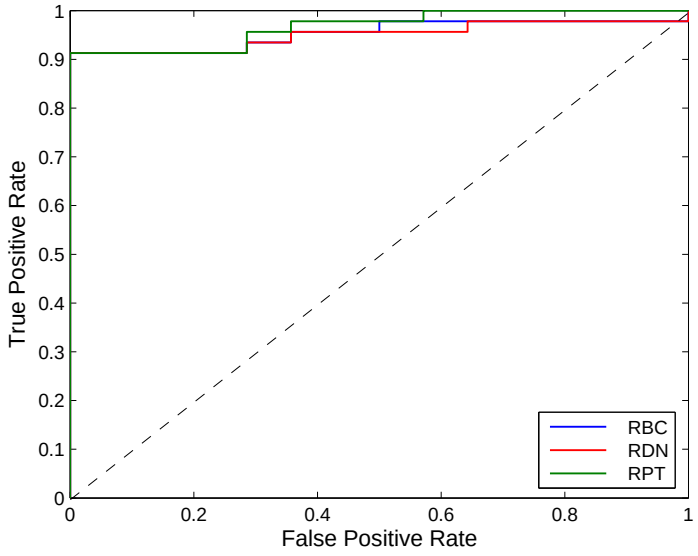


Figure 5.40: ROC HasNoIssue Experiment 13

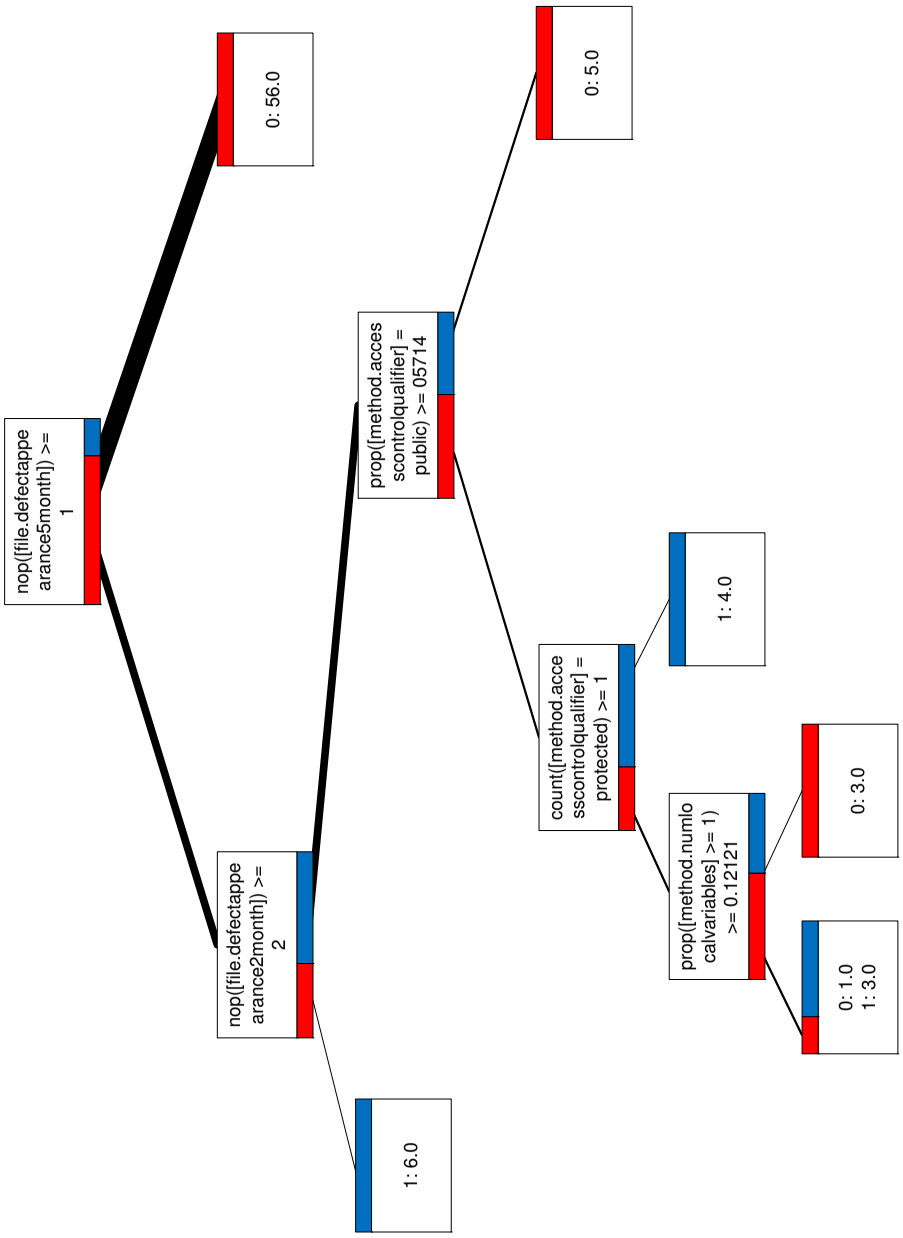


Figure 5.41: RPT Experiment 13

5.4.14 Summary

Relational data mining methods implemented in PROXIMITY were evaluated on two data sets. In the first test set (2006/07), only files with at least one revision during the time range were considered. In the second test set (2007), all files were considered, since the number of files having at least one revision during the time range was too small.

In all experiments where only significant features were provided, the value for the AUC was higher than in the comparison study. However, the value for the ACC was clearly smaller as in the comparison study. Even though these results are promising, it has to be done more work for finding better source code features.

For both test sets it can be stated that for predicting the location of bugs, temporal features play the main role, more precisely it depends on how many defect releases there were in the past and how many revisions were done. Files with no defect releases within the last five months are very likely to have no defects in the future. When a file had defect releases in the past but also revisions were done, it is less likely to have defects in the future than a file that had defect releases but only a few or no revisions. Furthermore, it was discovered that COM features alone can achieve good results, but SOM features alone have no chance. By providing all features available and letting the classifier choose resulted in bad results. Although RPT and RDN are selective classifiers, they achieved a worse performance than the RBC. In all experiments where only significant features were provided, a small decision tree was created. Assuming that these trees might be too small for a proper classification the pValue was increased by 0.01. However, the results in experiments with artificially enlarged trees had either a poorer performance than the smaller trees or were overfitted.

There were also differences observed between the two data sets. For the data set 2006/07 the results were slightly better if some selected SOM features were added. This in contrast to the data set 2007 where no SOM features were selected if the pValue was set to 0.05. In the data set 2006/07 better results with SOM features were achieved, but no SOM feature was so important that its replacement by another SOM feature had a big influence on the result. So, any combination of SOM features helped to improve the overall result. Since in many experiments temporal features that consider the last five months were chosen, new temporal features with an extended period were added (`revisionXmonth`, `reportedIXmonth` and `defectappearancesXmonth`, where X is 7, 10 or 20). In the data set 2006/07 these new features did not have any influence. This