OURMINE: An Open Source Data Mining Toolkit

Adam R. Nelson

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Tim Menzies, Ph.D., Chair
Frances VanScoy, Ph.D.
Tim McGraw, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2010

Keywords: Data Mining, Toolkit, Software Defect Prediction, Bash, Awk, Scripting

Abstract

OURMINE: An Open Source Data Mining Toolkit

Adam R. Nelson

When researchers want to repeat, improve or refute prior conclusions, it is useful to have a complete and operational description of prior experiments. If those descriptions are overly long or complex, then sharing their details may not be informative.

OURMINE is a scripting environment for the development and deployment of data mining experiments. Using OURMINE, data mining novices can specify and execute intricate experiments, while researchers can publish their complete experimental rig alongside their conclusions.

This is achievable because of OURMINEs succinctness. For example, this thesis presents three case studies documented in the OURMINE syntax. Thus, the brevity and simplicity of OURMINE recommends it as a better tool for documenting, executing, and sharing data mining experiments.

Acknowledgments

Among those to thank, I would first like to acknowledge my parents and my sister for their utmost support in both good times, and bad. Their dedication to my happiness will never be forgotten. I would also like to thank my advisor, Dr. Menzies, who believed in me and always looked out for my professional well-being.

Finally, I would like to acknowledge Greg Gay, Tomi Prifti, Andrew Matheny, and everyone else whose contribution, direct or indirect, was monumental in my research.

Contents

List of Figures

Chapter 1

Introduction

Since 2004, data mining researchers at West Virginia University have been involved in an open source data mining experiment. Researchers submitting to the PROMISE conference on predictive models in software engineering are encouraged to offer not just conclusions, but also the data they used to generate those conclusions. The result is nearly 100 data sets, all on-line, freely available for download [1].

An important aspect of this thesis is that after *data sharing* comes *experiment sharing*; i.e. repositories should grow to include not just data, but the code required to run experiments over that data. This simplifies the goal of letting other researchers repeat, or even extend, data mining experiments of others. Repeating experiments is not only essential in confirming previous results, but also in providing an insightful understanding of the fundamentals involving those experiments. Thus, it is important to have a prepared set of clearly defined and operational experiments that can be utilized in order to achieve similar prior results.

OURMINE was developed to help graduate students at West Virginia University document and share their data mining experiments. The toolkit uses UNIX shell scripting. As a result, any tool that can be executed from a command prompt can be seamlessly combined with other tools.

[1] `http://promisedata.org/data`

```
1 clean(){
2    local docdir=$1
3    local out=$2

4   for file in $docdir/*; do
5      cat $file | tokes | caps | stops $Lists/stops.txt > tmp
6      stems tmp >> $out
7      rm tmp
8   done
9 }
```

Figure 1.1: An OURMINE function to clean text documents and collect the results. *Tokes* is a tokenizer; *caps* sends all words to lower case; *stops* removes the stop works listed in ”$Lists/stops.txt”; and *stems* performs Porter's stemming algorithm (removes confusing suffixes).

For example, Figure 1.1 shows a simple bash function used in OURMINE to clean text data before conducting any experiments using it. Line 5 passes text from a file, performing tokenization, removing capitals and unimportant words found in a stop list, and then in the next line performing Porter's stemming algorithm on the result.

OURMINE allows connectivity between tools written in various languages as long as there is always a command-line API available for each tool. For example, the modules of Figure 1.1 are written using BASH, Awk and Perl.

1.1 Statement of Thesis

OURMINE stands as a satisfactory data mining toolkit for data mining

- researchers due to experiment reproducibility and sharing
- instructors due to the ease at which concepts can be taught through succinct code
- novices due to rapid yet knowledge-enhancing experiment building using widely applicable scripting languages

1.2 Contributions of this Thesis

This thesis contributes to existing work in the analysis of open source data mining environments. By first examining many already-used toolkits available to the open source community, and then highlighting general strong points as well as weaknesses of these environments, it is hoped that the scripting-based approach taken by OURMINE will be accepted for not only academic endeavors, but practical, industrial applications as well.

1.3 Papers from this Work

- Adam Nelson and Tim Menzies and Greg Gay. *Sharing Experiments Using Open Source Software*. In Software: Practice and Experience, 2nd. round review. 2009-2010
- Ashutosh Nandeshwar and Tim Menzies and Adam Nelson. *Learning Patterns of University Student Retention*. In Expert Systems with Applications, recently submitted. 2010

1.4 Structure of this Thesis

The remaining chapters of this thesis are structured follows:

- Chapter 2 explores a variety of widely used, open source data mining tools
- Chapter 3 introduces more of the specifics of OURMINE. Here exists a description of the environment, comparisons of textual versus visual programming, learning and teaching from within the environment, etc.
- Chapter 4 uses a case study to show how OURMINE can be used by practicioners to ”tune” a treatment to yield the best results.

- Chapter 5 contains a case study in which OURMINE was used to verify current results on cross-company defect prediction using relevancy filters.
- Chapter 6 details a case study of OURMINE in which the toolkit was used to predict university student retention.
- Chapter 7 examines a case study in which OURMINE was used to determine if learning on highly-dense software components provided any benefits over the standard methods.
- Chapter 8 describes a case study that examines empirical comparisons of clustering methods used in text mining via OURMINE.
- Chapter 9 concludes this thesis through a summarization of the varying aspects of OURMINE and also potential future work for the toolkit.

Chapter 2

Related Work

2.1 Existing Open Source Data Mining Tools

As OURMINE is an open source data mining tool, it is important to explore other freely available options. In doing so, it is hoped that insight can be gained by finding similarities and likenesses between these options and the tool discussed herein. It should be noted, however, that while OURMINE is presented in this thesis as an entirely adequate environment for performing data ming tasks, it is not intended to replace any existing options. Instead, its purpose is to supply the software community with yet another powerful alternative. Thus in this chapter other popular tools will briefly be explored.

2.1.1 ADaM

ADaM [1], or the Algorithm Development and Mining System, was developed at the University of Alabama. The tool provides over one hundred components including classifiers, clustering algorithms, and feature selectors. Along with standard data mining components, ADaM comes with tools to process images. These include programs for cropping, rotation, scaling and other image processing techniques standard in computer vision. Such components are provided as executables and Python modules.

Executables packaged with ADaM are utilized via the command line by issuing the component name. For example, in order to train a Naïve Bayes [32] classifier using this option, first the command

```
ITSC_NaiveBayesTrain
```

must be issued in order to learn characteristics of the classes of interest by using sample patterns. Then, the command

```
ITSC_NaiveBayesApply
```

is used to read the characteristics provided by the training module to then classify the input data. Thus, all classifiers in ADaM are structured in this manner – with training and application modules.

Python modules that come with the system are used in typical Python scripts. For example, below illustrates a Python script from the documentation provided with ADaM that performs a median filter on an image by computing the median pixel value in a neighborhood of that pixel in the image.

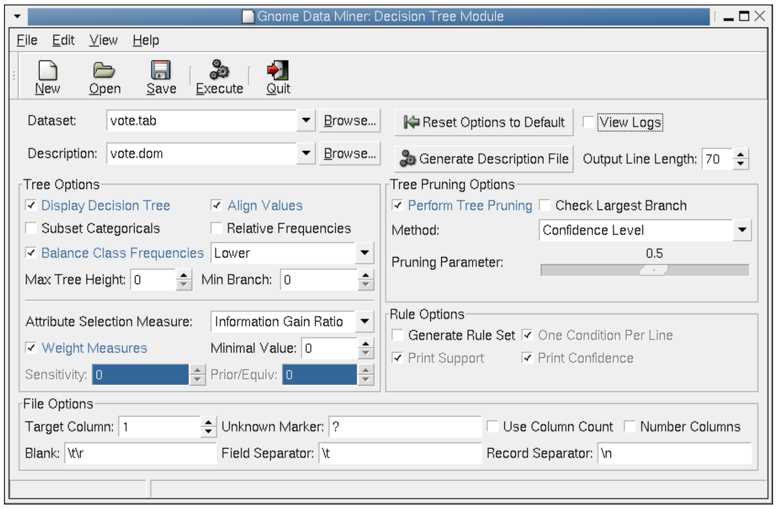
```
import sys
sys.path.append('E:/projects/ADaM/build/')
import ADaM
inFile = "input.bin"
winSize = 7
outFile = "filtered.bin"
id1 = ADaM.ReadImage(inFile) # return a handle as input for the filter funciton
id2 = ADaM.MedianFilter(id1, winSize)
ADaM.WriteImage(outFile, id2)
ADaM.DeleteImage(id1)
ADaM.DeleteImage(id2)
```

2.1.2 Databionic ESOM Tools

The Databionic ESOM Tool [2] takes a noteworthy approach in performing data mining tasks by using Emergent Self-Organizing Maps (ESOM) [10]. A self-organizing map utilizes theory from artificial neural networks to construct a discretized representation of training examples in a low-dimensional space. This representation is called a *map*. SOMs, therefore, remain a useful alternative in visualizing high-dimensional spaces.

2.1.3 Gnome Data Mining Tools

Gnome Data Mining Tools [3] is a collection of freely available tools that are packaged together to make a single collection of units that can be used for data mining tasks. The package requires that Python and Gnome be installed on the system used to run it.



Dataset: vote.tab
Description: vote.dom
Output Line Length: 70

Tree Options
✓ Display Decision Tree ✓ Align Values
Subset Categoricals Relative Frequencies
✓ Balance Class Frequencies Lower
Max Tree Height: 0 Min Branch: 0
Attribute Selection Measure: Information Gain Ratio
✓ Weight Measures Minimal Value: 0
Sensitivity: 0 Prior/Equiv: 0

Tree Pruning Options
✓ Perform Tree Pruning Check Largest Branch
Method: Confidence Level
Pruning Parameter: 0.5

Rule Options
Generate Rule Set ✓ One Condition Per Line
✓ Print Support ✓ Print Confidence

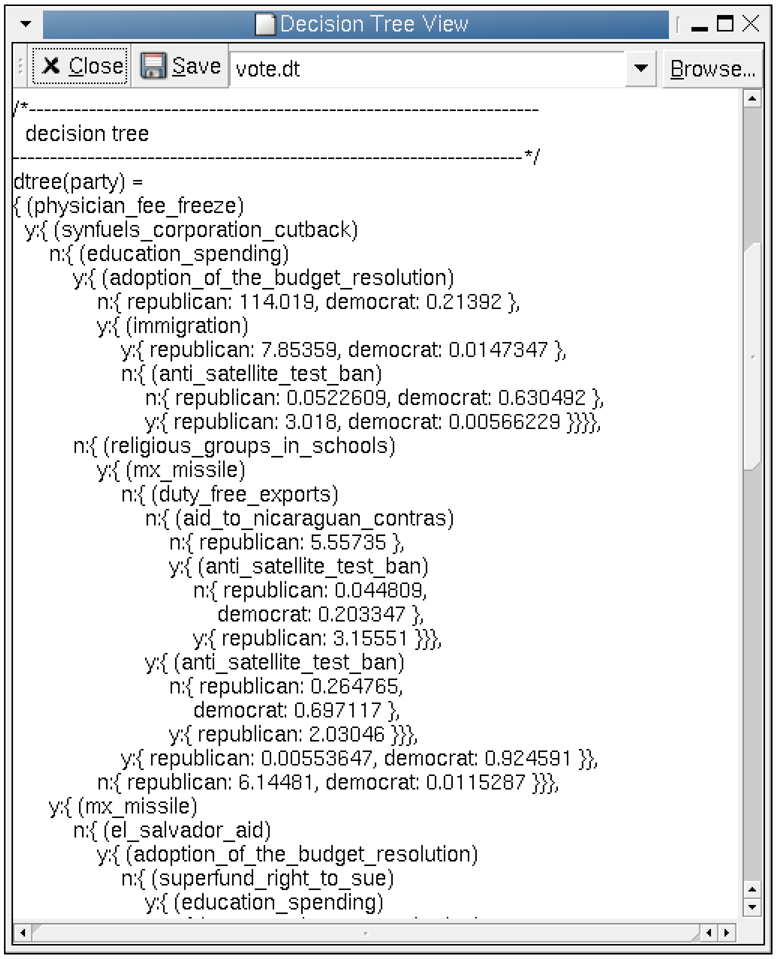
File Options
Target Column: 1 Unknown Marker: ? Use Column Count Number Columns
Blank: \t\r Field Separator: \t Record Separator: \n

Figure 2.1: Selecting Decision Tree options. Obtained from *http : //www.togaware.com/datamining/gdatamine/gdmdtree.html*.

The application takes an approach that utilizes command-driven GUIs in order to run machine learning algorithms on input data. For instance, Figure 2.1 shows the operation of a Decision Tree [25] via a GUI that is executed by typing

```
gdmdtree vote
```

where *vote* is the name of the data set. Many features are available to the user through this interface, such as the ability to select the data set, the option to display the decision tree (as in Figure 2.2) and perform tree pruning, etc. Using Gnome Data Mining Tools, visualization of the data can also be obtained through bar charts, and frequency distributions of data can be represented through bin charts.



```
/*------------------------------------------------------------------
 decision tree
------------------------------------------------------------------*/
dtree(party) =
{ (physician_fee_freeze)
 y:{ (synfuels_corporation_cutback)
    n:{ (education_spending)
       y:{ (adoption_of_the_budget_resolution)
          n:{ republican: 114.019, democrat: 0.21392 },
          y:{ (immigration)
             y:{ republican: 7.85359, democrat: 0.0147347 },
             n:{ (anti_satellite_test_ban)
                n:{ republican: 0.0522609, democrat: 0.630492 },
                y:{ republican: 3.018, democrat: 0.00566229 }}}},
       n:{ (religious_groups_in_schools)
          y:{ (mx_missile)
             n:{ (duty_free_exports)
                n:{ (aid_to_nicaraguan_contras)
                   n:{ republican: 5.55735 },
                   y:{ (anti_satellite_test_ban)
                      n:{ republican: 0.044809,
                         democrat: 0.203347 },
                      y:{ republican: 3.15551 }}},
                y:{ (anti_satellite_test_ban)
                   n:{ republican: 0.264765,
                      democrat: 0.697117 },
                   y:{ republican: 2.03046 }}},
             y:{ republican: 0.00553647, democrat: 0.924591 }},
          n:{ republican: 6.14481, democrat: 0.0115287 }}},
    y:{ (mx_missile)
       n:{ (el_salvador_aid)
          y:{ (adoption_of_the_budget_resolution)
             n:{ (superfund_right_to_sue)
                y:{ (education_spending)
```

Figure 2.2: Output of the decision tree learned. Obtained from *http : //www.togaware.com/datamining/gdatamine/gdmdtree.html*.

2.1.4 KNIME

The Konstanz Information Miner, KNIME [4] is a popular visual-based modular data exploration environment. It incorporates over 100 processing nodes for tasks such as data preprocessing, modeling, data mining and analysis, and supports integration to many other existing tools. Being based on the Eclipse platform, KNIME allows extensibility by constructing custom nodes within the environment that can be used in both production and research settings.

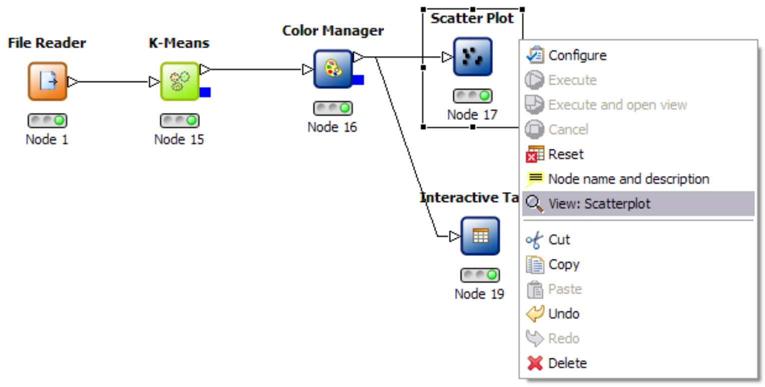


Figure 2.3: Connecting nodes in KNIME. Here, a data set is read through the *FileReader* node, which is then colored, clustered, and analyzed. Obtained from $http://www.knime.org/documentation/getting_started$

KNIME works by allowing the user to build personalized workflows from either preexisting or custom nodes. For example, Figure 2.3 shows a workflow created in order to parse a data set from an ASCII file, add color to it, and cluster the data using the K-means algorithm. The outcome is a scatter plot showing the results of the clustered data. Thus, workflows are created in this manner; icons are dragged to a *Workflow Editor* window, and then connected to complete a sequence of steps to be executed.

After nodes are connected, they must be configured and executed. Configuring nodes is accomplished by right-clicking a node, and then selecting desired characteristics for that node. Figure 2.4 shows this configuration by selecting the file (data set) for the *File Reader* node, which marks the

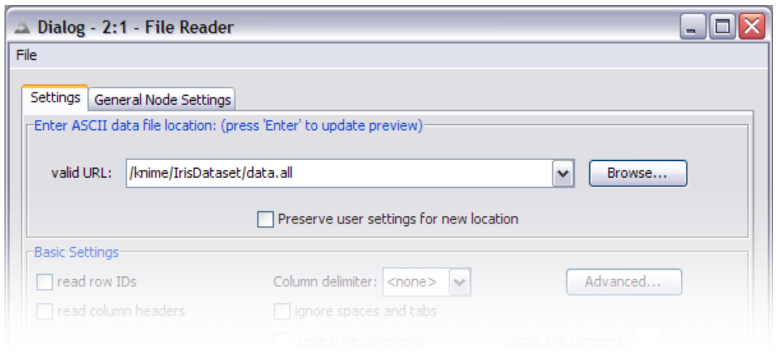

Dialog - 2:1 - File Reader
File
Settings
General Node Settings
Enter ASCII data file location: (press 'Enter' to update preview)
valid URL: /knime/IrisDataset/data.all
Browse...
Preserve user settings for new location
Basic Settings
read row IDs
Column delimiter: <none>
Advanced...
read column headers
ignore spaces and tabs

Figure 2.4: Configuring a node in KNIME. Obtained from $http : //www.knime.org/documentation/getting_started$

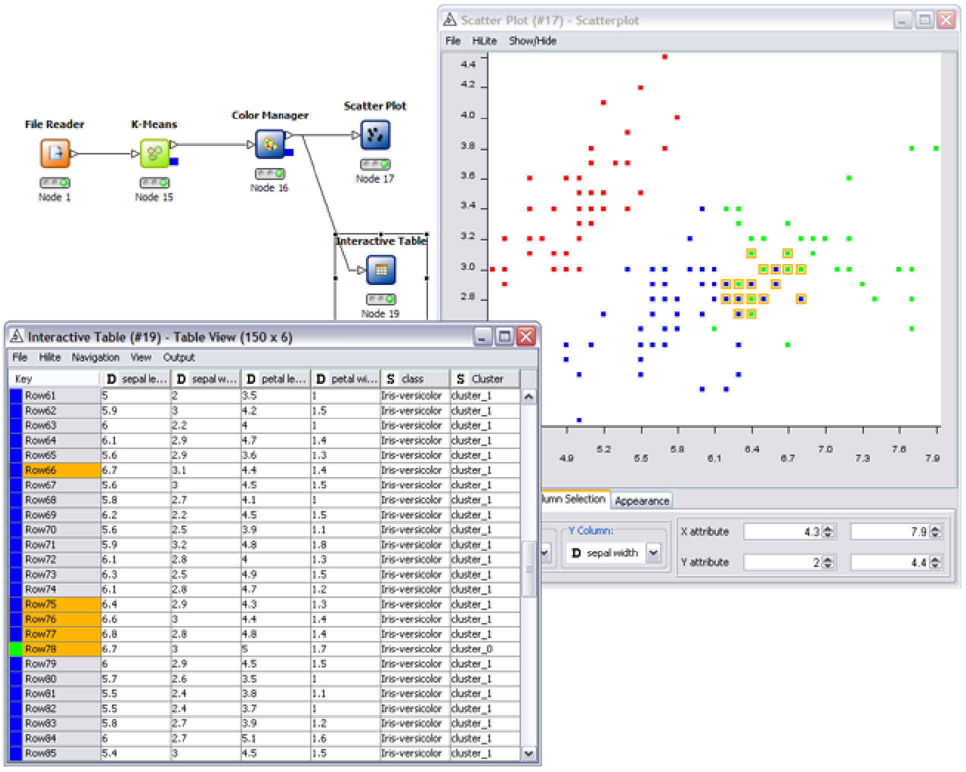
first step in this workflow. Finally, by executing the *Scatter Plot* node in Figure 2.3, all predecessor nodes are automatically executed for the user. The result of the sequential execution of these nodes in this workflow is shown in Figure 2.5.