in contrast to the data set 2007. There, the feature `defectappearance10month` was selected as the top splitting node and `revision7month` was selected as the second node. The feature `defectappearance20month` appeared as well in the tree but played a minor role. The fact that in the data set 2006/07 the new features do not play a role and in 2007 it does, makes sense. In 2007 more files did not have any revision in the last five months, meaning that for those temporal features the value is set to zero and that is why older information is used for predicting.

6

Conclusions

In this thesis software repositories were mined with the data mining tool PROXIMITY in order to predict the location of bugs. This was already tried with various other approaches, however never before with relational data mining methods. The goal was to test, whether a relational combination of revision, bug and source code data yields a better performance than traditional approaches. To measure our approach, we compared the results with a successful traditional study.

First, common relational data mining models and different data mining tools were introduced. Then, we presented the data for our experiments and the necessary data preparations. In the evaluation we showed that if a good combination of temporal and source code features is available, a higher AUC value is achieved compared to the the traditional study mentioned above. The evaluation also reveals different weaknesses. Using only source code features led to very bad results. By providing all available features and letting the classifier choose, resulted in poor results although the classifier itself is selective. In addition, the very high accuracy achieved in the comparison study could not be reached by this approach.

6.1 Limitations

One big limitation of our approach is that we only tested the `compare` plugin of the Eclipse project.

All of our preparation steps were run in memory which used up to 11GB of main memory. For further experiments with more plugins we would have to consider the memory shortage.

The WEKA Ranker algorithm was applied to find the best features. As WEKA only handles propositional data, we first had to aggregate the data with MAL queries. This is time intensive and time was tight and so we did not aggregate every feature with all of the seven aggregators provided by PROXIMITY . This implies that the ranking of the features might be suboptimal since not all possible versions were available.

The revisions with no assigned link of the name `hasNextRevision` were marked as target revisions in order to define the actual source code state of a file. In CVS it is possible to branch revisions, meaning that it is possible that there exist several revisions for a file without the link `hasNextRevision`. In this case only the revision with the latest date was considered.

6.2 Future Work

This was just a first attempt at using relational data mining methods for predicting bugs from software data. Future studies could deal with the following issues:

- Significant features created in the comparison study plus the features already available in the source code were used. We tested the applicability of these features to relational models. A next step would be to find better source code features. In other studies, there were various metrics proposed for describing the complexity of source code. So, for starting, in addition to the features utilized in this thesis, other existing measures could be added and further improved.
- Only the `compare` plugin was considered. In future experiments more projects should be tested. Thus, it is examined whether the significant features are project specific or if they can be applied to other projects as well.
- In our opinion PROXIMITY seems to be the most user friendly tool for relational data mining. However, it might be worth investigating the other tools mentioned in this thesis as they cover other models which might lead to better results.

A

Algorithms

A.1 Chi Square Test of Independence

The Chi Square Test measures if two categorical (or categoricized) variables are dependent or not. For explanation an example is added here [1].

In this example there are two variables, `author` and `issue`. `Author` has the values "Mr. Bauer" and "Others", `issue` has the values "HasIssue" and "NoIssue". It should be find out whether Mr. Bauer is a worse programmer than the others.

Therefore, the two Hypotheses:

H_0: HasIssue and Mr. Bauer are independent
H_1: HasIssue and Mr. Bauer are dependent

In a contigency table the number of observations (file has issue/no issue and files programmed by Mr. Bauer/others) are summarized and summed up (Total).

Table A.1: Number of files with issues

	Mr. Bauer	Others	Total
HasIssue	5	10	15
NoIssue	10	80	90
Total	15	90	105

Putting the numbers of this example into the formula for χ^2 (see Equation A.1 [Fahrmeir et al., 2003]) results to the following value (Equation A.1):

$$\chi^2 = \sum_{i=1}^{k} \sum_{j=1}^{m} \frac{h_{ij} - \tilde{h}_{ij}}{\tilde{h}_{ij}} \quad , with \;\; \tilde{h}_{ij} = \frac{h_{i.} h_{.j}}{n} \tag{A.1}$$

$$\chi^2 = \frac{105[5*80 - 10*10]^2}{15*90*15*90} = 5.18$$

[1] `http://math.hws.edu/javamath/ryan/ChiSquare.html`, visited 2008-01-30

Assuming the significance level to be 0.05 (called pValue in Proximity) whose probability level is 3.841 (from Chi square table). In this examples 5.18 exceeds 3.841 which means H_0 is rejected and H_1 is accepted.

For splitting, PROXIMITY chooses the features with the highest score, this is the highest χ^2 value, the one that depends mostly on the class label (issue). If χ^2 were smaller than 3.841 Mr. Bauer would not depend on issue and this feature would be dropped.

A.2 MCMC/Gibbs Sampling

Gibbs sampling is a simplified form of Markov Chain Monte Carlo Methods [Neal, 1993]. It is is a very popular inference algorithm for estimating joint distributions. Approximations are needed because calculating joint distributions often leads to computational difficulties. Gibbs Sampling as well as MCMC methods are based on sampling Markov Chains. When a Markov Chain is long enough it can be used to estimate a distribution. In Markov Chains variables depend only on the current state, everthing before can be forgotten.

As Neal [Neal, 1993] explains, the main procedure of generating a Markov Chain, done by Gibbs Samplers, works as follows:

$$\begin{aligned}
&\text{Pick } X_1^{(t)} \text{ from distribution for } X_1 \text{ given } X_2^{t-1}, X_3^{t-1}, ..., X_n^{t-1} \\
&\text{Pick } X_2^{(t)} \text{ from distribution for } X_2 \text{ given } X_1^{t-1}, X_3^{t-1}, ..., X_n^{t-1} \\
&... \\
&\text{Pick } X_n^{(t)} \text{ from distribution for } X_n \text{ given } X_2^{t-1}, X_3^{t-1}, ..., X_{n-1}^{t-1}
\end{aligned} \tag{A.2}$$

Every variable X_i is updated by picking it from the conditional distribution $P(X_i|X_{other})$. The challenge is to find suitable starting values for the variables X_i (referenz suchen). This procedure is repeated several times, in our experiments 2000 times.

For a more formal explanation of MCMC/Gibbs sampling methods [Neal, 1993] is a good reference.

B

Data Preparation Steps

One way of preparing the data for PROXIMITY is explained here, started from the separated OWL files.

Note: Step 3 to 7 has to be run twice, once for the training data and once for the testing data.