2.1.5 Orange

Orange [5] is another popular open source alternative available to both novices and experts that allows data mining to be conducted through the use of either Python scripting or visual programming. By using only a few lines of Python (assuming Python is correctly installed on the system), a user can apply data mining modules to data sets, as in the following code:

```
import orange
data = orange.ExampleTable("voting")
classifier = orange.BayesLearner(data)
for i in range(5):
    c = classifier(data[i])
    print "original", data[i].getclass(), "classified as", c
```

Here, *data* is assigned the data set labeled “voting”. The classifier is set to Naïve Bayes , which is learned on the first five instances in the training data. The original class is then printed adjacent


File Reader
Node 1
K-Means
Node 15
Color Manager
Node 16
Scatter Plot
Node 17
Interactive Table
Node 19
Scatter Plot (#17) - Scatterplot
File HiLite Show/Hide
Interactive Table (#19) - Table View (150 x 6)
File Hilite Navigation View Output
Column Selection Appearance

Key	D sepal le...	D sepal w...	D petal le...	D petal wi...	S class	S Cluster
Row61	5	2	3.5	1	Iris-versicolor	cluster_1
Row62	5.9	3	4.2	1.5	Iris-versicolor	cluster_1
Row63	6	2.2	4	1	Iris-versicolor	cluster_1
Row64	6.1	2.9	4.7	1.4	Iris-versicolor	cluster_1
Row65	5.6	2.9	3.6	1.3	Iris-versicolor	cluster_1
Row66	6.7	3.1	4.4	1.4	Iris-versicolor	cluster_1
Row67	5.6	3	4.5	1.5	Iris-versicolor	cluster_1
Row68	5.8	2.7	4.1	1	Iris-versicolor	cluster_1
Row69	6.2	2.2	4.5	1.5	Iris-versicolor	cluster_1
Row70	5.6	2.5	3.9	1.1	Iris-versicolor	cluster_1
Row71	5.9	3.2	4.8	1.8	Iris-versicolor	cluster_1
Row72	6.1	2.8	4	1.3	Iris-versicolor	cluster_1
Row73	6.3	2.5	4.9	1.5	Iris-versicolor	cluster_1
Row74	6.1	2.8	4.7	1.2	Iris-versicolor	cluster_1
Row75	6.4	2.9	4.3	1.3	Iris-versicolor	cluster_1
Row76	6.6	3	4.4	1.4	Iris-versicolor	cluster_1
Row77	6.8	2.8	4.8	1.4	Iris-versicolor	cluster_1
Row78	6.7	3	5	1.7	Iris-versicolor	cluster_0
Row79	6	2.9	4.5	1.5	Iris-versicolor	cluster_1
Row80	5.7	2.6	3.5	1	Iris-versicolor	cluster_1
Row81	5.5	2.4	3.8	1.1	Iris-versicolor	cluster_1
Row82	5.5	2.4	3.7	1	Iris-versicolor	cluster_1
Row83	5.8	2.7	3.9	1.2	Iris-versicolor	cluster_1
Row84	6	2.7	5.1	1.6	Iris-versicolor	cluster_1
Row85	5.4	3	4.5	1.5	Iris-versicolor	cluster_1

Y Column: D sepal width

X attribute 4.3 7.9

Y attribute 2 4.4

Figure 2.5: The result of executing all nodes in the workflow. Note the scatter plot of clustered values. By selecting on each instance from the plot, its corresponding row is shown in the interactive table. Obtained from $http://www.knime.org/documentation/getting_started$

to the class assigned by the classifier.

Visual programming is conducted in Orange by using the *Orange canvas* – a space containing widgets, that when used together, create a *schema*. Orange makes use of widgets due to their modularity. Widgets are visual components on the canvas that supply instructions to other widgets through connected "channels". These channels provide the means for widgets to communicate by being the medium through which information is transmitted. The schemas built from these interconnected widgets can yield very informative visualizations. Figure 2.6 shows a schema consisting of a *File* widget, which "feeds" the data into a *Classification Tree* widget, and finally into both *Classification Tree Viewer 2D* and *Scatterplot* widgets. The result is the visualization of the data itself in the scatter plot, as well as the branching nodes learned from the classification tree.

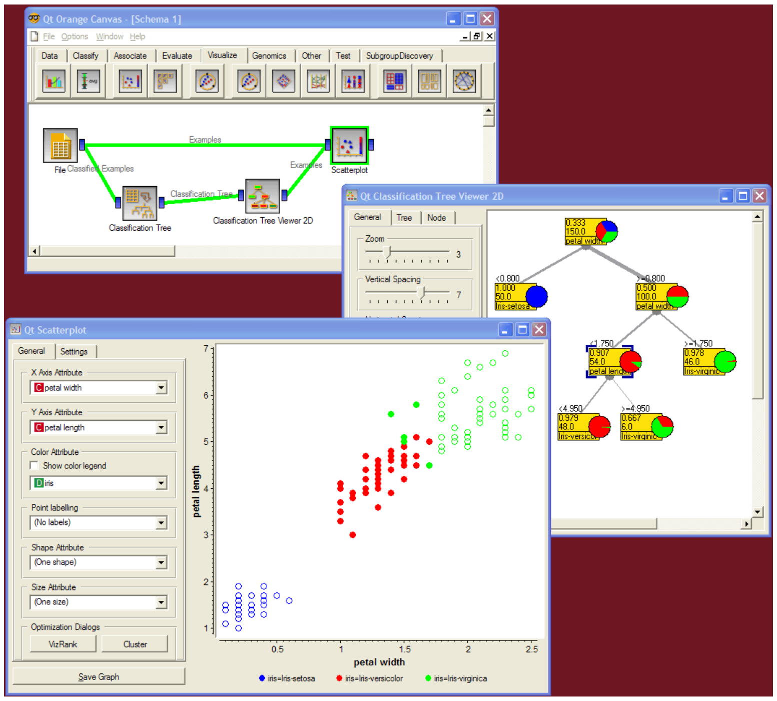
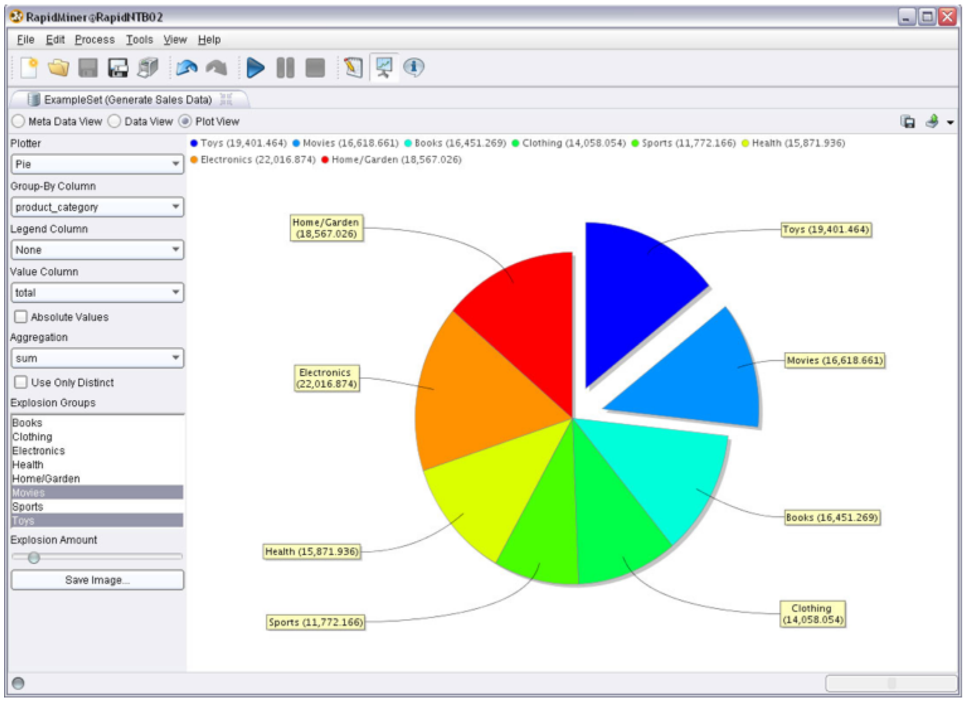


Figure 2.6: A simple schema constructed in Orange. Widgets represent modular parts of the schema that communicate to one another to form the resulting visualizations. Obtained from *http : //www.ailab.si/orange/screenshots.psp*

Thus, Orange remains another powerful tool that can be used by both advanced and novice data miners.

2.1.6 RapidMiner

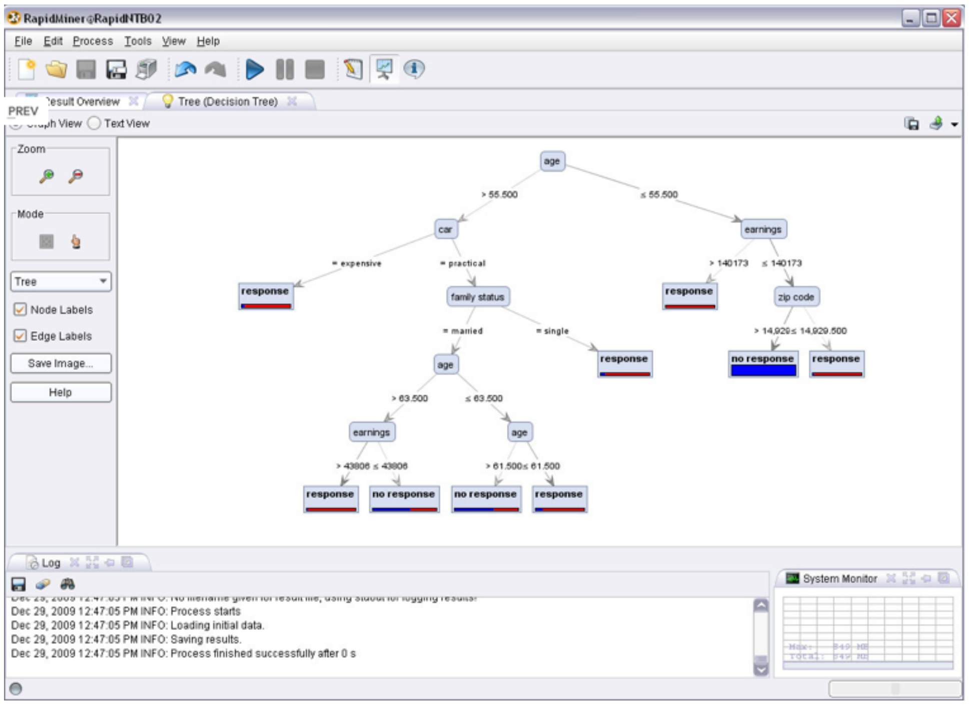
RapidMiner [7] is both an open source and commercially available data mining system, used worldwide. It provides not only a graphical user interface for processes, but also a custom scripting environment for certain desired operations. Workflows are constructed as per previously seen tools – by combining nodes representing components, and connecting them to build a fully operational stream of execution. Data stores can be accessed from existing popular spreadsheet packages, database environments, and other forms such as text files. This data can then be inserted into the workflow, and following nodes can act on this data to accomplish specific data mining tasks.



Plotter: Pie
Group-By Column: product_category
Legend Column: None
Value Column: total
☐ Absolute Values
Aggregation: sum
☐ Use Only Distinct
Explosion Groups: Books, Clothing, Electronics, Health, Home/Garden, Movies, Sports, Toys
Explosion Amount

● Toys (19,401.464) ● Movies (16,618.661) ● Books (16,451.269) ● Clothing (14,058.054) ● Sports (11,772.166) ● Health (15,871.936) ● Electronics (22,016.874) ● Home/Garden (18,567.026)

Figure 2.7: One of the many types of plotting facilities available in RapidMiner. Obtained from $http://www.ailab.si/orange/screenshots.psp$



```
Dec 29, 2009 12:47:05 PM INFO: No filename given for result file, using stdout for logging results!
Dec 29, 2009 12:47:05 PM INFO: Process starts
Dec 29, 2009 12:47:05 PM INFO: Loading initial data.
Dec 29, 2009 12:47:05 PM INFO: Saving results.
Dec 29, 2009 12:47:05 PM INFO: Process finished successfully after 0 s
```

Figure 2.8: One of the many types of visualizations available in RapidMiner. Here, a decision tree is represented as though it was drawn on a piece of paper – from the root node, branching left and right. Obtained from $http://www.ailab.si/orange/screenshots.psp$

Another potentially powerful aspect of RapidMiner is its many visualization tools and plotting facilities. Colorful plots and charts (Figure 2.7), as well as accurate visualization of algorithms' results (Figure 2.8) provide a good explanation medium for those new to data mining.

2.1.7 Rattle

Rattle (the R Analytical Tool To Learn Easily) [8] is a freely available open source data mining toolkit. The authors of this toolkit attempt to provide an easy gateway into learning about the complex statistical language, R [6], while performing data mining through graphical user interfaces. By leading the user through the basics of data mining, it is hoped that he/she will become more familiar with R once it is illustrated how it is used to perform these basics operations.

Rattle is used as an educational device in various places of the world, such as for the Data

Mining Workshop in Canberra, Australia, and also in the Shenzhen Graduate School in China. Rattle is used professionally by others as well. For instance, Australia's largest data mining team in the Australian Taxation Office utilizes Rattle on a daily basis.

2.1.8 Weka

Weka (the Waikato Environment for Knowledge Analysis) [9], developed in New Zealand, is an extremely popular, open source collection of machine learning algorithms used for various data mining tasks. The tools are written in Java, and thus have the ability to be executed on any system with the Java Runtime Environment installed. Standard operations such as data preprocessing, classification and clustering are available to the user through either a command line environment, as in Figure 2.9, or from an easily navigable graphical user interface like shown in Figure 2.10, Figure 2.11, Figure 2.12, and Figure 2.13.

Weka allows access to practical portions of the toolkit through the GUI. These include the simple command line interface of Figure 2.9, the *Experimenter* of Figure 2.10, the *Explorer* in Figure 2.11 and the *KnowledgeFlow* from Figure 2.12. The command line interface is a valuable and powerful portion of Weka. With it, commands can be issued to the system to utilize each module of the environment. However, its tools are also available through a (e.g. UNIX) terminal. This allows for operations to be performed from an external setting, which can be called using any desired language. For instance, in order to call a simple Naïve Bayes classifier using a BASH [30] function, the following code could be used

```
nb () {
    local learner=weka.classifiers.bayes.NaiveBayes;
    $Weka $learner -p 0 -t $1 -T $2
}
```

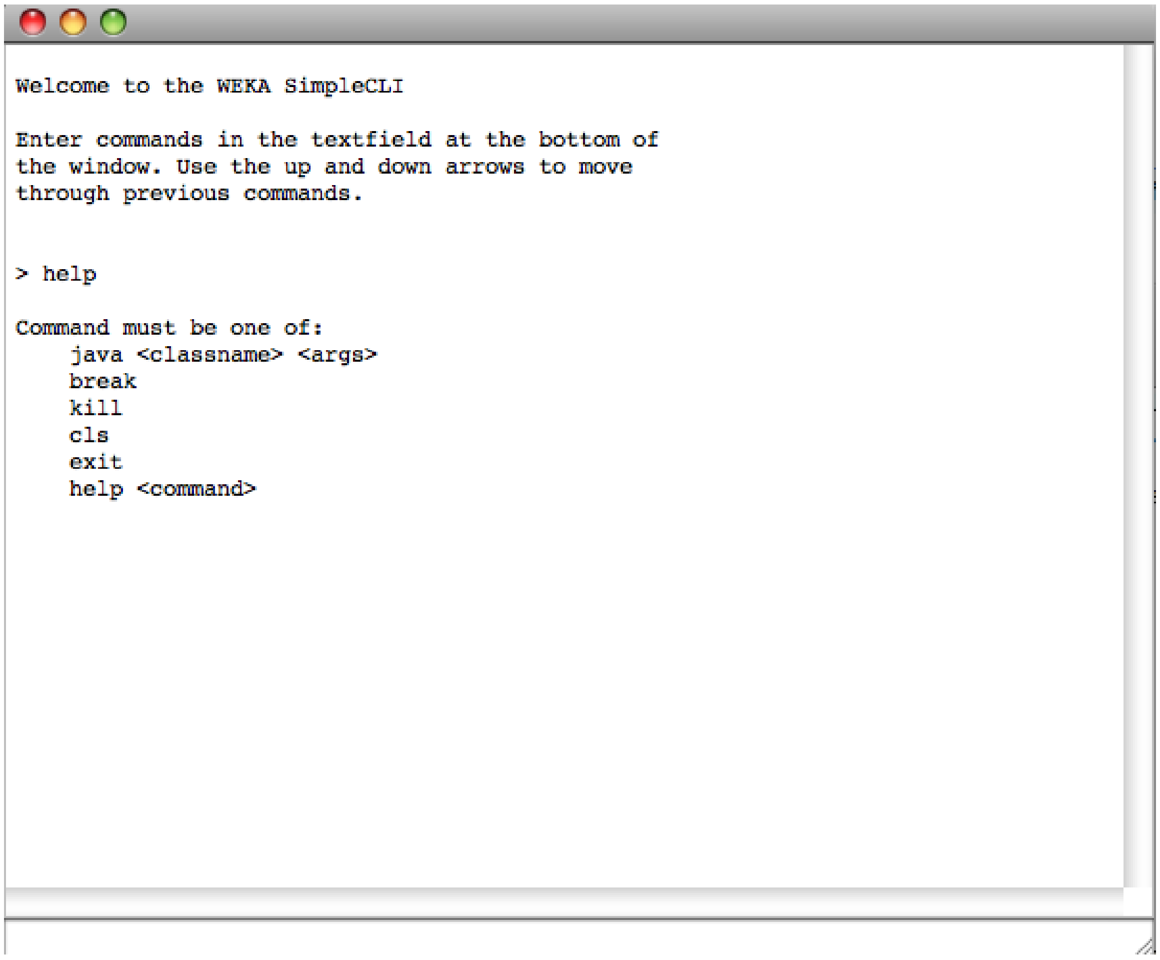
```
nb train.arff test.arff
```

where $*Weka* represents the path of the Weka compiled Java code.

The *Experimenter* option of the toolkit allows standard experiments to be run from a graphical user interface. This provides the means to experiment with various components of data mining. For instance, using this method, statistical comparisons can be made between different treatments on a data set.

For possibly more immediately available results, the *Explorer* option of the GUI gives the user the ability to execute single algorithms that provide quick insight into the properties of the data. For instance, suppose one wished to display the most important attributes ranked using Information Gain (whose results are shown in Figure 2.13 for the popular *weather* data set) as discussed in [25], this interface yields rapid loading of data and execution.

Popular workflows seen in many other existing tools can also be constructed using Weka's *KnowledgeFlow*. In Figure 2.12, a fairly dense system of interconnected nodes perform complex operations. Thus, in short, Weka provides much variety in conducting data mining experimentation.