1. Merge OWL files: run DataMerge.java, run with ca. -Xms10000m -Xmx10000m
2. SPARQL queries: ReducedQueries.java, run it with ca. -Xms11000m -Xmx11000m
3. Data Transformation: DataTransformation.java, ca. -Xms2000m -Xmx2000m
4. Generating Proximity XML file: perl script by Proximity: text2xml.pl
5. Import Data: import-text.sh, provided by Proximity
6. Creating Subgraphs: Run python script: createSubgraphs.py, name it learn respectively apply
7. Generate features: COM features: comfeatues.py, SOM features: somfeatures.py
8. Run Mining Methods
   (a) learning scripts: learn_rbc.py/learn_rpt.py/learn_rdn.py
   (b) switch to the apply database
   (c) applying scripts: apply_rbc.py/apply_rpt.py/apply_rdn.py
9. Output features for WEKA: run according outputfeature script: e.g. revision1monthOut.py (ARFF file has to be created by yourself)

C

Weka Outputs

C.1 Output Weka Ranker Experiment 7

```
Ranked attributes:
 0.3747   28 defectappearance5month
 0.3439    8 reportedi5month
 0.3299   24 defectreleases
 0.3251   27 defectappearance3month
 0.2769   30 lineAddedIRLAdd
 0.2502   29 reportedissues
 0.2493    6 reportedi3month
 0.2253   26 defectappearance2month
 0.221    31 lineDeletedIRLDEL
 0.1617    4 reportedi2month
 0.1086   15 numlocalvariables1
 0.1053   14 numformalparameters0
 0.0369   10 modeaccesscontrolqualifier
 0.0118   16 prevrevisionwithissue
 0        23 releases
 0         5 revision3month
 0        22 propprotectedmethod
 0         7 revision5month
 0         2 reportedi1month
 0         1 revision1month
 0        25 defectappearance1month
 0         3 revision2month
 0        19 numforeignmethodsinvoked
 0        20 nummethodsinvoked
 0        17 invclasswissue
 0        18 numattributes
 0        21 proplocalvariable
```

```
 0         9 revisionauthor
 0        13 countisstaticfalse
 0        11 avgnumformalparam
 0        12 countforeignmethinvoked0
```

Listing C.1: Weka Output Experiment 7

C.2 Output Weka Ranker Experiment 9

```
Ranked attributes:
 0.17297   30 defectreleases
 0.15523   35 reportedissues
 0.10964   33 defectappearance3month
 0.10666   34 defectappearance5month
 0.10358   36 lineAddedIRLAdd
 0.09734    9 reportedi5month
 0.09566    7 reportedi3month
 0.08426   37 lineDeletedIRLDEL
 0.07133   24 nummethodsinvoked
 0.06451   32 defectappearance2month
 0.05865    6 revision3month
 0.05814    8 revision5month
 0.04513    5 reportedi2month
 0.00969   11 accesscontrolqualifier
 0.00482   19 prevrevwithissue
 0         27 proppublicmeth
 0         28 sumnumformalparameters
 0         25 proplocalvariable
 0         26 propprotectedmethods
 0         10 revisionauthor
 0          3 reportedi1month
 0          2 revision1month
 0         29 releases
 0          4 revision2month
 0         31 defectappearance1month
 0         23 nummethods
 0         18 numlocalvar1
 0         22 numforeignmethinvoked
 0         20 countinvclasswissue
 0         21 numattributes
 0         14 numforeignmethodinvoked0
 0         12 avgnumformalparameters
 0         13 countisstatictrue
 0         17 numformalparameter0
```

```
0          15 countisinterfacefalse
0          16 countisstaticfalse
```

Listing C.2: Weka Output Experiment 9

D

CD

On this CD all files produced in this thesis are available.

- PDF and Latex source code
- Java Source Code
- Proximity Scripts
- Protocols of conducted experiments

List of Figures

2.1 Steps in the KDD process . . . 5
2.2 Schema RBM . . . 9
2.3 Relational Skeleton . . . 9
2.4 Dependency Structure . . . 9

3.1 QGraph query . . . 12
3.2 Contents of a Subgraph in Text Form . . . 12
3.3 Contents of a Subgraph as a Graph . . . 13
3.4 Data Flattened [Neville et al., 2003b] . . . 14
3.5 Relational Probability Tree output . . . 15
3.6 Dependency Graph . . . 15