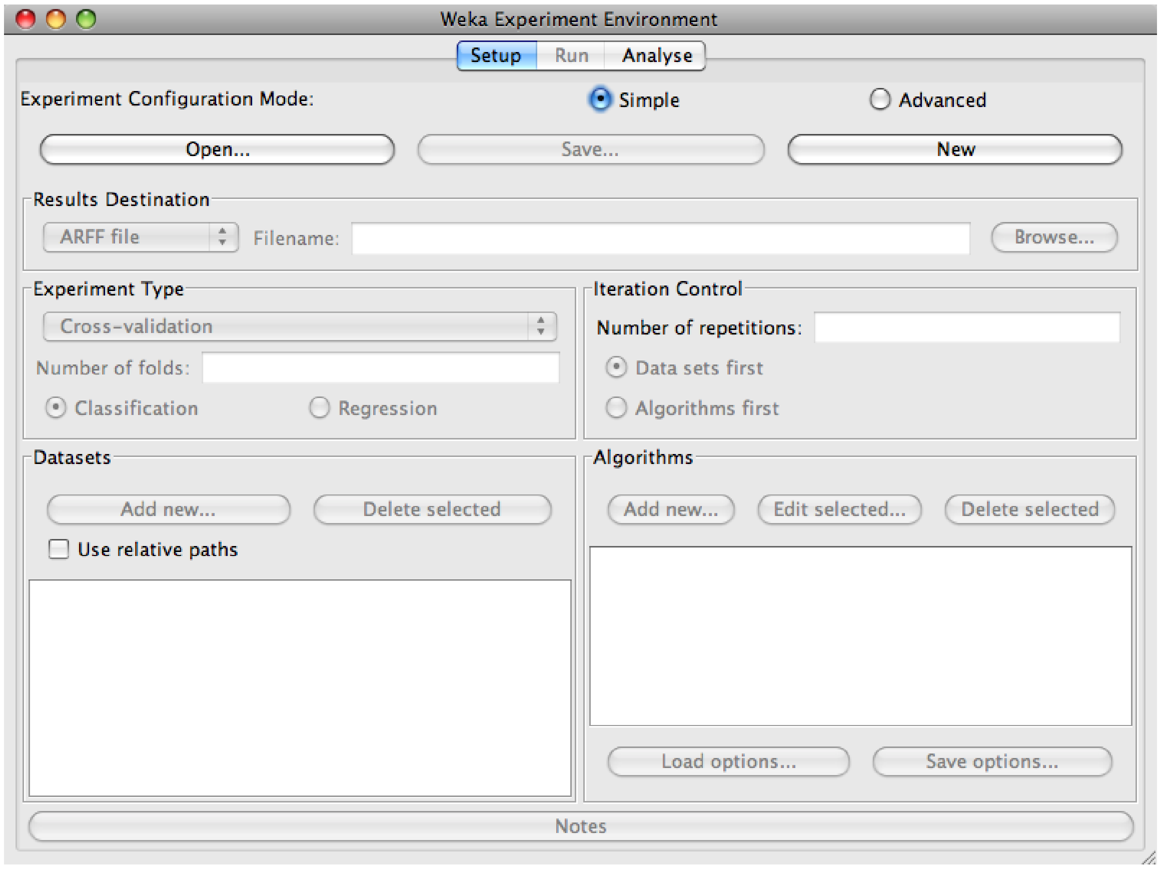
```
Welcome to the WEKA SimpleCLI

Enter commands in the textfield at the bottom of
the window. Use the up and down arrows to move
through previous commands.


> help

Command must be one of:
    java <classname> <args>
    break
    kill
    cls
    exit
    help <command>
```

Figure 2.9: Weka's command line interface (CLI) available to users of the toolkit.



Experiment Configuration Mode: [✓ Simple] Advanced

ARFF file Filename:

Cross-validation

Number of folds:

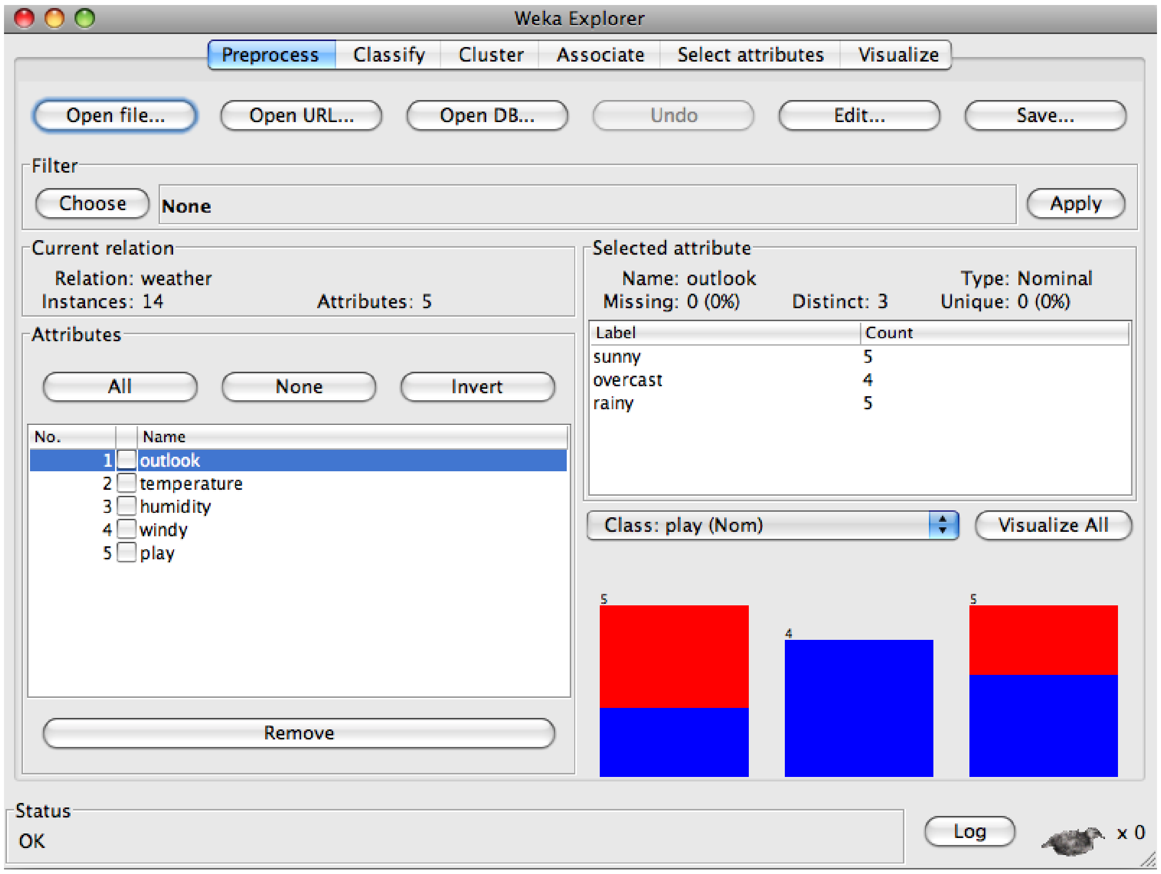
[✓ Classification] Regression

Number of repetitions:

[✓ Data sets first] Algorithms first

Use relative paths

Figure 2.10: Weka's experimenter interface, allowing standard data mining experiments to be set up through the graphical user interface.



Filter: None

Relation: weather
Instances: 14
Attributes: 5

Name: outlook
Type: Nominal
Missing: 0 (0%)
Distinct: 3
Unique: 0 (0%)

Label	Count
sunny	5
overcast	4
rainy	5

No.		Name
1	☐	outlook
2	☐	temperature
3	☐	humidity
4	☐	windy
5	☐	play

Class: play (Nom)

Status: OK

Figure 2.11: Weka's explorer interface, allowing quick execution of many machine learning algorithms to accompish a variety of data mining tasks.

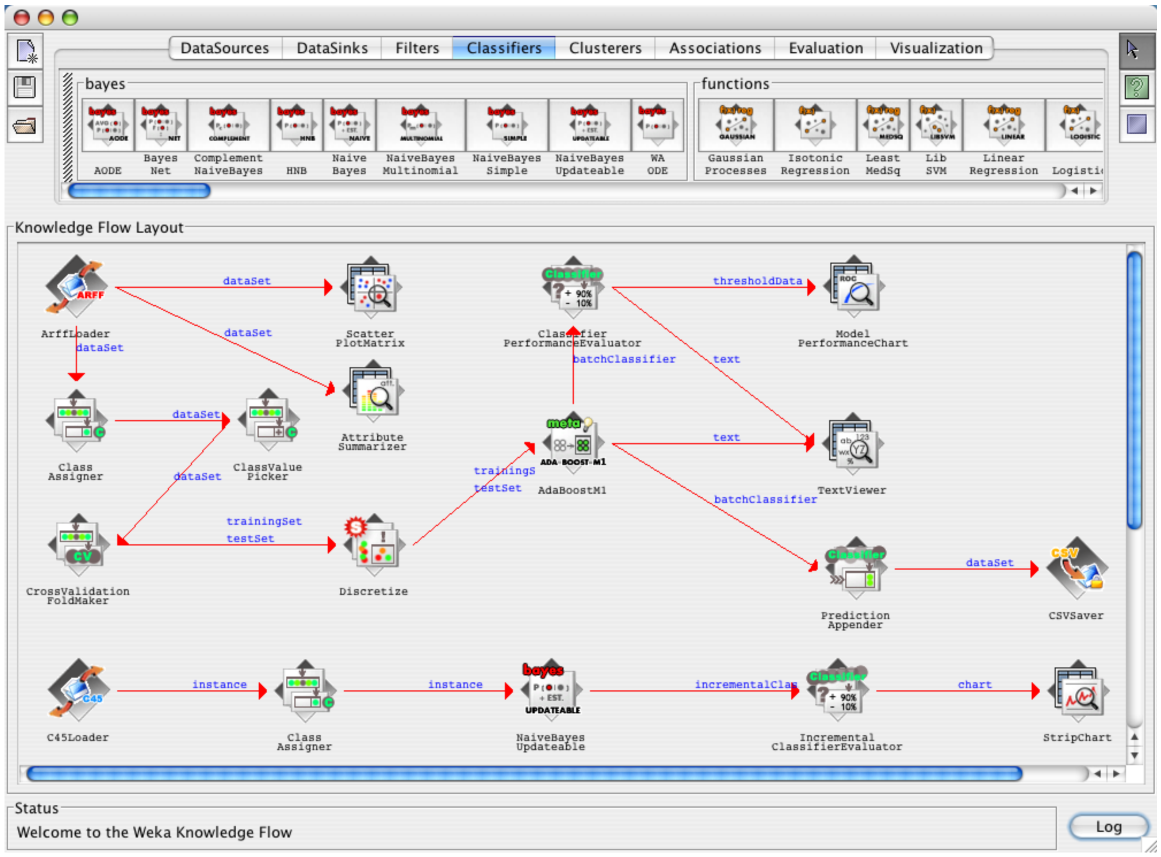
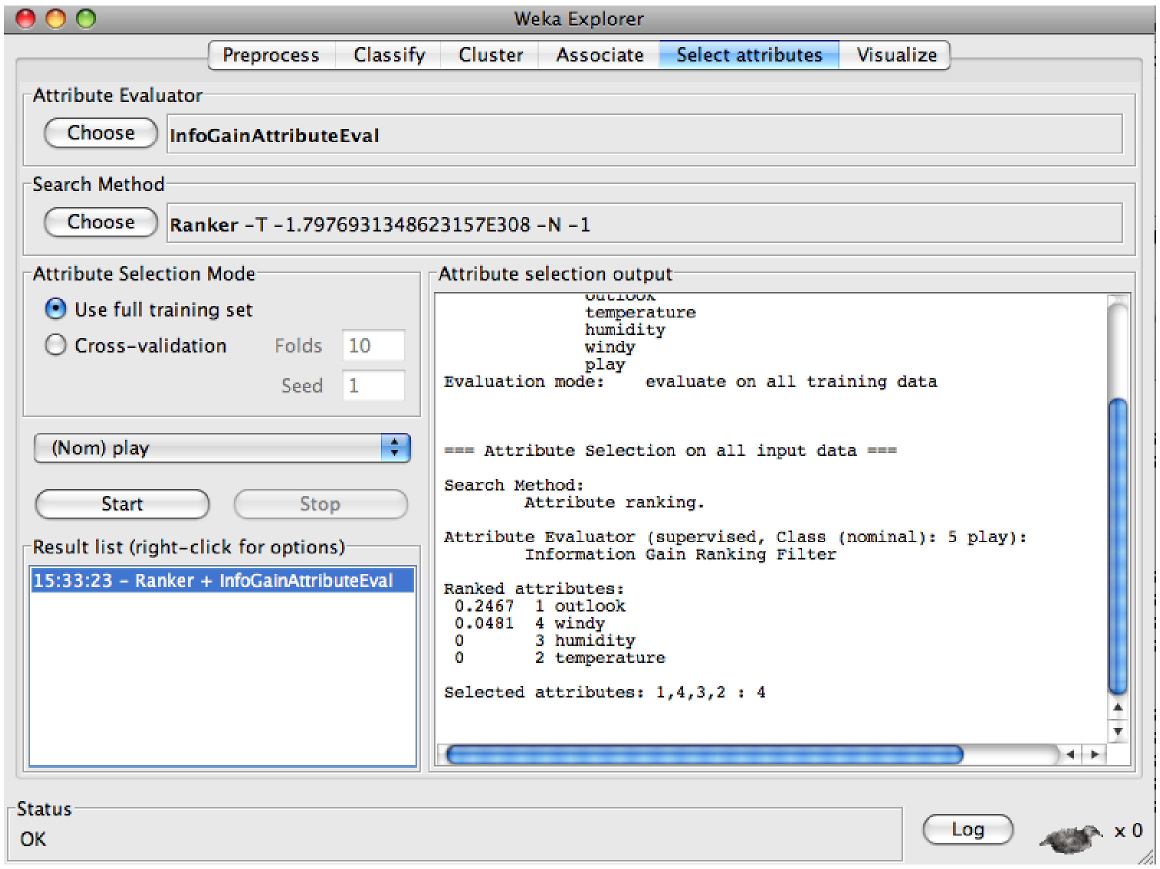


Figure 2.12: Weka's Knowledge Flow builder provides the opportunity to build a workflow seen in many existing toolkits, by attaching nodes sequentially to perform complex tasks. Obtained from *http : //www.cs.waikato.ac.nz/ mhall/knowledgeFlow.png*.



Attribute Evaluator: InfoGainAttributeEval

Search Method: Ranker -T -1.7976931348623157E308 -N -1

Attribute Selection Mode: [✓ Use full training set] Cross-validation Folds 10 Seed 1

(Nom) play

Result list: 15:33:23 - Ranker + InfoGainAttributeEval

Status: OK

```
              outlook
              temperature
              humidity
              windy
              play
Evaluation mode:    evaluate on all training data



=== Attribute Selection on all input data ===

Search Method:
        Attribute ranking.

Attribute Evaluator (supervised, Class (nominal): 5 play):
        Information Gain Ranking Filter

Ranked attributes:
 0.2467  1 outlook
 0.0481  4 windy
 0       3 humidity
 0       2 temperature

Selected attributes: 1,4,3,2 : 4
```

Figure 2.13: Ranking attributes using Information Gain (available through the Explorer) on the popular *weather* data set.

2.2 The Benefits of Textual Programming

A variety of the more widely-used data mining toolkits have been assessed, and it is found that they are are not suitable for publishing experiments. My complaint with these tools is same as that offered by Ritthoff et al. [34]:

- Proprietary tools such as MATLAB are not freely available.
- Real-world experiments are more complex than running one algorithm.
- Rather, such experiments or applications require *intricate combinations* of a large number of tools that include data miners, data pre-processors and report regenerators.

However, the need to "wire up" data miners within a multitude of other tools has been addressed in many ways. In WEKA's visual *knowledge flow* programming environment, for example, nodes represent different pre-processors/ data miners/ etc while arcs represent how data flows between them. A similar visual environment is offered by many other tools including ORANGE (see Figure 2.6).

Ritthoff et al. argues (and I agree) that the standard interface of (say) WEKA does not support the rapid generation of these intricate combinations. For many, these visual environments are either overly elaborate, discouraging or distracting:

- These standard data mining toolkits may be overly elaborate. As shown throughout this thesis, very simple UNIX scripting tools suffice for the specification, execution, and communication of experiments.

- Some data mining novices are discouraged by the tool complexity. These novices shy away from extensive modification and experimentation.

- Other developers may become so enamored with the impressive software engineering inside these tools that they waste much time building environments to support data mining, but never get around to the data mining itself.

- According to Menzies [23], while visual systems provide motivation for beginners, they fail in providing a good explanation device for many software engineering and knowledge engineering problems, as many of these problems are not inherently spatial. For example, suppose an entity-relationship diagram is drawn on a piece of paper. While inferences can be made from the diagram, they are not entirely dependent on the physical location of (e.g.) an entity.

A similar experience is reported by our colleagues in WVU's Department of Statistics. In order to support teaching, they hide these data mining tools inside high-level scripting environments. Their scripting environments shield the user from the complexities of “R” by defining high-level LISP code that, internally, calls “R”. While a powerful language, many find that LISP can be arcane to many audiences.

OURMINE is a data mining scripting environment. The current kit has tools written in BASH/ GAWK/ JAVA/ PERL/ and there is no technical block to adding other tools written in other languages. Other toolkits impose strict limitations of the usable languages:

- MLC++ requires C++
- Extensions to WEKA must be written in JAVA.

My preference for BASH [30]/GAWK [11] over, say, LISP is partially a matter of taste but I defend that selection as follows. Once a student learns, for example, RAPID-I’s XML configuration tricks, then those learned skills are highly specific to that toolkit. On the other hand, once a student learns BASH/GAWK methods for data pre-processing and reporting, they can apply those scripting tricks to any number of future UNIX-based applications.

Chapter 3

Ourmine

3.1 Introduction

OURMINE is a scripting environment developed at West Virginia University for the development and deployment of data mining experiments. Using this toolkit, those new to data mining can specify and execute complex experiments, while researchers in the field can publish their entire experimental rig alongside their results. Sharing experiments is important for not only the reproduction of results, but also in *understanding* those results – by examining the experiment itself, more insight can be gained from theories involved in the experiment's execution. This is achievable due to OURMINE's succinctness. For example, this thesis presents four case studies documented in the OURMINE syntax. Thus, the brevity and simplicity of OURMINE recommends it as an ideal tool for not only constructing experiments, but also documenting, executing, and sharing them with the data mining community.

The toolkit primarily uses UNIX shell scripting and GAWK [11]. As a result, any tool that can be executed from a command prompt can be easily and seemlessly combined with other tools. As I will show in this thesis, components written in various languages such as Perl, Java, GAWK, BASH, etc. can be connected through the environment of OURMINE. Therefore, this strictly scripting approach can be thought to provide the "glue" needed to create an interconnected network of components required in performing data mining experiments.

Also demonstrated in this thesis are the advantages of using scripting languages for not only research, but also in teaching data mining concepts to university students. In essence, while learning complex tasks in a visual environment is more than possible, the user is left with only knowledge of how to use properties specific to that environment. On the other hand, learning to use modern and popular scripting languages utilized by a toolkit such as OURMINE leaves the user with knowledge that can be brought to many general applications outside of data mining.

The following sections describe OURMINE's functions and applications.