4.1 Data Preparation . . . 20
4.2 Software Ontology Model . . . 21
4.3 Version Ontology Model . . . 22
4.4 Bug Ontology Model . . . 22
4.5 Interconnection of the Three Ontology Models . . . 22
4.6 QGraph Query . . . 27

5.1 Time Range . . . 31
5.2 RPT . . . 33
5.3 RPT . . . 35
5.4 ROC HasIssue Experiment 1 . . . 35
5.5 ROC HasNoIssue Experiment 1 . . . 36
5.6 ROC HasIssue Experiment 2 . . . 37
5.7 ROC HasNoIssue Experiment 2 . . . 38
5.8 RPT Experiment 2 . . . 39
5.9 ROC HasIssue Experiment 3 . . . 41
5.10 ROC HasNoIssue Experiment 3 . . . 41
5.11 RPT Experiment 3 . . . 42
5.12 RPT Experiment 4 . . . 44
5.13 ROC HasIssue Experiment 4 . . . 44
5.14 ROC HasNoIssue Experiment 4 . . . 45

5.15 RPT Experiment 5 46
5.16 ROC HasIssue Experiment 5 47
5.17 ROC HasNoIssue Experiment 5 47
5.18 RPT Experiment 6 48
5.19 ROC HasIssue Experiment 6 49
5.20 ROC HasNoIssue Experiment 6 49
5.21 ROC HasIssue Experiment 7 51
5.22 ROC HasNoIssue Experiment 7 51
5.23 RPT Experiment 7 52
5.24 ROC HasIssue Experiment 8 54
5.25 ROC HasNoIssue Experiment 8 54
5.26 RPT Experiment 8 55
5.27 ROC HasIssue Experiment 9 57
5.28 ROC HasNoIssue Experiment 9 57
5.29 RPT Experiment 9 58
5.30 ROC HasIssue Experiment 10 60
5.31 ROC HasNoIssue Experiment 10 60
5.32 RPT Experiment 10 61
5.33 ROC HasIssue Experiment 11 62
5.34 ROC HasNoIssue Experiment 11 63
5.35 RPT Experiment 11 63
5.36 ROC HasIssue Experiment 12 65
5.37 ROC HasNoIssue Experiment 12 65
5.38 RPT Experiment 12 66
5.39 ROC HasIssue Experiment 13 68
5.40 ROC HasNoIssue Experiment 13 68
5.41 RPT Experiment 13 69

List of Tables

2.1 Customer table 7
2.2 Purchase table 7
2.3 Aggregation table 8

5.1 Features 30
5.2 Source Code Features 32
5.3 Distribution Data 32
5.4 Result Experiment 1 34
5.5 Result Experiment 2 37
5.6 Result Experiment 3 40
5.7 Result Experiment 4 43
5.8 Result Experiment 5 46
5.9 Result Experiment 6 48
5.10 Result Experiment 7 50
5.11 Result Experiment 8 53
5.12 Result Experiment 9 56
5.13 Result Experiment 10 59
5.14 Result Experiment 11 62
5.15 Result Experiment 12 64
5.16 Result Experiment 13 67

A.1 Number of files with issues 73

List of Listings

2.1 SQL Query for Cliques 10
3.1 Python Script 13
4.1 Merge Data 23
4.2 Define SPARQL Queries 24
4.3 Execute SPARQL Queries 24
4.4 Object Specification File 26
4.5 Link Specification File 26
4.6 MAL Query 28
4.7 Attribute Function 28
C.1 Weka Output Experiment 7 77
C.2 Weka Output Experiment 9 78

Bibliography

[Bernstein et al., 2007] Bernstein, A., Ekanayake, J., and Pinzger, M. (2007). Improving Defect Prediction Using Temporal Features and Non Linear Models.

[Blau et al., 2002] Blau, H., Immermann, N., and Jensen, D. (2002). A Visual Language for Querying and Updating Graphs.

[Domingos et al., 2006] Domingos, P., Kok, S., Poon, H., Richardson, M., and Singla, P. (2006). Unifying Logical and Statistical AI.

[Domingos and Richardson, 2007] Domingos, P. and Richardson, M. (2007). *Markov Logic: A Unifying Framework for Statistical Relational Learning*, volume Introduction to Statistical Relational Learning, pages 339–371. MIT Press, Cambridge.

[Džeroski, 2007] Džeroski, S. (2007). Inductive Logic Programming in a Nutshell. volume Introduction to Statistical Relational Learning, pages 58 – 92. MIT Press.

[Džeroski and Lavrač, 2001] Džeroski, S. and Lavrač, N. (2001). *Relational Data Mining*. Springer, Ljubljana.