3.2 Built-in Data and Functions

In order to encourage more experimentation, the default OURMINE installation comes with numerous data sets:

- *Text mining data sets:* including STEP data sets (numeric): ap203, ap214, bbc, bbcsport, law, 20 Newsgroup subsets [sb-3-2, sb-8-2, ss-3-2, sl-8-2][1]

- *Discrete UCI Machine Learning Repository datasets:* anneal, colic, hepatitis, kr-vs-kp, mushroom, sick, waveform-5000, audiology, credit-a, glass, hypothyroid, labor, pcolic, sonar, vehicle, weather, autos, credit-g, heart-c, ionosphere, letter, primary-tumor, soybean, vote, weather.nominal,breast-cancer, diabetes, heart-h, iris, lymph, segment, splice, vowel;

- *Numeric UCI Machine Learning Repository datasets:* auto93, baskball, cholesterol, detroit, fruitfly, longley, pbc, quake, sleep, autoHorse, bodyfat, cleveland, echoMonths, gascons, lowbwt, pharynx, schlvote, strike, autoMpg, bolts, cloud, elusage, housing, mbagrade, pollution, sensory, veteran, autoPrice, breastTumor, cpu, fishcatch, hungarian, meta, pwLinear,

[1]`http://mlg.ucd.ie/datasets`

```
function train() { #update counters for all words in the record
  Docs++;
  for(I=1;I<NF;I++) {
    if( ++In[$I,Docs]==1)
    Doc[$I]++
    Word[$I]++
    Words++ }
}
function tfidf(i) { #compute tfidf for one word
  return Word[i]/Words*log(Docs/Doc[i])
}
```

Figure 3.1: A GAWK implementation of TF-IDF.

servo, vineyard.

- The defect prediction data sets from the *PROMISE repository*: CM1, KC1, KC2, KC3, MC2, MW1, PC1

OURMINE also comes with a variety of built-in functions to perform data mining and text mining tasks. For a complete list, see the appendix.

3.3 Learning & Teaching with Ourmine

Data mining concepts become complex when implemented in a complex manner. For this reason, OURMINE utilizes simple scripting tools (written mostly in BASH or GAWK) to better convey the inner-workings of these concepts. For instance, Figure 3.1 shows a GAWK implementation used by OURMINE to determine the TF-IDF [31] (Term Frequency * Inverse Document Frequency, described below) values of each term in a document. This script is simple and concise, while a C++ or Java implementation would be large and overly complex. An additional example demonstrating the brevity of OURMINE script can be seen in Figure 3.2, which is a complete experiment whose form can easily be taught and duplicated in future experiments.

```
1 demo004(){
2     local out=$Save/demo004-results.csv
3      [ -f $out ] && echo "Caution: File exists!" || demo004worker $out
4 }

5 # run learners and perform analysis
6 demo004worker(){

7     local learners="nb j48"
8     local data="$Data/discrete/iris.arff"
9     local bins=10
10    local runs=5
11    local out=$1

12    cd $Tmp
13    (echo "#data,run,bin,learner,goal,a,b,c,d,acc,pd,pf,prec,bal"
14    for((run=1;run<=$runs;run++)); do
15        for dat in $data; do

16           blab "data=`basename $dat`,run=$run"
17           for((bin=1;bin<=$bins;bin++)); do

18               rm -rf test.lisp test.arff train.lisp train.arff
19               makeTrainAndTest $dat $bin $bin
20               goals=`cat $dat | getClasses --brief`

21               for learner in $learners; do

22                   $learner train.arff test.arff | gotwant > produced.dat
23                   for goal in $goals; do

24                   cat produced.dat |
25                   abcd --prefix "`basename $dat`,$run,$bin,$learner,$goal" \
26                       --goal "$goal" \
27                       --decimals 1
28                   done
29               done
30           done
31       blabln
32       done
33   done | sort -t, -r -n -k 11,11) | malign  > $out

34   winLossTie --input $out --test w --fields 14 --key 4 --perform 11
35  }
```

Figure 3.2: A demo OURMINE experiment. This worker function begins by being called by the top level function *demo004* on lines 1-4. Noteworthy sections of the demo code are at: line 19, where training sets and test sets are built from 90% and 10% of the data respectively, lines 25-27 in which values such as *pd,pf* and *balance* are computed via the *abcd* function that computes values from the confusion matrix, and line 34 in which a *Wilcoxon* test is performed on each learner in the experiment using *pd* as the performance measure.

```
#naive bayes classifier in gawk
#usage:  gawk -F, -f nbc.awk Pass=1 train.csv Pass=2 test.csv

Pass==1 {train()}
Pass==2 {print $NF "|" classify()}

function train(    i,h) {
   Total++;
   h=$NF;     # the hypotheis is in the last column
   H[h]++;    # remember how often we have seen "h"
   for(i=1;i<=NF;i++) {
     if ($i=="?")
           continue;         # skip unknown values
     Freq[h,i,$i]++
     if (++Seen[i,$i]==1)
           Attr[i]++}  # remember unique values
}
function classify(        i,temp,what,like,h) {
   like = -100000;         # smaller than any log
   for(h in H) {           # for every hypothesis, do...
     temp=log(H[h]/Total); # logs stop numeric errors
     for(i=1;i<NF;i++) {
       if ( $i=="?" )
             continue;     # skip unknwon values
       temp += log((Freq[h,i,$i]+1)/(H[h]+Attr[NF])) }
     if ( temp >= like ) { #  better hypothesis
        like = temp
        what=h}
   }
   return what;
}
```

Figure 3.3: A Naïve Bayes classifier for a CSV file, where the class label is found in the last column.

Another reason to prefer scripting in OURMINE over the complexity of RAPID-I, WEKA, “R”, etc, is that it reveals the inherent simplicity of many of our data mining methods. For example, Figure 3.3 shows a GAWK implementation of a Naïve Bayes classifier for discrete data where the last column stores the class symbol. This tiny script is no mere toy- it successfully executes on very large data sets such as those seen in the 2001 KDD cup and in [27]. WEKA cannot process these large data sets since it always loads its data into RAM. Figure 3.3, on the other hand, only requires memory enough to store one instance as well as the frequency counts in the hash table “*F*”.

More importantly, in terms of teaching, Figure 3.3 is easily customizable. The simplicity of this customizations fosters a spirit of “this is easy” for novice data miners. This in turn empowers them to design their own extensive and elaborate experiments.

Also from the teaching perspective, demonstrating on-the-fly a particular data mining concept helps not only to solidify this concept, but also gets the student accustomed to using OURMINE as a tool in a data mining course. As an example, if a Naïve Bayes classifier is introduced as a topic in the class, an instructor can show the workings of the classifier by hand, and then immediately afterwards complement this by running Naïve Bayes on a small data set in OURMINE. Also, since most of OURMINE does not use pre-compiled code, an instructor can make live changes to the scripts and quickly show the results.

Data mining colleagues at WVU are not alone in favoring GAWK for teaching purposes. Ronald Loui uses GAWK to teaching artificial intelligence at Washington University in St. Louis. He writes:

> *There is no issue of user-interface. This forces the programmer to return to the question of what the program does, not how it looks. There is no time spent programming a binsort when the data can be shipped to /bin/sort in no time.* [21]

Function documentation provides a way for newcomers to OURMINE to not only get to know

```
Function: j4810
Arguments: <data (arff)>
Example(s): j4810 weather.arff
Description: Uses a j48 decision tree learner on the input data

Function Code:
==============
j4810 () {
    local learner=weka.classifiers.trees.J48
    $Weka $learner -C 0.25 -M 2 -i -t $1
}
```

Figure 3.4: Function help in OURMINE.

the workings of each function, but also add to and modify the current documentation. Instead of asking the user to implement a more complicated "man page", OURMINE uses a very simple system consisting of keywords such as *name, args* and *eg* to represent a function name, its arguments and an example of how to use it. Using this documentation is simple. Entering *funs* at the OURMINE prompt provides a sorted list of all available functions in ourmine. Then, by typing *help X*, where *X* is the name of the function, information about that function is printed to the screen. See Figure 3.4 for an example of viewing the help document for the function *j4810*. Documentation for a function is added by supplying a text file to the helpdocs directory in OURMINE named after the function.

3.4 Using Ourmine

While this section represents important and helpful "tips and tricks" for immediately getting up and running with OURMINE, it also serves a purpose as a stand-alone manual to be read by students taking an introductory data mining course to quickly familiarize themselves with the environment. Therefore, subsequent sections are broken into the following:

- **An understanding of the OURMINE environment**

In order to use the environment quickly and effectively, having a basic awareness of its inner-workings is essential. Thus, I first give a brief but ample overview of how OURMINE works by utilizing neatly distributed code segments. This alone should give way to the ability to add and modify scripts as desired.

- **How to construct a simple experiment in OURMINE using easy scripting**

Building experiments in OURMINE is simple. However, there are certain conventions that must be understood and adhered to before any true experimentation can be conducted using it. This portion discusses how to, in detail, construct a simple data mining experiment using OURMINE. The results from this experiment are used in further sections to demonstrate analyses.

- **Ranking experimental treatments using statistical tests**

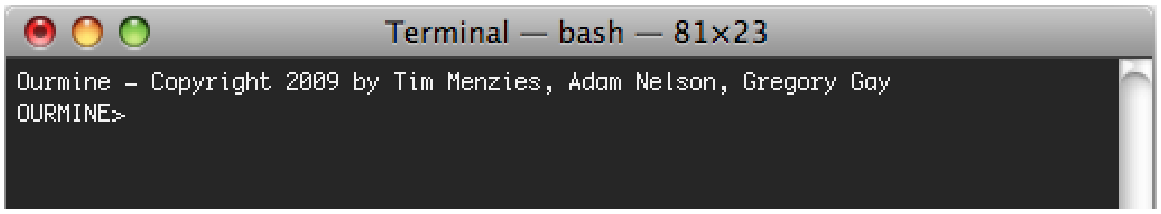
Comparing experimental results is as important as building the experiment itself. For this reason, two approaches to analyze experimental results will be discussed here. First, I show how to rank treatments from an experiment using OURMINE's built-in statistical ranking tests.

- **Examining the variance and median values of experimental results using quartile charts**

Second, I show how to analyze the variance and median values after an experiment has executed. This is essential, in combination with the statistical ranking, to measure the overall performance of a treatment used in any data mining experiment.

3.4.1 The Environment

OURMINE's environment was constructed with *modularity* in mind. Each and every different type of process used in the toolkit is located in its own location and give way to easily customizable scripts. Here I will begin from the initial execution of the environment to modifying scripts to provide a better understanding of how the interconnected code of OURMINE works as a whole. By the time this section is finished, the user should have a working knowledge of where code



```
Ourmine - Copyright 2009 by Tim Menzies, Adam Nelson, Gregory Gay
OURMINE>
```

Figure 3.5: The OURMINE homescreen after installation.

segments are located in OURMINE, and also how to modify and add new segments as desired. After installing OURMINE (see the Appendix), the default prompt is given, as in Figure 3.5.

This prompt notifies the user that the OURMINE environment is available for use, regardless of location. Since the data mining environment is loaded into memory in *addition* to standard shell commands (this becomes very powerful in later use), normal command line operations can still be utilized to its full extent while in OURMINE. To exit the environment, type **exit** at the prompt. To re-enter OURMINE, navigate to where the toolkit was installed, for example in the default location *$HOME/opt/ourmine/our*, and issue the command **./ourmine**. By executing the script *ourmine*, a variable is set to the current directory (default install), and then this value is passed to a shell script called *minerc.sh*, located in *$HOME/opt/ourmine/our/lib/sh*. While this may sound complicated, it's actually very simple. Figure 3.6 shows the contents of the *minerc* file as it is upon installation. Note that this can grow to become much more large and complex as code is added to the environment as desired.

As can be seen in Figure 3.6, the variable **Base** is set to the default installation path. Using this variable, we can construct other variables representing locations of important directories within OURMINE. For example, the variable **Data** is set to the location of the *$HOME/opt/ourmine/our/arffs* directory. This is where all of the data that comes packaged with OURMINE is stored. On the same note, the variables **Sh** and **Java** store the locations of shell code to be executed, as well as java code respectively.

These variables set to their respective paths can be used for many applications. For instance,

```
#define and create required directories

Base=$OurMine
Data=$Base/arffs
Docs=$Base/docs
Help=$Base/helpdocs
Tmp=$HOME/tmp
Var=$Tmp/var
Awk=$Base/lib/awk
Sh=$Base/lib/sh
Java=$Base/lib/java
Perl=$Base/lib/perl
Lists=$Base/lib/lists
Save=$Base/save

mkdir -p "$Var $Tmp"
mkdir -p $Tmp

# useful globals

Weka="java -Xmx2048M -cp $Java/weka.jar "
Clusterers="java -Xmx1024M -jar $Java/Clusterers.jar "
Reducers="java -Xmx1024M -jar $Java/Reduce.jar "

# define and load files

Files="
                $Sh/effort.sh
                $Sh/util.sh
                $Sh/preprocess.sh
                $Sh/learn.sh
                $Sh/cluster.sh
                $Sh/fss.sh
                $Sh/analysis.sh
                $Base/workers/worker_cluster.sh
                $Base/workers/worker_cluster_analysis.sh
                $Base/workers/worker_reduce.sh
                $Base/workers/worker_defects.sh
                $Base/workers/worker_learner_analysis.sh
                $Base/demos.sh
                $Base/demos1.sh
                $Base/demos2.sh
                $Base/demos3.sh
                $Base/demos4.sh
                $Base/demos5.sh
                $Base/demos6.sh
                $Base/demos7.sh
                $Base/demos8.sh
                $Base/demos9.sh
                $Base/demos10.sh
                "
#load all from Files above

for config in $Files; do
        [ -f  "$config" ] && . $config
done

echo "Ourmine - Copyright 2009 by Tim Menzies, Adam Nelson, Gregory Gay"
PS1="OURMINE> "
```

Figure 3.6: The *minerc* file used in OURMINE for setting up the environment.

we can see in Figure 3.6 that **$Java** is used to set another variable, **$Weka**, which allows us to now access all of the data mining properties of the Weka toolkit. Afterwards, every file declared in the string **$Files** is used to import all BASH functions from their respective files. Let's examine further one of these files, such as *$Sh/learn.sh*.

learn.sh is one of many shell scripting files located in *$HOME/opt/ourmine/our/lib/sh*, and each are named according to use; *util.sh* contains code for standard utility functions used in building experiments in OURMINE, *fss.sh* utilizes code crucial for Feature Subset Selection (FSS), *cluster.sh* has scripts responsible for calling code used to cluster data, etc. Since our main concern is with *learn.sh* at the moment, I will spend a bit of time discussing how this works, and how its scripts can easily be modified and extended for further use.

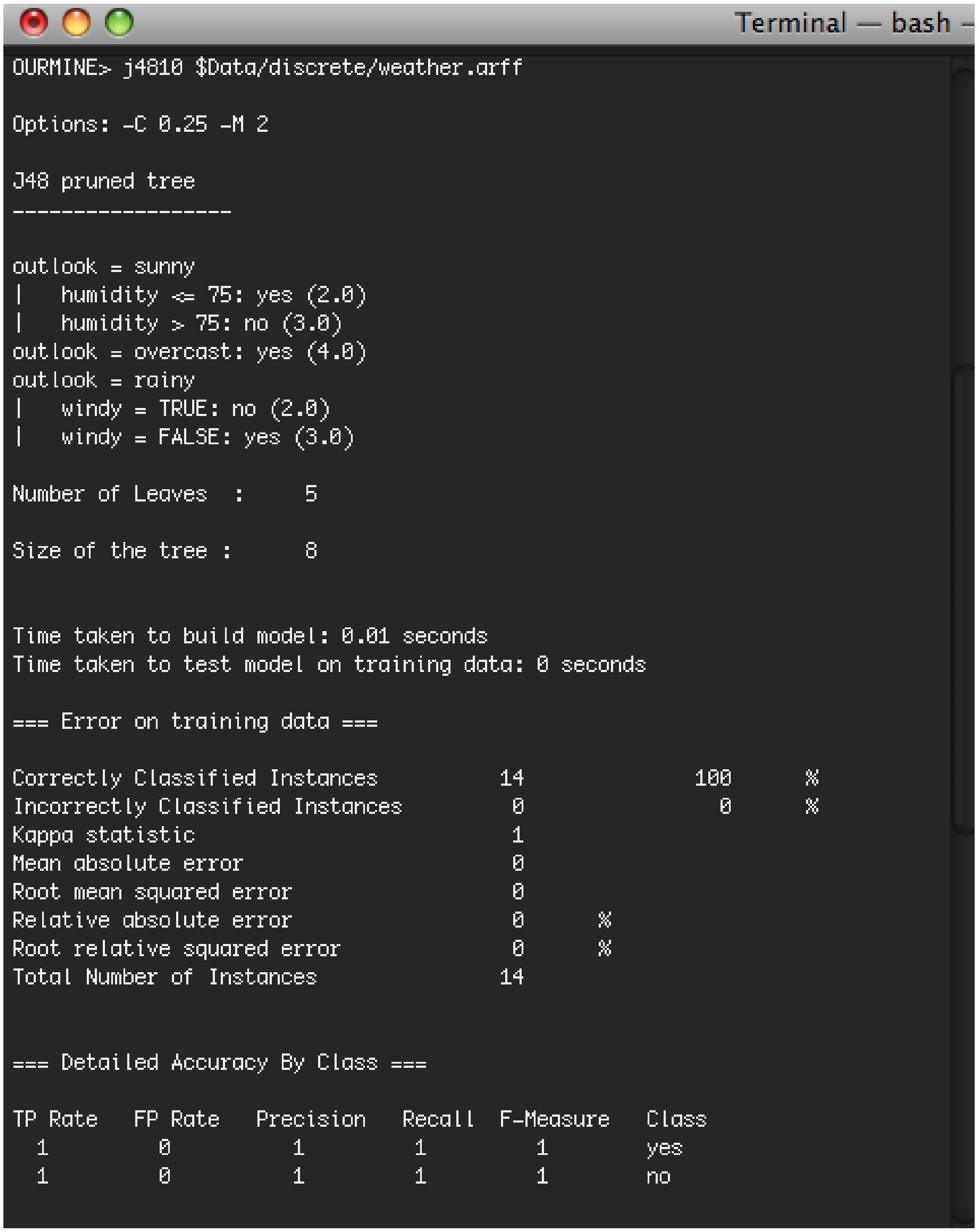
Figure 3.7 shows a subset of the available functions in *learn.sh* that can be accessed from the OURMINE prompt. Here, we can see the function **nb()** that is used to call Weka's Naïve Bayes using as input a training set, followed by a testing set. Likewise, **j48()** executes the J48 decision tree on the training set. Notice those functions ending in 10, such as **nb10** and **j4810**. This is an arbitrary convention that has been used since the earlier days of OURMINE's construction, and simply means that there is no testing data provided, and thus all testing is conducted on the training set.

Running a learner on input data is simple, and is detailed in Figure 3.8. At the top of the screen shot, we can see the command **j4810 $Data/discrete/weather.arff** being issued to OURMINE. Internally, this calls Weka's J48 decision tree classifier on the extremely popular weather data set found in the **$Data** directory under discrete (for discrete data). Further information about the results of running the learner is also printed to the screen, such as the decision tree itself (and corresponding information pertaining to it), the execution time, classification accuracy, and per-class statistics obtained from the confusion matrix, such as precision and recall.

Configuring shell functions in OURMINE for any desired task is simple. For example, suppose a user wished to import the OneR rule learner to be used in future experiments. This operation

```
nb() {
        local learner=weka.classifiers.bayes.NaiveBayes
        $Weka $learner -p 0 -t $1 -T $2
}
nb10() {
        local learner=weka.classifiers.bayes.NaiveBayes
        $Weka $learner -i -t $1
}
j48() {
        local learner=weka.classifiers.trees.J48
        $Weka $learner -p 0 -C 0.25 -M 2 -t $1 -T $2
}
j4810() {
        local learner=weka.classifiers.trees.J48
        $Weka $learner  -C 0.25 -M 2 -i -t $1
}
zeror() {
        local learner=weka.classifiers.rules.ZeroR
        $Weka $learner -p 0 -t $1 -T $2
}
zeror10() {
        local learner=weka.classifiers.rules.ZeroR
        $Weka $learner -i -t $1
}
```

Figure 3.7: A few of the contents of the *learn.sh* file used in OURMINE for data training and classification.



```
OURMINE> j4810 $Data/discrete/weather.arff

Options: -C 0.25 -M 2

J48 pruned tree
------------------

outlook = sunny
|   humidity <= 75: yes (2.0)
|   humidity > 75: no (3.0)
outlook = overcast: yes (4.0)
outlook = rainy
|   windy = TRUE: no (2.0)
|   windy = FALSE: yes (3.0)

Number of Leaves  :     5

Size of the tree :      8


Time taken to build model: 0.01 seconds
Time taken to test model on training data: 0 seconds

=== Error on training data ===

Correctly Classified Instances          14              100      %
Incorrectly Classified Instances         0                0      %
Kappa statistic                          1
Mean absolute error                      0
Root mean squared error                  0
Relative absolute error                  0      %
Root relative squared error              0      %
Total Number of Instances               14


=== Detailed Accuracy By Class ===

TP Rate   FP Rate   Precision   Recall  F-Measure   Class
  1         0          1         1         1        yes
  1         0          1         1         1        no
```

Figure 3.8: Running the J48 algorithm in OURMINE.

requires the addition of one BASH function. The example in Figure 3.9 shows that the environment does not recognize the command **oner10** at first using the input training data. However, by simply editing *learn.sh* to include the function code for the learner, and then supplying the command **reload** (discussed in Section 3.4.2), the environment is *reloaded* and now contains working function code. Figure 3.9 shows the rules learned from running the newly created function.

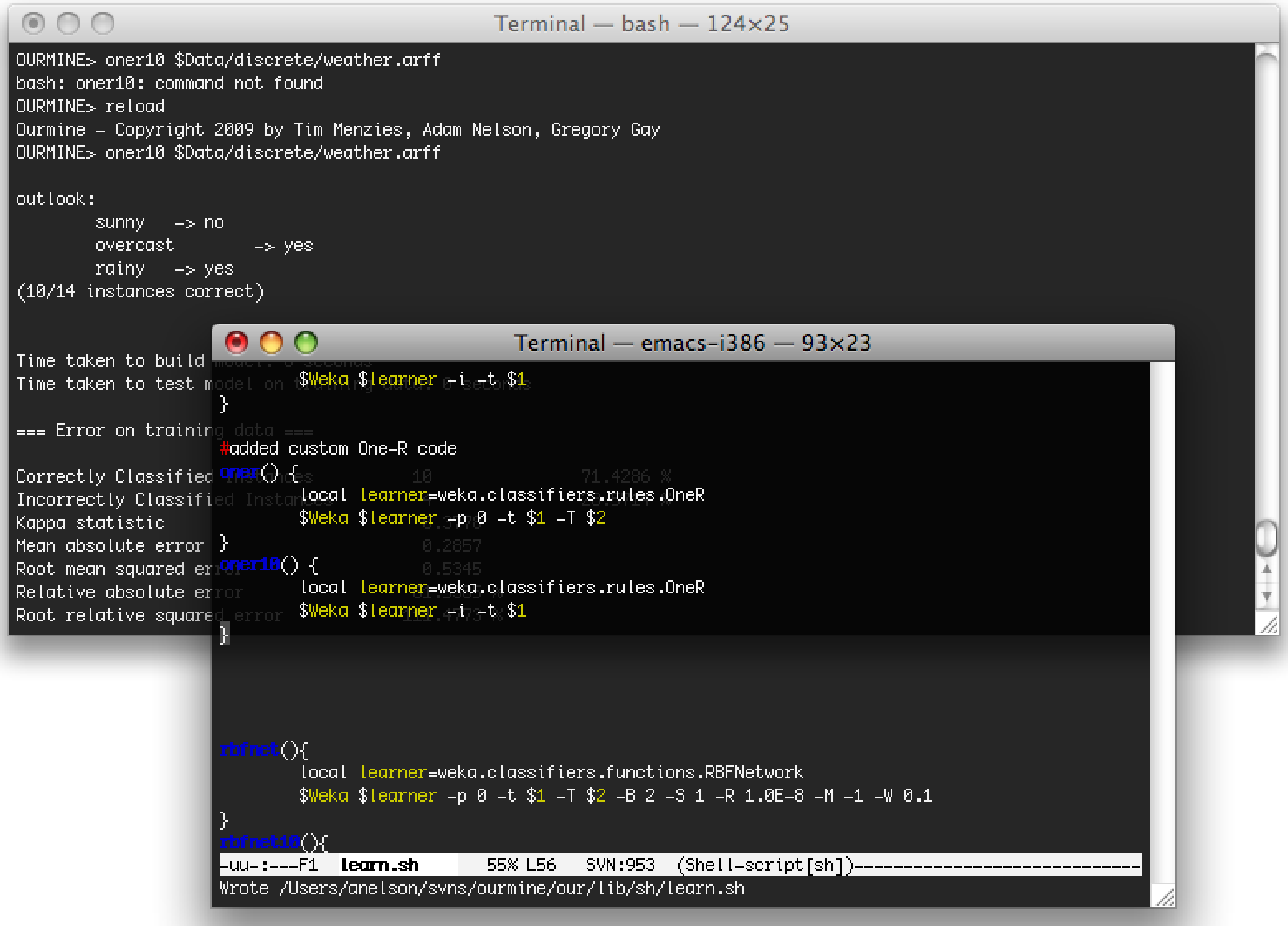
Thus, in summary, OURMINE's structure operates by taking advantage of BASH functions located in clearly defined files. These functions, however, call any number of other functions and libraries written in almost any other programming language. The result is an extremely powerful ”sandbox” in which massive amounts of modification and experimentation can take place.

3.4.2 Tips, Tricks and Useful Ourmine Functions

This section highlights the more helpful and abundantly used functions in OURMINE that, when used effectively, drastically increase the speed and ease at which the environment can be used for data preparation, experimentation, analysis and more.

The first, and perhaps most helpful, function to examine is **funs**. **funs** is a function requiring no parameters that returns a sorted list of the available functions in OURMINE. This is a useful utility ready for use from the command line that enables a user to identify a function by its use through searching for keywords. For example, suppose the user knew that a function existed that could be used to align data, but cannot recall its name. Instead of flipping through a lengthy manual, the command line itself can be used extremely quickly. By entering **funs** from the OURMINE prompt, a large list of functions can be seen. Since an alignment function is in question, he/she can provide the BASH utility *grep* with the output from the **funs** command, as in

```
OURMINE> funs | grep align
malign
OURMINE>
```



```
OURMINE> oner10 $Data/discrete/weather.arff
bash: oner10: command not found
OURMINE> reload
Ourmine - Copyright 2009 by Tim Menzies, Adam Nelson, Gregory Gay
OURMINE> oner10 $Data/discrete/weather.arff

outlook:
        sunny   -> no
        overcast        -> yes
        rainy   -> yes
(10/14 instances correct)


Time taken to build
Time taken to test m

=== Error on trainin

Correctly Classified
Incorrectly Classifi
Kappa statistic
Mean absolute error
Root mean squared er
Relative absolute er
Root relative square
```

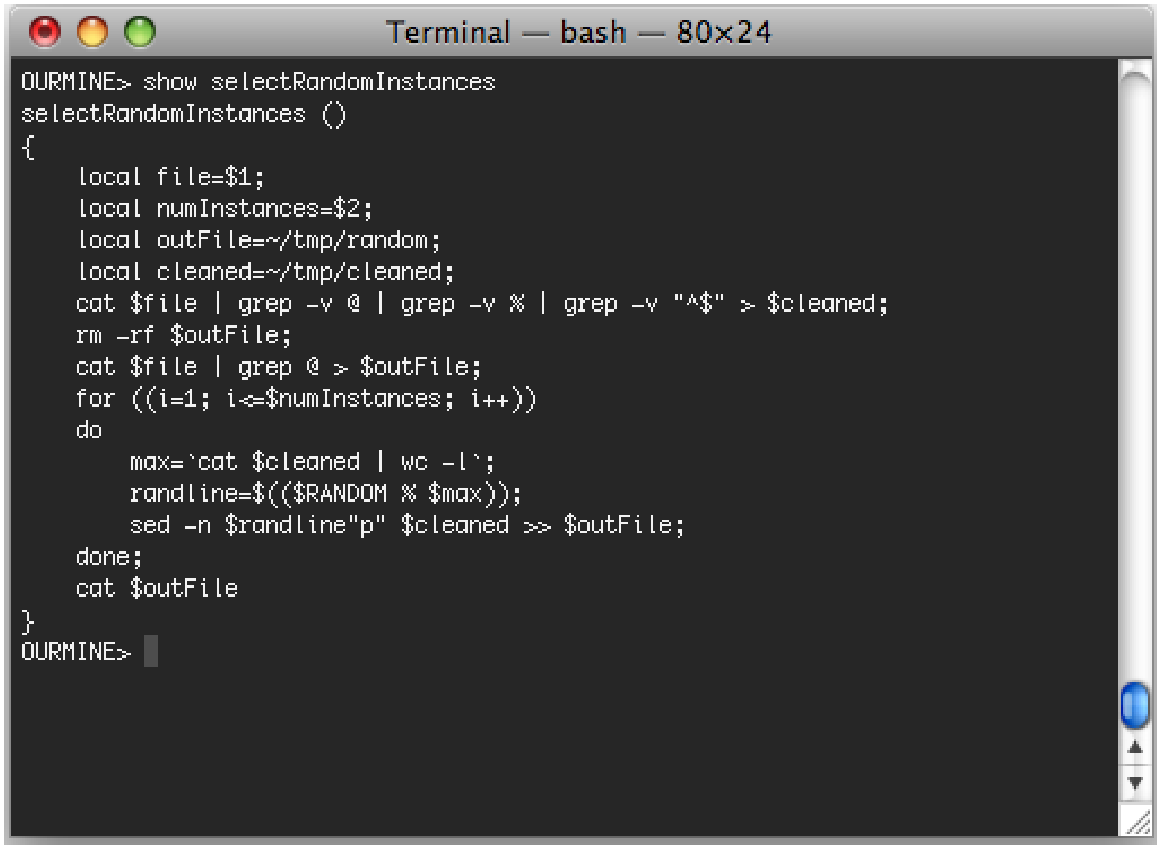
```
        $Weka $learner -i -t $1
}

#added custom One-R code
oner() {
        local learner=weka.classifiers.rules.OneR
        $Weka $learner -p 0 -t $1 -T $2
}
oner10() {
        local learner=weka.classifiers.rules.OneR
        $Weka $learner -i -t $1
}




rbfnet(){
        local learner=weka.classifiers.functions.RBFNetwork
        $Weka $learner -p 0 -t $1 -T $2 -B 2 -S 1 -R 1.0E-8 -M -1 -W 0.1
}
rbfnet10(){
-uu-:---F1  learn.sh       55% L56    SVN:953   (Shell-script[sh])------------------------------
Wrote /Users/anelson/svns/ourmine/our/lib/sh/learn.sh
```

Figure 3.9: Adding Weka's One-R implementation into OURMINE for further use.



```
OURMINE> show selectRandomInstances
selectRandomInstances ()
{
    local file=$1;
    local numInstances=$2;
    local outFile=~/tmp/random;
    local cleaned=~/tmp/cleaned;
    cat $file | grep -v @ | grep -v % | grep -v "^$" > $cleaned;
    rm -rf $outFile;
    cat $file | grep @ > $outFile;
    for ((i=1; i<=$numInstances; i++))
    do
        max=`cat $cleaned | wc -l`;
        randline=$(($RANDOM % $max));
        sed -n $randline"p" $cleaned >> $outFile;
    done;
    cat $outFile
}
OURMINE>
```

Figure 3.10: Using *show* to examine the code (and hence, parameters) of the built-in function **selectRandomInstances**.

From this it is immediately obvious that **malign** is the function desired.

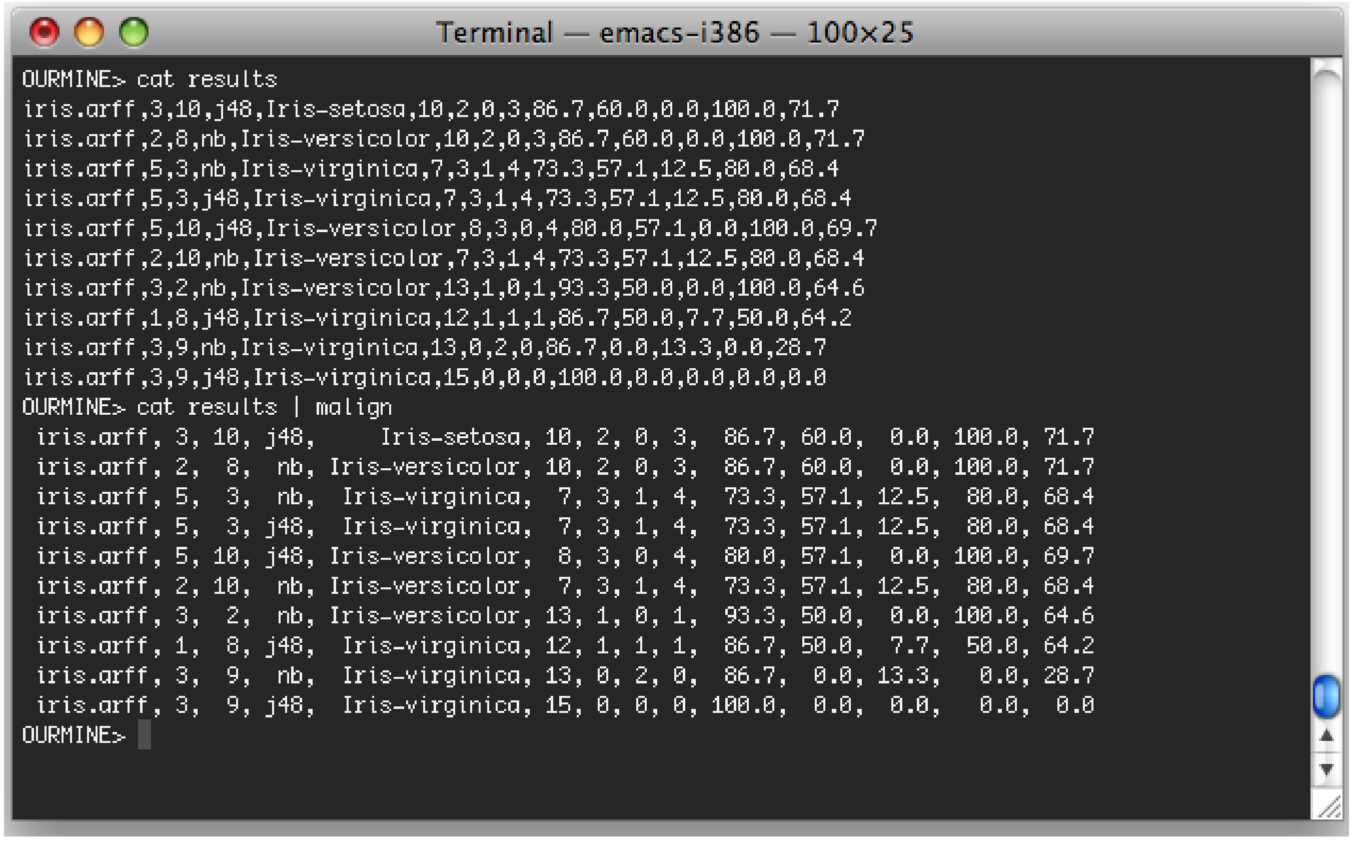
Another very important utility function within OURMINE is **show**. **show** provides the user with the contents of the function in question. This drastically assists in the identification of the number, order and types of parameters to pass to the function. For instance, Figure 3.10 shows the code and parameters for the function **selectRandomInstances**, which simply selects instances at random up to a number that is user-specified. There we can see that in order to successfully call the function, the path of the input file (in arff format), and the number of instances to select. The result is printed to the terminal where further actions can be performed on the output.

Modifying or adding new shell scripts to OURMINE is done easily with the use of the **reload** command. **reload** simply reloads the scripts of all supplied OURMINE functions into working memory (see Section Figure 3.6). This way, as in Figure 3.9, custom scripts can be added to or modified within the environment without having to tirelessly load the environment. For example, two terminals can be used simultaneously for editing and reloading/execution. This encourages rapid modification and experimentation to the existing code-base of OURMINE.

Printing to the screen to provide updates during an experiment's execution can be essential; particular segments of the experiment can be identified as problematic if they yield a large runtime. In most cases, the standard *echo* command issued through the terminal is not sufficient for this purpose. To illustrate, the demo ourmine experiment of Figure 3.2 encapsulates the output of the experiment through the use of *echo* on lines 13 and 33. This is very useful, as this output data can be neatly filtered and aligned as desired by the experimenter. Line 16 prints to the screen the current data set of the experiment, as well as the run number. Using *echo* for this would not have a desirable effect, however, as it would incorporate this text into the output file. Consequently, the results from the experiment would contain extraneous, unwanted information, as well as making the experiment progress of Line 16 not visible during execution. Thus, OURMINE uses the **blab** and **blabln** commands, which print to the screen without or with (respectively) a newline character, using forked processes. This allows the periodic output of experiments to be shown without being combined with the results of the experiment.

Missing values in training data can yield problems for many classifiers, and thus replacing them may be used as a preprocessing step to prepare data for learning. For this reason, OURMINE uses the **replaceMissingValues** function. This function calls Weka's Java code in order to replace missing values, labeled as "?", by the mode for nominal attributes, and the mean for numeric attributes. The function is used by supplying the name (**replaceMissingValues**), with the location of the training arff as a parameter.

Raw results from experiments can not only be harsh on the eye, but can also complicate the analysis of them if not properly aligned. To combat this, OURMINE uses the **malign** function. When output, in a comma separated form, is passed to **malign**, columns are neatly spaced to allow



```
OURMINE> cat results
iris.arff,3,10,j48,Iris-setosa,10,2,0,3,86.7,60.0,0.0,100.0,71.7
iris.arff,2,8,nb,Iris-versicolor,10,2,0,3,86.7,60.0,0.0,100.0,71.7
iris.arff,5,3,nb,Iris-virginica,7,3,1,4,73.3,57.1,12.5,80.0,68.4
iris.arff,5,3,j48,Iris-virginica,7,3,1,4,73.3,57.1,12.5,80.0,68.4
iris.arff,5,10,j48,Iris-versicolor,8,3,0,4,80.0,57.1,0.0,100.0,69.7
iris.arff,2,10,nb,Iris-versicolor,7,3,1,4,73.3,57.1,12.5,80.0,68.4
iris.arff,3,2,nb,Iris-versicolor,13,1,0,1,93.3,50.0,0.0,100.0,64.6
iris.arff,1,8,j48,Iris-virginica,12,1,1,1,86.7,50.0,7.7,50.0,64.2
iris.arff,3,9,nb,Iris-virginica,13,0,2,0,86.7,0.0,13.3,0.0,28.7
iris.arff,3,9,j48,Iris-virginica,15,0,0,0,100.0,0.0,0.0,0.0,0.0
OURMINE> cat results | malign
 iris.arff, 3, 10, j48,     Iris-setosa, 10, 2, 0, 3,  86.7, 60.0,  0.0, 100.0, 71.7
 iris.arff, 2,  8,  nb, Iris-versicolor, 10, 2, 0, 3,  86.7, 60.0,  0.0, 100.0, 71.7
 iris.arff, 5,  3,  nb,  Iris-virginica,  7, 3, 1, 4,  73.3, 57.1, 12.5,  80.0, 68.4
 iris.arff, 5,  3, j48,  Iris-virginica,  7, 3, 1, 4,  73.3, 57.1, 12.5,  80.0, 68.4
 iris.arff, 5, 10, j48, Iris-versicolor,  8, 3, 0, 4,  80.0, 57.1,  0.0, 100.0, 69.7
 iris.arff, 2, 10,  nb, Iris-versicolor,  7, 3, 1, 4,  73.3, 57.1, 12.5,  80.0, 68.4
 iris.arff, 3,  2,  nb, Iris-versicolor, 13, 1, 0, 1,  93.3, 50.0,  0.0, 100.0, 64.6
 iris.arff, 1,  8, j48,  Iris-virginica, 12, 1, 1, 1,  86.7, 50.0,  7.7,  50.0, 64.2
 iris.arff, 3,  9,  nb,  Iris-virginica, 13, 0, 2, 0,  86.7,  0.0, 13.3,   0.0, 28.7
 iris.arff, 3,  9, j48,  Iris-virginica, 15, 0, 0, 0, 100.0,  0.0,  0.0,   0.0,  0.0
OURMINE>
```

Figure 3.11: Using *malign* to neatly align comma-separate output.

for easier reading. Figure 3.11 shows this command in action. A sample of the output from the experiment in Figure 3.2 is shown in the upper portion of the terminal *without* using **malign** (the use of this function was removed from line 33 of Figure 3.2 for this example). The bottom half of the terminal demonstrates the function being used by reading the file first using **cat**, and then piping the output to the function. The result is a much neater and easily readable format.