[Fahrmeir et al., 2003] Fahrmeir, L., Knsteler, R., Pigeot, I., and Tutz, G. (2003). *Statistik: Der Weg zur Datenanalyse*. Springer, Berlin Heidelberg New York.

[Fayyad et al., 1996] Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. (1996). Mining to Knowledge Discovery: An Overview. *MIT Press*.

[Getoor et al., 2007] Getoor, L., Friedmann, N., Koller, D., Pfeffer, A., and Taskar, B. (2007). Probabilistic Relational Models. volume Introduction to Statistical Relational Learning, pages 129 – 174. MIT Press.

[Getoor and Taskar, 2007] Getoor, L. and Taskar, B. (2007). *Introduction to Statistical Relational Learning*. MIT Press.

[Graves et al., 2000] Graves, T., Karr, A., Marron, J., and Siy, H. (2000). Predicting Fault Incidence Using Software Change History.

[Halstead, 1977] Halstead, M. H. (1977). *Elements of Software Science, Operating, and Programming Systems*. Elsevier, Berlin Heidelberg New York.

[Hassan and Holt, 2005] Hassan, A. and Holt, R. (2005). The Top Ten List: Dynamic Fault Prediction.

[Ivanova et al., 2007] Ivanova, M., Nes, N., Goncalves, R., and Kersten, M. (2007). MonetDB/SQL Meets SkyServer: the Challenges of a Scientific Database.

[Kiefer et al., 2007] Kiefer, C., Bernstein, A., and Tappolet, J. (2007). Mining Software Repositoires with iSPARQL and a Software Evolution Ontology.

[Knab et al., 2006] Knab, P., Pingzer, M., and Bernstein, A. (2006). Predicting Defect Densities in Source Code Files with Decision Tree Learners.

[Lee et al., 1994] Lee, A. T., Gunn, T., Pham, T., and Ricaldi, R. (1994). *Software Analysis Handbook: Software Complexity Analysis and Software Reliability Estimation and Prediction*. National Aeronautics and Space Administration.

[Macskassy, 2007] Macskassy, S. A. (2007). Improving Within-Network Classification with Local Attributes.

[Macskassy and Provost, 2005] Macskassy, S. A. and Provost, F. (2005). NetKit-SRL: A Toolkit for Network Learning and Inference.

[McCabe, 1976] McCabe, T. J. (1976). A Complexity Measure. Technical report, Department of Defense, National Security Agency.

[Nagappan et al., 2005] Nagappan, N., Ball, T., and Zeller, A. (2005). Mining Metrics to Predict Component Failures. Technical report, Microsoft Research Redmond, Washington.

[Neal, 1993] Neal, R. M. (1993). Probabilistic Inference Using Markov Chain Monte Carlo Methods. Technical report, Department of Computer Science, University of Toronto.

[Neuhaus et al., 2007] Neuhaus, S., Zimmermann, T., and Zeller, A. (2007). Predicting Vulnerable Software Components. Technical report, Universitt des Saarlandes, Saarbrcken, Germany.

[Neville and Jensen, 2007] Neville, J. and Jensen, D. (2007). *Relational Dependency Networks*, volume Introduction to Statistical Relational Learning, pages 239 – 268. MIT Press.

[Neville et al., 2003a] Neville, J., Jensen, D., Friedland, L., and Hay, M. (2003a). Learning Relational Probability Tree.

[Neville et al., 2003b] Neville, J., Jensen, D., Gallagher, B., and Fairgrieve, R. (2003b). Simple Estimators for Relational Bayesian Classifiers. Technical report, Knowledge Discovery Laboratory, University of Massachusetts.

[Page and Craven, 2003] Page, D. and Craven, M. (2003). Biological Applications of Multi-Relational Data Mining.

[Parnas, 1994] Parnas, D. L. (1994). Software Aging. Technical report, McMaster University, Hamilton, Canada.

[Perlic and Huang,] Perlic, C. and Huang, Z. Relational Learning for Customer Relationship Management.

[Tappolet, 2007] Tappolet, J. (2007). Mining Softwawre Repositories - A Semantic Web Approach. Master's thesis, University of Zurich.

[Taskar et al., 2007] Taskar, B., Abbeel, P., Wong, M.-F., and Koller, D. (2007). Relational Markov Networks. volume Introduction to Statistical Relational Learning, pages 175 – 199. MIT Press.

[Witten and Frank, 2005] Witten, I. and Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, New York.

[Zimmermann et al., 2007] Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting Defects for Eclipse.