Learners are assessed by how well they perform on unseen data. This is important for real world applications, as the classifiers are always used to make predictions about the future – data that has not yet been obtained. To test this performance, the standard method is to break up the original data set into a training set and a testing set. The training set is used by the classifier to learn intrinsic characteristics about the data, and the testing set is used to make predictions (hopefully correct) based on these learned characteristics.

Breaking up the data set is simple in OURMINE, and is conducted by calling the function **makeTrainAndTest**. **makeTrainAndTest** requires the user to specify 1.) the original data set, 2.) the total number of bins to use and 3.) the bin to select as the testing set. For a typical 10-way cross validation, the total number of bins to select will be 10, while the bin to choose for the random testing set is chosen by the user. Using the function is as follows

```
OURMINE> makeTrainAndTest $Data/discrete/weather.arff 10 2
```

places *train.arff*, *test.arff* as well as train.lisp and test.lisp in the current directory. For instance, if the above was executed in the $HOME directory, the resulting files would be placed in that directory. From this point, a classifier, such as Naïve Bayes , can be trained on *train.arff* and tested using *test.arff* by calling

```
OURMINE> nb train.arff test.arff
```

Quartile charts provide a succinct representation of experimental results, and therefore provide the ability to quickly assess the performance of treatments. This is due to the ability of the quartile chart to represent minimum, maximum and medians pertaining to any combinations of methods used in the experiment. OURMINE provides the ability to create ASCII quartile charts by calling **quartile**. **quartile** operates on columns of data supplied to it through any filtering means desired by the user. For example, assume an output CSV file (the construction of which is explained in Section 3.4.3) from an experiment contains statistics about the performance of training a Naïve Bayes classifier, among others. Further assume that the metric in question is, say, probability of detection (*pd*) and is located in column number 12 (from left to right). Quartiles can be obtained from this file by extracting the column first, and supplying the output to the **quartile** function. For example, the following would provide statistics about the pd for *every* classifier or treatment used in the experiment.

```
OURMINE> cat results.csv | cut -f12 -d, | quartile
 10.0, 31.0, 54.0, 77.0,100.0,[    -----------           |          ++++++++++++ ]
```

The quartile chart shown works as follows. The minimum and maximum values (in this case for pd) are the floating point numbers found at the very left and just before where the ASCII chart begins, or 10.0 and 100.0 respectively. The median value is shown in the center, as 54.0, and the 25% and 75% quartiles are shown as 31.0 and 77.0 respectively. The ASCII representation provides a quick visualization of the distributions between quartile values. For instance, the dashes, starting at 10.0 and moving to 31.0 display the variance between the minimum value and the first quartile. The space between the dashes and the upright "pipe", —, represent the variance between the first quartile and the median. The median, therefore, is depicted by the pipe. The same holds true for the markings above the median; the space between the pipe and the first "+" represents the the variance between the median and the third quartile.

It should be noted, however, that the above does not concentrate on one particular classifier, in this example. To obtain quartiles, for, say, only Naïve Bayes , we can extract quartiles that meet a particular filter, such as "nb" (short for Naïve Bayes).

```
OURMINE> cat results.csv | grep nb | cut -f12 -d, | quartile
 42.0, 55.0, 70.0, 85.0,100.0,[                    -------       |      ++++++++ ]
```

Here we can intuitively see a smaller variance in the statistics, as indicated by the decreased length in the horizontal dashed and "+" lines, as well as the existence of fewer spaces between each line and the median.

3.4.3 Building a Simple Experiment

The ability to build an experiment using OURMINE is quite possibly the most important skill required when using the toolkit to its full potential. For example, the experiments in Chapters 6 to 7 detail results obtained from experiments constructed in OURMINE.

As this is such a crucial skill when working within the environment, I have written here a tutorial on how simple experiments are constructed using the shell scripting of OURMINE. While the following is a smaller scale experiment, the exercises used here can easily be extended to a much larger and more advanced experiment once the user knows both shell scripting and also the details of how the toolkit functions. It is hoped that this section will be a satisfactory introduction to building experiments so that the user of OURMINE can utilize the skills learned here and apply them to his/her own unique data mining experiments.

For this section I am going to start from scratch to build a complete experiment. Then, in Section 3.4.4, I show how the results obtained from the work in this section can be analyzed. Thus, the end result will be a fully operational experiment whose results could be included in a variety of research papers. But, before building the experiment, we must first determine what it will do. For this tutorial, I have chosen to apply various classifiers on discretized data sets to determine which combination of methods performs the best using varying criteria (explained in Section 3.4.4).

We must first choose an executable file that contains our shell code used in the experiment. For now, this can be named *my-first-experiment.sh*, located anywhere seen fit. Once the file is created and can be executed, the required scripting code can then be added to it.

First, let's construct the function that will be the main experiment. For this, I have named the experiment as *exp*1, and the structure should resemble the following:

```
exp1(){
        #place experiment code here...
}
```

This simple structure will contain all of the operations of our experiment. *exp*1 will, however, call other already-defined functions within OURMINE (such as the learners, splitting training and testing sets, etc.).

For the next step, we need to declare local variables used in the experiment. These variables will make it easy to change certain parameters of the experiment later on, such as the number of runs and bins, the learners or data sets used, and others. So now we will declare the required variables for the experiment. These are *runs*, *bins*, *datas*, and *learners*. *runs* will tell us how many times we wish to perform a cross-validation. For instance, a 10 X 10-way cross-validation means that we wish to perform a 10-way cross-validation *ten* times. *bins* tells us how the cross-validation is performed – using 10 bins means testing on $\frac{1}{10}$-th and training on $\frac{9}{10}$-ths of the data, while using 5 bins means testing on $\frac{1}{5}$-th and training on $\frac{4}{5}$-ths. *datas* simply lists the data sets that we wish to use in the experiment. *learners* lists the learners used in the experiment.

The experiment code is now

```
exp1(){
    local runs=10
    local bins=10
    local datas="audiology.arff mushroom.arff vote.arff soybean.arff"
    local learners="zeror oner nb j48"
}
```

Now that the required variables are set, we will need to construct the ”meat” of the experiment by first building the data loop, and discretizing each data set before continuing.

```
exp1(){
    local runs=10
    local bins=10
    local datas="audiology.arff mushroom.arff iris.arff vote.arff soybean.arff"
    local learners="zeror oner nb j48"

    for data in $datas; do
    done
}
```

Here I am iterating through each data set in the, located in the same directory as the experiment file for simplicity. Now I add the ability to execute each learner

```
exp1(){
    local runs=10
    local bins=10
    local datas="audiology.arff mushroom.arff vote.arff soybean.arff"
    local learners="zeror oner nb j48"

    for data in $datas; do
             for learner in $learners; do

            done
    done
}
```

Next, we wish to add the functionality of performing a cross-validation. In this experiment, I am going to use the standard 10X10-way for consistency. First, we must add the run and bin loops required to perform the cross-validation a certain number of times.

```
exp1(){
   local runs=10
   local bins=10
   local datas="audiology.arff mushroom.arff vote.arff soybean.arff"
   local learners="zeror oner nb j48"

   for data in $datas; do
           for learner in $learners; do
             for((run=1;run<=$runs;run++)); do
                 for((bin=1;bin<=$bins;bin++)); do

                 done
             done
           done
    done
}
```

Each learner now has to be trained using a *training* set, and tested using a *testing* set. These are built using the aforementioned OURMINE function **makeTrainAndTest**. **makeTrainAndTest** builds a training set and testing set from the original data set using parameters passed to it. Remember from Section 3.4.2 that we can tell the function how to construct the training and testing sets by supplying it with the bin number. Also recall that train.arff and test.arff are placed into the current working directory.

```
exp1(){
   local runs=10
   local bins=10
   local datas="audiology.arff mushroom.arff vote.arff soybean.arff"
   local learners="zeror oner nb j48"

   for data in $datas; do
           for learner in $learners; do
             for((run=1;run<=$runs;run++)); do
                 for((bin=1;bin<=$bins;bin++)); do
                     makeTrainAndTest $data $bins $bin
                     goals=`cat train.arff | getClasses --brief`
                     $learner train.arff test.arff | gotwant > result.dat

                 done
             done
           done
    done
}
```

There are two parts of this newer expansion of the experiment that have yet to be discussed. The first is the *goals* variable. *goals* is set to the result of the sub-shell giving all of the classes (using the built-in function **getClasses** from the newly created training set. These represent all classes we wish to focus on for the proceeding testing phases.

Running a learner on a training and testing set has been shown before. However, the **gotwant** function has not. **gotwant** outputs results in the form of classes that were predicted by the classifier for the test instance, as well as the actual classes for that test instance. These values are then used to build the confusion matrix using the **abcd** function, like the following:

```
exp1(){
   local runs=10
   local bins=10
   local datas="audiology.arff mushroom.arff vote.arff soybean.arff"
   local learners="zeror oner nb j48"

   for data in $datas; do
           for learner in $learners; do
             for((run=1;run<=$runs;run++)); do
                 for((bin=1;bin<=$bins;bin++)); do
                     makeTrainAndTest $data $bins $bin
                     goals=`cat train.arff | getClasses --brief`
                     $learner train.arff test.arff | gotwant > result.dat
                     for goal in $goals; do
                        cat result.dat |
                        abcd --prefix "$data,$learner,$bin,$run,$goal" \
                             --goal "$goal" \
                             --decimals 1
                    done
                    blabln "$data,$learner,$bin,$run,$goal"
                 done
             done
           done
    done
}
```

Each goal is passed to the **abcd** function, along with the output from the classification stage. The function then builds the confusion matrix from these results. The **–prefix** passed to the function simply states that the contents of the string will be printed before the statistics from the matrix. By executing the **show abcd** command, we can see that the values printed by the function are: *a, b, c, d, accuracy, pd, pf, precision, balance*. Thus, a single call to the function as written in the above experiment code could produce something such as

```
audiology.arff, nb, 5, 1, cochlear_unknown, 17, 1, 0, 4, 95.5, 80.0, 0.0, 100.0, 85.9
```

these results will eventually be concatenated into one file for further and easy analysis. **blabln** was also added to the experiment to provide the user with a quick update on the current status of the experiment.

The experiment is nearly complete. However, we still must place all of the experiment's output to a properly formatted holding file. This is accomplished by encapsulating the results in memory until being finally written to the output file. While this gives us a rapid way to store results, it is not recommended for experiments that require large amounts of time. In the event of a machine crash or power outage, incrementally saving results to files allows for the ability to restart the experiment at a desired location without the need for a completely fresh run. For the current example, however, this is adequate.

```
exp1(){
   local runs=10
   local bins=10
   local datas="audiology.arff mushroom.arff vote.arff soybean.arff"
   local learners="zeror oner nb j48"

   (echo "#dataset,learner,bin,run,goal,a,b,c,d,acc,pd,pf,prec,bal"
   for data in $datas; do
           for learner in $learners; do
             for((run=1;run<=$runs;run++)); do
                 for((bin=1;bin<=$bins;bin++)); do
                     makeTrainAndTest $data $bins $bin
                     goals=`cat train.arff | getClasses --brief`
                     $learner train.arff test.arff | gotwant > result.dat

                     for goal in $goals; do
                        cat result.dat |
                        abcd --prefix "$data,$learner,$bin,$run,$goal" \
                             --goal "$goal" \
                             --decimals 1
                    done
                    blabln "$data,$learner,$bin,$run,$goal"
                 done
             done
           done
    done) | malign > exp1-output.csv
}
```

Note the addition of a header that is given to the output file (exp1-output.csv) describing the meaning of the column values. This is crucial if others are to be looking at the results and attempting to analyze them. Also see the use of **malign** to neatly organize the results.

The addition of one extra line is required to run the experiment.

```
exp1(){
   local runs=10
   local bins=10
   local datas="audiology.arff mushroom.arff vote.arff soybean.arff"
   local learners="zeror oner nb j48"

   (echo "#dataset,learner,bin,run,goal,a,b,c,d,acc,pd,pf,prec,bal"
   for data in $datas; do
           for learner in $learners; do
             for((run=1;run<=$runs;run++)); do
                 for((bin=1;bin<=$bins;bin++)); do
                     makeTrainAndTest $data $bins $bin
                     goals=`cat train.arff | getClasses --brief`
                     $learner train.arff test.arff | gotwant > result.dat

                     for goal in $goals; do
                        cat result.dat |
                        abcd --prefix "$data,$learner,$bin,$run,$goal" \
                             --goal "$goal" \
                             --decimals 1
                    done
                    blabln "$data,$learner,$bin,$run,$goal"
```

```
                done
            done
          done
    done) | malign > exp1-output.csv
}


exp1
```

The experiment is now finished. To run the experiment from within the OURMINE environment, navigate to the directory containing *my-first-experiment.sh*, and enter

```
OURMINE> . first-experiment.sh
```

This concludes the building of the experiment. Please note that many more complex tasks can be accomplished in much the same way in OURMINE by using both custom and built-in functions. In the Section 3.4.4, I analyze the results from this experiment using statistical analyses tools (also built into OURMINE).

3.4.4 Evaluating Results using Ourmine

There are two main ways discussed in this thesis to analyze classification results. The first way, which I've already demonstrated, is through the use of quartile charts. As previously discussed, quartile charts are used to represent minimum, maximum, median and variance of certain evaluation measures. However, in order to determine if these values are *statistically* different, a statistical test is required. In this section, I will discuss (briefly) the statistical tests used in OURMINE to *rank* treatments, and then give examples as to how these tests are conducted using OURMINE on the results from the previous section.

The Mann-Whitney test (also referred to as the Mann-Whitney U test) is a rank sum, statistical test used to determine if values in two unpaired samples of observations are equally large. For an evaluation measure that is to be maximized, such as *pd*, this is essential because it gives us information regarding the size of values making up the entire distribution.

Running the Mann-Whitney test is simple in OURMINE, but there are a few parameters that must be passed to the function in order for it to work properly. An example of how this can be conducted on the output of the experiment in Section 3.4.3.

```
OURMINE> winLossTie --input exp1-output.csv --test mw --fields 14 --perform 11 --key 2
```

The parameters are as follows:

- **–input**: simply gives the file containing the results from the experiment in a comma-separated form
- **–test**: provides the script with the test to be used. These can either be *mw* for the Mann-Whitney test, or *w* for the Wilcoxon test (described below)
- **–fields**: the total number of fields in the output file
- **–perform**: provides the function with the column number in which to perform the statistical test, from left to right. In this case, field 11 points to the column giving *pd* values.
- **–key**: provides the function with the column number of the key to be used during the statistical test. In this experiment, the key is the learner to be tested. Since the output file prints the data set and then the learner, the key is then 2

Thus, the results obtained from the above is as follows:

```
 #key, ties, win, loss, win-loss
  j48,    1,   2,    0,        2
   nb,    3,   0,    0,        0
zeror,    2,   0,    1,       -1
 oner,    2,   0,    1,       -1
```

where the **#key** is the learner. Ties are labeled in the second column, and detail how many treatments (in this case, learners), that particular treatment tied with in the comparison of distributions. For example, *j48* tied with only one another treatment. However, the decision tree classifier won against another two, and lost against none. Therefore, the end result is in the last column represented by the difference in the number of wins and losses. Here, it can be shown that *j48* performed the best, followed by *nb*. *oner* and *zeror*, however, obtained the same value, and so can be grouped together and considered as having the same rank.

Note that while this ranking works in comparing equally *large* values, such as those that are to be maximized, there is no problem. However, when ranking measures that are to be minimized, such as *pf*, for instance, the performance column is to be negated. In other words, by doing this, normally large values (great for *pd*) are made to be very small, and very small values are made to be large. Since the Mann-Whitney test looks at the size of the values in these distributions and ranks based on largeness, previously large values (bad for pf) will not fair as well as previously small values, which are now large.

Another way to accomplish this for minimized values is by simply negating the values in the *win-loss* column. By doing this, rankings are reversed. However, this may not be ideal, as ranking can be based on whatever the experimenter chooses, such as *wins*, *losses*, or *wins – losses*.

The Wilcoxon test is used to rank *paired* samples of observations from an experiment's results. For example, if in Section 3.4.3, the learners in the experiment would have shared testing sets at classification time, the Wilcoxon test would be performed on the results.

To run the Wilcoxn test in OURMINE is as easy as it is to run the Mann-Whitney test. An example of this would be as follows – almost identical to the previous test, but with an adjustment in the $test$ parameter.

```
OURMINE> winLossTie --input exp1-output.csv --test w --fields 14 --perform 11 --key 2
```

Note the slight change from **–test mw** to **–test w**.

Chapter 4

Case Study 1: Commissioning a Learner through Incremental Random Sampling

4.1 Commissioning a Learner through Incremental Random Sampling

OURMINE's use is not only important to researchers, but also to practitioners. To demonstrate this, two very simple experiments were conducted using seven PROMISE data sets (CM1, KC1, KC2, KC3, MC2, MW1, PC1). The aim of this experiment is to commission a data mining system for a local site. When selecting the right data miners for a particular source of data, three issues are:

1. What learners to use?
2. What discretizers to use?
3. How much data is required for adequate learning?

Given software being released in several stages, OURMINE can be used on stage 1 data to find the right answers to the above questions. These answers can be applied on stages 2,3,4, etc.

In Part A of this experiment, four learners (Naive Bayes, J48, ADTree, One-R) are trained using undiscretized data, and then the same learners are trained on discretized data for a total of eight combinations. While this represents a highly simplified proof-of-concept, from it we can illustrate how our learners can be integrated with other techniques to find the best treatment for the data set(s) of concern. Practitioners can then short-circuit their experiments by only performing further analyses on methods deemed worthwhile for the current corpus. In turn, this decreases the amount of time in which results can be obtained.

In Part B of the experiment, the winning treatment from Part A is used with randomly selected, incremental sizes of training data. In this experiment, it can be shown how early our quality predictors can be applied based on the smallest number of training examples required to learn an adequate theory. Predictors were trained using N = 100, N = 200, N = 300... randomly selected instances up to N = 1000. For each training set size N, 10 experiments were conducted using $|Train|$ = 90% * N, and $|Test|$ = 100. Both $Train$ and $Test$ were selected at random for each experiment. Our random selection process is modeled after an experiment conducted by Menzies et. al. [24]. In that paper, it was found that performance changed little regardless of whether fewer instances were selected (e.g. 100), or much larger sizes on the order of several thousand. Additionally, it was determined that larger training sizes were actually a disadvantage, as variance increased with increasing training set sizes.

Results

Figure 4.1 and Figure 4.2 show the results from Part A of the experiment. As can be seen, ADTrees (Alternating Decision Trees) [14] trained on discretized data yields the best results:

- Mann-Whitney test ranking (95% confidence): ADTrees + discretization provides the highest rank for both PD and PF results

- Medians & Variance: ADTrees + discretization results in the highest median/lowest variance for PD, and the lowest variance for PF, with only a 1% increase over the lowest median

Results from Part B of the experiment are shown in Figure 4.3 and Figure 4.4. Using the winning method from Part A, or learning using ADTrees on discretized data, results show that unquestionably a training size of $N = 600$ instances is the best number to train our defect predictors using the aforementioned experiment setup. This is clear when considering $N = 600$ maintains the highest Mann-Whitney ranking (again using 95% percent confidence), as well as the highest PD and lowest PF medians.

More importantly, perhaps, it should be noted that while the most beneficial number of training instances remains 600, we can still consider a significantly smaller value of, say, $N = 300$ without much loss in performance; a training size of just 300 yields a loss in PD ranking of only one, with a 3% decrease in median, while PF ranking is identical to our winning size and sporting medians of a mere 3% increase.

		pd percentiles			2nd quartile median,
Rank	Treatment	25%	50%	75%	3rd quartile
1	J48 + discretization	20	66	99	
1	ADTree + discretization	25	66	97	
2	NB + discretization	63	73	82	
2	One-R + discretization	13	56	99	
3	J48	29	82	96	
3	NB	40	79	91	
4	ADTree	21	83	97	
4	One-R	17	83	97	

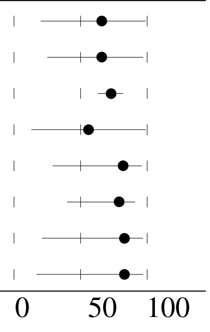


Figure 4.1: Experiment #1 - Part A - (Learner tuning). Probability of Detection (PD) results, sorted by rank then median values.

Rank	Treatment	pf percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	One-R + discretization	0	10	86	
1	ADTree + discretization	2	11	75	
1	J48 + discretization	1	11	80	
2	NB	9	19	60	
2	J48	4	33	72	
3	NB + discretization	12	25	33	
3	ADTree	3	37	79	
4	One-R	3	44	83	

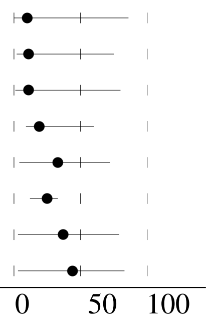


Figure 4.2: Experiment #1 - Part A - (Learner tuning). Probability of False Alarm (PF) results, sorted by rank then median values.

Rank	Training Size	pd percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	600	32	88	99	
1	700	30	86	98	
1	900	32	86	98	
1	800	31	85	98	
1	500	30	84	99	
1	1000	30	84	99	
2	300	31	85	98	
2	400	29	84	99	
3	200	29	83	98	
3	100	29	77	96	

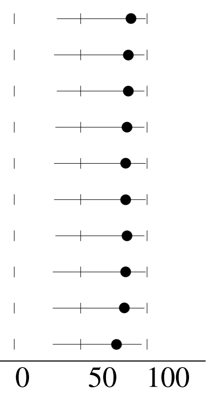


Figure 4.3: Experiment #1 - Part B - (Random Sampling). Probability of Detection (PD) results, sorted by rank then median values.

		pf percentiles		
Rank	Training Size	25%	50%	75%
1	600	1	12	68
1	700	1	14	70
1	900	1	14	68
1	1000	1	14	69
1	500	1	15	69
1	800	1	15	69
1	300	2	15	69
1	400	1	15	71
2	200	2	17	71
3	100	4	23	71

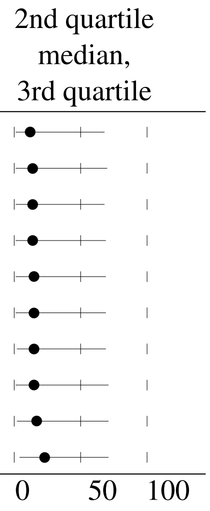


Figure 4.4: Experiment #1 - Part B - (Random Sampling). Probability of False Alarm (PF) results, sorted by rank then median values.

Chapter 5

Case Study 2: Cross-Company Defect Prediction using Relevancy Filtering

5.1 Cross-Company Defect Prediction using Relevancy Filtering

OURMINE was used to reproduce Turhan et al.'s experiment - with a Naive Bayes classifier in conjunction with a k-Nearest Neighbor (k-NN) relevancy filter. Relevancy filtering is used to group similar instances together in order to obtain a learning set that is homogeneous with the testing set. Thus, by using a training set that shares similar characteristics with the testing set, it is assumed that a bias in the model will be introduced. The k-NN filter works as follows. For each instance in the test set:

- the k nearest neighbors in the training set are chosen.
- duplicates are removed
- the remaining instances are used as the new training set.

Building the Experiment

The entire script used to conduct this experiment is shown in Figure 5.1.

To begin, the data in this study were the same as used by Gay et al. [15]; seven PROMISE defect data sets (CM1, KC1, KC2, KC3, MC2, MW1, PC1) were used to build seven combined data sets each containing $\frac{6}{7}$-th of the data. For instance, the file *combined_PC1.arff* contains all seven data sets *except* PC1. This is used as the training set for the cross-company (CC) data. For example, if we wished to learn on all training data except for PC1, this would be a valid data set representation for a cross-company experiment.

Next, as can be seen in line 15 of Figure 5.1, a 10-way cross validation was conducted by calling *makeTrainAndTest*, which is a built-in OURMINE function that randomly shuffles the data and constructs both a test set, containing 10% of the data, and a training set containing 90% of the data. This was repeated ten times, and the resulting data was used in proceeding studies. For instance, in lines 18-24, the original test and training sets are used for the first WC study. However, in the WC experiment using relevancy filtering (lines 25-31), the same test set is used, but with the newly created training set. Lines 32-38 show the first CC study. This study is identical to the WC study except that as we saw before, we use $combined\_X.arff$ files, instead of $shared\_X.arff$.

I chose to use a Naive Bayes classifier for this study because this is what was chosen in the original experiment conducted by Turhan et al. in [35], as well as because Naive Bayes has been shown to be competitive on PROMISE defect data against other learners [20].

Results

Our results for this experiment can be found in Figure 5.2 and Figure 5.3. Figure 5.2 shows *pd* (probability of detection) values sorted in decreasing order, while Figure 5.3 shows pf (probability of false alarm) values sorted in increasing order. These values are calculated as follows. If $\{a,b,c,d\}$ are the true negatives, false negatives, false positives, and true positives (respectively)

```
promiseDefectFilterExp(){
 local learners="nb"
 local datanames="CM1 KC1 KC2 KC3 MC2 MW1 PC1"
 local bins=10
 local runs=10
 local out=$Save/defects.csv
 for((run=1;run<=$runs;run++)); do
   for dat in $datanames; do
       combined=$Data/promise/combined_$dat.arff
        shared=$Data/promise/shared_$dat.arff
        blabln "data=$dat run=$run"
        for((bin=1;bin<=$bins;bin++)); do
            rm -rf test.lisp test.arff train.lisp train.arff
            cat $shared |
            logArff 0.0001 "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19" > logged.arff
            makeTrainAndTest logged.arff $bins $bin
            goals=`cat $shared | getClasses --brief`

            for learner in $learners; do
             blabln "WC"
             $learner train_shared.arff test_shared.arff | gotwant > produced.dat
             for goal in $goals; do
                 extractGoals goal "$dat,$run,$bin,WC,$learner,$goal" `pwd`/produced.dat
             done
             blabln "WCkNN"
             rm -rf knn.arff
             $Clusterers -knn 10 test_shared.arff train_shared.arff knn.arff
             $learner knn.arff test_shared.arff | gotwant > produced.dat
             for goal in $goals; do
                extractGoals goal "$dat,$run,$bin,WCkNN,$learner,$goal" `pwd`/produced.dat
             done
             blabln "CC"
             makeTrainCombined $combined > com.arff
             cat com.arff | logArff 0.0001 "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19" > logged.arff
             $learner logged.arff test_shared.arff | gotwant > produced.dat
             for goal in $goals; do
                extractGoals goal "$dat,$run,$bin,CC,$learner,$goal" `pwd`/produced.dat
             done
             blabln "CkNN"
             makeTrainCombined $combined > com.arff
             cat com.arff |
             logArff 0.0001 "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19" > logged.arff
             $Clusterers -knn 10 test_shared.arff logged.arff knn.arff
             $learner knn.arff test_shared.arff | gotwant > produced.dat
             for goal in $goals; do
                extractGoals goal "$dat,$run,$bin,CkNN,$learner,$goal" `pwd`/produced.dat
             done
    done
  done
  done ) | malign | sort -t, -r -n -k 12,12 > $out
```

Figure 5.1: The OURMINE script used in conducting the WC vs. CC experiment.

found by a binary detector then $pd = recall = \frac{d}{b+d}$ and $pf = \frac{c}{a+c}$. Note that a higher *pd* is better, while a lower pf is better.

The last column of each figure shows quartile charts of the methods' *pd* and pf values. The black dot in the center of each plot represents the median value, and the line going from left to right from this dot show the second and third quartile respectively.

Column one of each figure gives a method its rank based on a Mann-Whitney test at 95% confidence. A rank is determined by how many times a learner or learner/filter wins compared to another. The method that wins the most number of times is given the highest rank.

The following are important conclusions derived from these results:

- When CC data is used, relevancy filtering is crucial. According to our results, cross-company data with no filtering yields the worst *pd* and pf values.
- When relevancy filtering is performed on this CC data, we obtain better *pd* and pf results than using just CC and Naive Bayes.
- When considering only filtered data or only unfiltered data, the highest *pd* and lowest pf values are obtained by using WC data as opposed to CC data. This suggests that WC data gives the best results.

These finds were consistent with Turhan et al.'s results:

- Significantly better defect predictors are produced from using WC data.
- However, CC data leads to defect predictors nearly as effective as WC data when using relevancy filtering.

Thus, this study also makes the same conclusions as Turhan et al. A company should use local data to develop defect predictors if that local development data is available. However, if local data is not available, relevancy-filtered cross-company data provides a feasible means to build defect predictors.

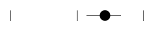
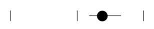
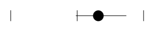
Rank	Treatment	pd percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	WC (local data) + relevancy filter	66	73	80	
2	CC (imported data) + relevancy filter	57	71	83	
2	WC (local data)	59	69	83	
3	CC (imported data)	49	66	87	
					0 50 100

Figure 5.2: Experiment #2 (WC vs. CC). Probability of Detection (PD) results, sorted by rank then median values.

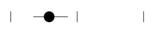
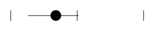
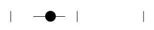
Rank	Treatment	pf percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	CC (imported data) + relevancy filter	17	29	43	
1	CC (imported data)	13	34	51	
2	WC (local data)	17	30	41	
3	WC (local data) + relevancy filter	20	27	34	
					0 50 100

Figure 5.3: Experiment #2 (WC vs. CC). Probability of False Alarm (PF) results, sorted by rank then median values.

Chapter 6

Case Study 3: Predicting Student Retention

6.1 Predicting Student Retention

6.1.1 Building the Experiment

To construct the experiment, certain aspects were first determined to be pertinent in the final selection of top, actionable attributes in the data. The following represents brief explanations of each method used. Results obtained from a combination of which are then analyzed.

6.1.2 Number of Attributes

An attribute in the data could be something such as GPA, or ZIPCODE. The number of attributes to select is crucial in the analysis of the data, because it allows us to conclude how many of the attributes selected we should concentrate on. This is central in selecting actionable attributes. For example, suppose a data set consists of 1000 attributes, but the results from experimentation find that only 15 of the 1000 are actually important. The bulk of subsequent attention could then be spent on what actions to take based on the 15 found, as opposed to the rest of the 985.

In this experiment, I chose n to be the number of attributes selected in increments of 5. Thus, with a maximum of 103 attributes in each data set used in the experiment, 20 different intervals of n were chosen by our feature subset selectors (described below).

6.1.3 Classifiers

Classifiers are used in data mining by employing machine learning techniques in order to learn patterns in data. Once these patterns are learned, we can then begin to attempt to predict outcomes in the data by reflecting on data that has already been examined. We can also determine how well a classifier predicts for the data. This is done by learning on a certain portion of the data, and reflecting on how well predictions are made by another portion of the data that has not yet been seen in the learning process. By examining overall performance, we can make a statement about how much better one classifier predicts on a specific data set than another.

- Naive Bayes - A naive Bayes classifier is a simple and fast probabilistic classifier that uses Bayes' theorem to classify training data. Bayes' theorem, as shown in Equation 6.1, determines the probability P of an event H occurring given an amount of evidence E. The classifier also assumes feature independence; the algorithm examines features independently to contribute to probabilities, as opposed to the assumption that features depend on other features. Surprisingly, even though feature independence is an integral part of the classifier, it often outperforms many other learners [33].

$$Pr(H|E) = \frac{Pr(E|H) * Pr(H)}{Pr(E)} \tag{6.1}$$

- C4.5 - C4.5 [29] is a type of classifier known as a decision tree, and is an extension to the ID3 [28] algorithm. A decision tree [26] (shown in Figure 6.1) is constructed by first determining the best attribute to make as the root node of the tree. ID3 decides this root attribute by using one that best classifies training examples based upon the attribute's information

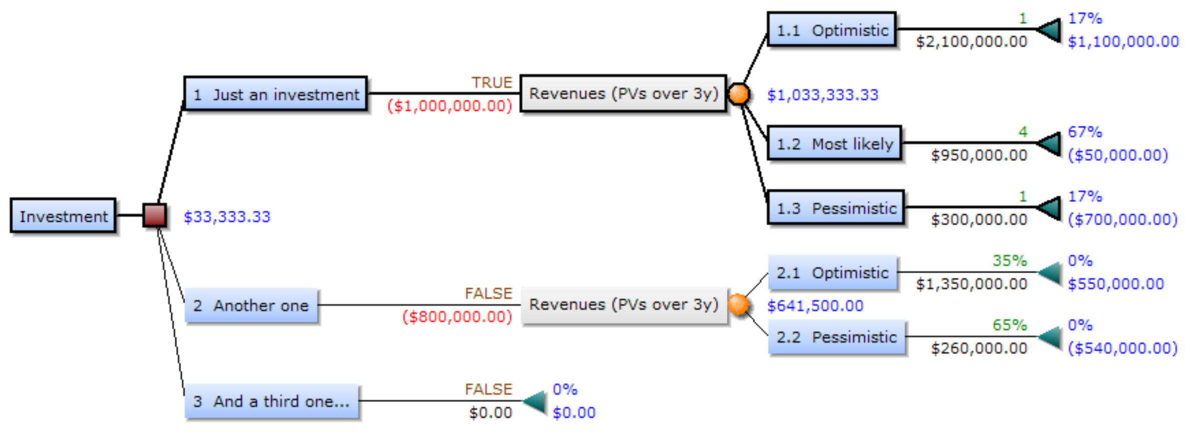


Figure 6.1: A decision tree consists of a root node and descending children nodes who denote decisions to make in the tree's strucure. This tree, for example, was constructed in an attempt to optimize investment portfolios by minimizing budgets and maximizing payoffs. The top-most branch represents the best selection in this example.

gain (described below). Then, for each value of the attribute representing any node in the tree, the algorithm recursively builds child nodes based on how well another attribute from the data describes that specific branch of its parent node. The stopping criteria are either when the tree perfectly classifies all training examples, or until no attribute remains unused. C4.5 extends ID3 by making several improvements, such as the ability to operate on both continuous as well as discrete attributes, training data that contains missing values for a given attribute(s), and employ pruning techniques on the resulting tree.

- One-R - One-R, described in [19], builds rules from the data by iteratively examining each value of an attribute and counting the frequency of each class for that attribute-value pair. An attribute-value is then assigned the most frequently occurring class. Error rates of each of the rules can then be calculated, and the best rules can be ranked based on the lowest error rates.

- Zero-R - Often used to evaluate the success of other classification algorithms, Zero-R is an extremely simple algorithm that gives the majority class from the training data.

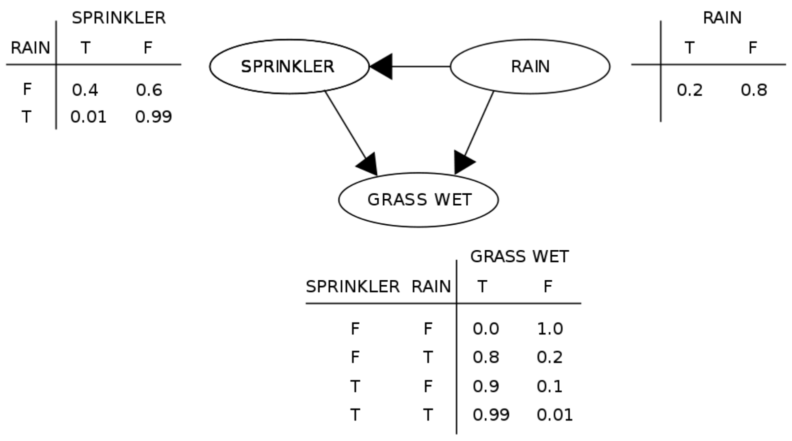


Figure 6.2: In this simple bayesian network, the variable *Sprinkler* is dependent upon whether or not its raining; the sprinkler is generally not turned on when it's raining. However, either event is able to cause the grass to become wet - if it's raining, or if the sprinkler is caused to turn on. Thus, Bayesian networks excel at investigating information relating to relationships between variables.

- Alternating Decision Trees - ADTrees [14] are decision trees that contain both decision nodes, as well as prediction nodes. Decision nodes specify a condition, while prediction nodes contain only a number. Thus, as an example in the data follows paths in the ADTree, it only traverses branches whose decision nodes are true. The example is then classified by summing all prediction nodes that are encountered in this traversal. ADTrees, however, differ from binary classification trees, such as C4.5, in that in those trees an example only traverses a single path down the tree.

- Bayesian Network - Bayesian networks, illustrated in Figure 6.2, are graphical models that use a directed acyclic graph (DAG) to represent probabilistic relationships between variables. As stated in [18] Bayesian networks have four important elements to offer:

  1. Incomplete data sets can be handled well by Bayesian networks. Because the networks encode a correlation between input variables, if an input is not observed, in will not necessarily produce inaccurate predictions, as would other methods.

2. Causal relationships can be learned about via Bayesian networks. For instance, if an analyst wished to know if a certain action taken would produce a specific result, and also to what degree.
3. Bayesian networks promote the amalgamation of data and domain knowledge by allowing for a straightforward encoding of causal prior knowledge, as well as the ability to encode causal relationship strength.
4. Bayesian networks avoid over fitting of data, as ”smoothing” can be used in a way such that all data that is available can be used for training.

- Radial Basis Function Network - A radial basis function network (RBFN) [12] is a type of network called an artificial neural network (ANN). However, RBFNs are specialized in that they utilize a radial basis function as an activation function. An ANN's activation function is used in order to offer non-linearity to the network. This is important for multi-layer networks containing many hidden layers, because their advantages lie in their ability to learn on non-linearly separable examples.

6.1.4 Feature Subset Selectors

Feature Subset Selection (FSS) methods provide ways to determine how important the attributes (or features) are in the data set, and how we can keep the best scoring ones, and throw out the rest. However, we must experiment with varying FSS procedures, because each method can return strikingly different results. Thus, just by experimenting with attributes selected from a handful of FSS, we are not left with a sense of how well attributes were selected from a data set compared to other feature selection tools.

A brief overview of the FSS methods used in this study were as follows:

- CFS - Correlation-Based Feature Selection [17] begins by constructing a matrix of feature to feature, and feature-to-class correlations. It then uses a best first search by expanding the best subsets until no improvement is made, in which case the search falls to the unexpanded subset having the next best evaluation until a subset expansion limit is met.

- Information Gain - Information Gain works by using a concept from information theory known as entropy. Entropy measures the amount of uncertainty, or randomness, that is associated with a random variable. Thus, high entropy can be seen as a lack of purity in the data. Information gain, as described in [25] is an expected reduction of the entropy measure that occurs when splitting examples in the data using a particular attribute. Therefore an attribute that has a high purity (high information gain) is better at describing the data than one with a low purity. The resulting attributes are then ranked by sorted their information gain scores in a descending order.

- Chi-squared - Attributes can also be ranked using the chi-squared statistic. The chi-squared statistic [13] is used in statistical tests to determine how distributions of variables are different from one another. Note that these variables must be categorical in nature. Thus, the chi-squared statistic can evaluate an attribute's worth by calculating the value of this statistic with respect to a class. Attributes can then be ranked based on this statistic.

- One-R - One-R (as described above), can also be used to deliver top-ranking attributes. Since each rule contains one attribute and a corresponding value, we can evaluate attributes by sorting them based on the error rate of the rule associated with that attribute-value pair. Using this, top attributes are those whose rules result in the lowest error rates.

6.1.5 Cross-Validation

In the process of experimentation, it is crucial to determine a method's performance. Using performance criteria, further analysis can be conducted on experimental results to aid in the search for an optimal solution. Cross-validation provides the ability to discover how well a classifier performs on any given data set or a treatment of that data set. This is conducted by randomly partitioning the data into two subsets, called the training set, and the testing set. Specifically for this experiment, the data prior to partitioning has been reduced given n attributes selected using an FSS method.

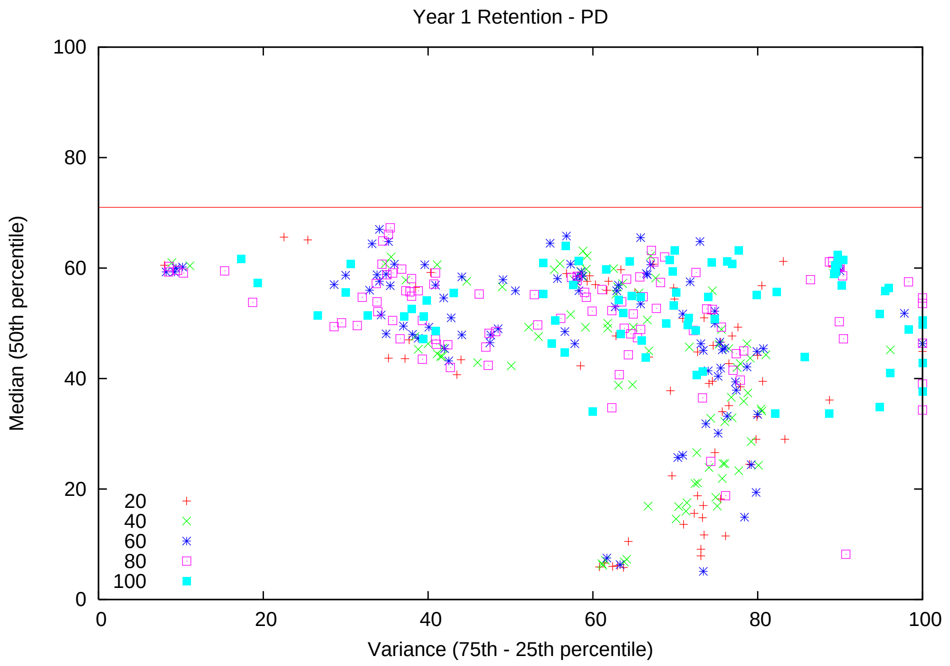
In the learning phase, only the training subset is used by the classifier. The testing set is then used to determine how well the concepts learned from the training phase can be applied to unseen data. However, to reduce variability, the partitioning of the data and reclassification of resulting subsets is generally conducted multiple times. In this experiment, for example, a 5 X 5 cross-validation was performed. This means that five times I partitioned the data into a testing set consisting of $\frac{1}{5}$-th of the data, and a training set of $\frac{4}{5}$-ths of the data. After the five rounds, median values of the validation results are examined, and are assigned to a particular combination of the above facets.

6.1.6 Analysis of Experimental Results

6.1.7 Evaluation Metrics

The evaluation metrics used in this experiment are standard in data mining to measuring the performance of a method. These are represented as probability of detection (PD), probability of false alarm (PF), and variance. PD denotes the probability that the classifier will predict correctly for a given class, given both its correct and incorrect predictions. Thus, PD values should be maximized. PF, on the other hand, is the probability that the classifier will predict incorrectly for a given class, also given its correct and incorrect predictions. For this reason, PF results should be minimized.

Variance was also used in the experiment based on PD and PF values independently as an extra means of determining performance. Variance in these values provides insight into how much reliability a classifier supports on the data. For example, if a method's PD values range from very low to very high, we can determine that the particular method is not consistent in its probabilities of detection. Therefore, it is desired to have a very small variance in both PD and PF values.



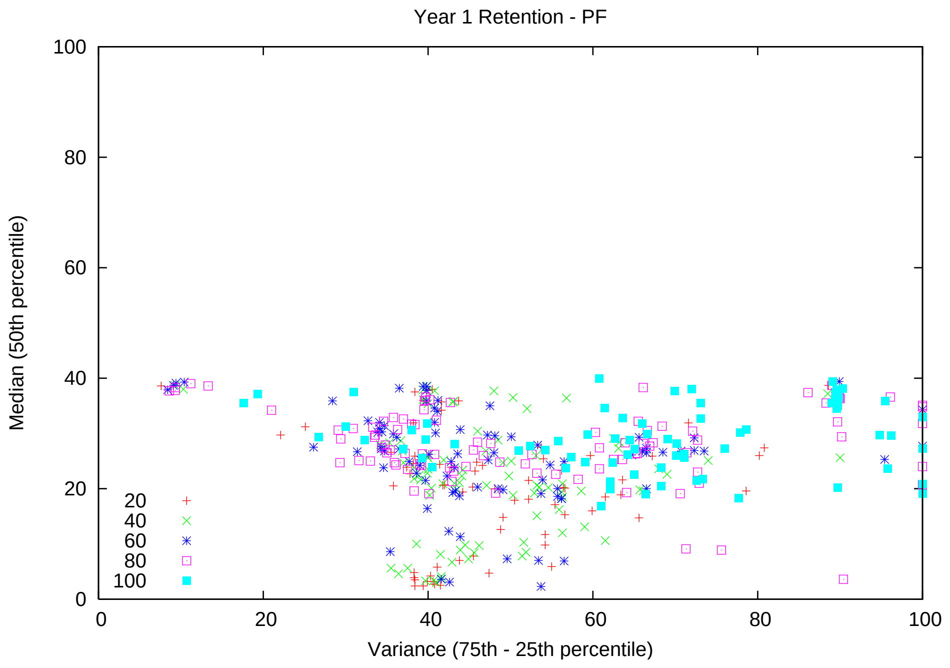
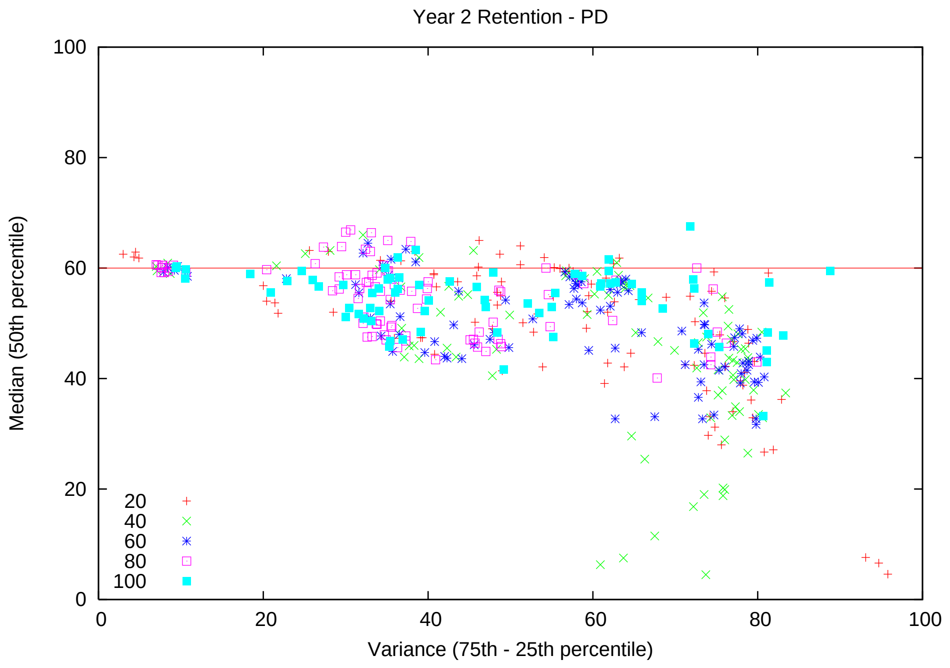


Figure 6.3: Probability of Detection (PD) and Probability of False Alarm (PF) with variances for first year retention.



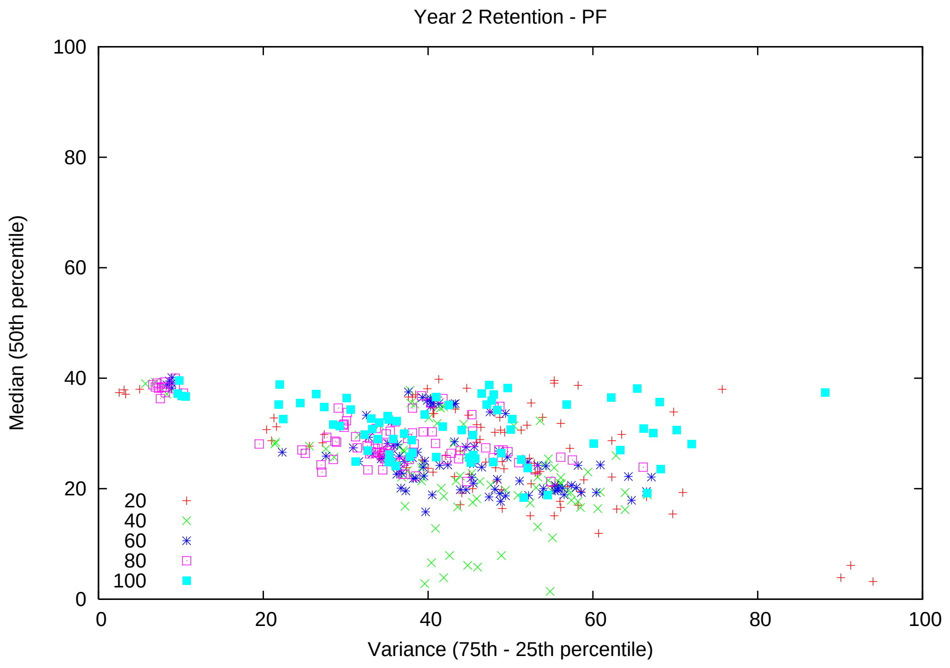
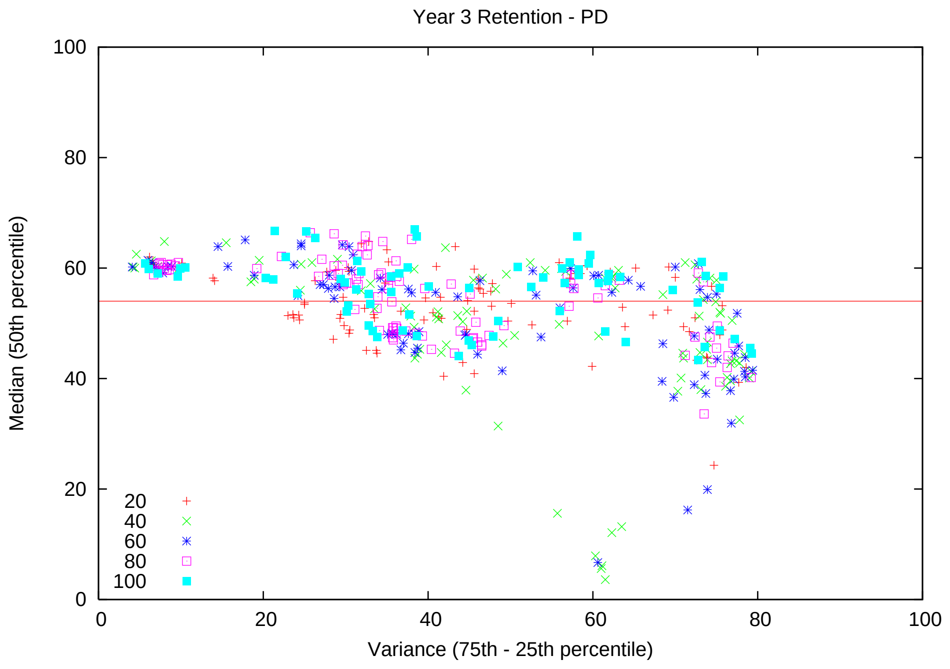


Figure 6.4: Probability of Detection (PD) and Probability of False Alarm (PF) with variances for second year retention.



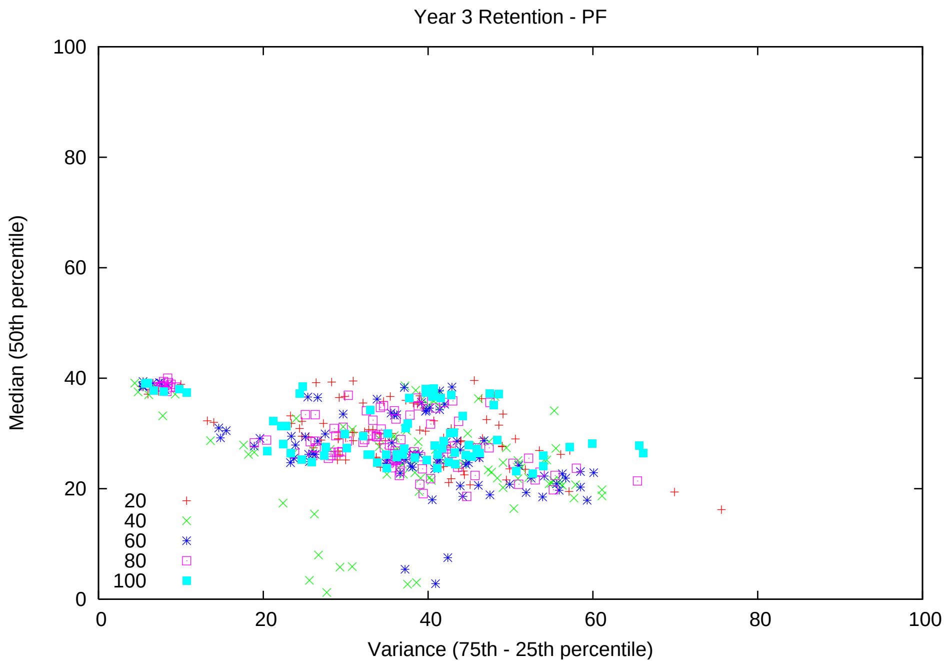


Figure 6.5: Probability of Detection (PD) and Probability of False Alarm (PF) with variances for third year retention.

6.1.8 Visualizing the Results

Figures 6.3, 6.4, and 6.5 show the PD and PF median results for first, second and third year retention against the variance of these values. Each point represents a specific combination of the number of attributes selected, the feature subset selector used to select them, and the classifier used to train on the resulting data. For example, one point on a graph could be seen as 50/Information Gain/Naive Bayes, where 50 denotes the number of attributes used. The color of each point shows the number of attributes used for that particular combination representing that point.

The horizontal line segmenting the PD graphs are given as a baseline reference designated by the already existing retention rates in the data. Thus, to predict for retention in a given year, it is desirable to yield results higher than the baseline. As can be seen in the figures, the median probability of detection of retention values for the first year do not meet the baseline, and therefore we can assume that first year retention cannot accurately be predicted for using our methods. Second year retention provides better results than first year retention, but these results are hardly significant. For example, most of the points lie at or below the baseline. For this reason, second year retention is also not considered in further analysis. Lastly, third year PD values successfully exceed the baseline, and so require more thorough examination.

6.1.9 Narrowing the Search

From the visualizations described above, we can narrow our space of possible combinations to examine for third year retention. The graphs for PD and PF medians show that the range of number of attributes that maximizes PD and minimizes PF values while maintaining minimal variance is approximately 20 to 60. This is significant, as it allows filtering of the results so that concentration can be placed on only treatments whose attribute numbers lie in this range.

Rank	Number of Attributes	FSS	Classifier
61	30	oneR	bnet
61	50	cfs	adtree
57	50	oneR	adtree
56	30	oneR	adtree
55	30	cfs	adtree
52	50	oneR	bnet
51	30	infogain	adtree
51	30	cfs	bnet
48	50	infogain	adtree

Figure 6.6: The top ten ranking treatments for third year retention. Ranks represent how many times a particular treatment wins over all other treatments in the experiment.

Ranking with the Mann-Whitney Test

At the moment of pruning the results based on attribute ranges, we are left with many combinations to be analyzed. In order to rank each combination, I performed a statistical Mann-Whitney test at 95% confidence in order to rank a treatment. A rank is determined by how many times a combination wins compared to another. The method that won the most number of times is then given the highest rank. The table in Figure 6.6 shows the top ten ranking combinations based on a PD performance measure. Note that identical ranks are given to those treatments whose win value is equal in magnitude.

6.1.10 Selected FSS and Classifier

Figure 6.6 shows the top-most ranking combination of FSS and classifier is obtained by either using 30 attributes, or 50. Since, the two numbers of attributes (along with their own FSS and classifier) result in the same Mann-Whitney rank, we can make the statement that the two are statistically similar, and thus by focusing on only 30 attributes selected, we can concentrate on approximately 1/3 of the original data. Thus, our analysis of the results show that 30 attributes selected using One-R as the feature subset selection method are the most critical to third year retention.

Chapter 7

Case Study 4: Component vs. Whole-based Defect Prediction

7.1 Component vs. Whole-based Defect Prediction

7.1.1 The Experiment

In order to test the implications of learning using components dense with software defects, an experiment was constructed using five NASA defect data sets (CM1, KC1, MC1, PC1, PC3). These data sets were chosen because they have been studied in the field extensively, and also that they are widely available to the PROMISE community. Five were chosen due to the limited number of data sets containing noteworthy numbers of components.

For each data set, components are extracted by first determining both defective and non-defective modules contained in that data set. Once the modules are obtained, those components (named after a unique identifier) containing these modules are selected for further analyses.

After extracting the components and corresponding number of defects, components were retrieved for further analysis from each data set if the number of defective modules per component

```
1 For run = 1 to 10
2   For each dense component C in data set D

3    Let Train = C
4    Let Test = All components in D except for C

5     For bin = 1 to 10
6      Train′ = Randomly select 90% modules from Train
7      Test′ = Randomly select 10% modules from Test

8      Learn using the Naive Bayes classifier using Train′ and Test′
9     end bin
10   end component
11 end run
```

Figure 7.1: Training on dense components versus all components. The experiment performs training on modules residing in dense components, and testing on modules contained in all other components in the data set.

exceeded the median number of defects across *all* components in that data set. For example, in Figure 7.2 the bottom horizontal line represents the median number of defects in the KC1 data set. Thus, those components lying under this line are not used in further stages of the experiment. The components selected (at or above the median number of defects) are considered *dense* components. The pseudocode in Figure 7.1 illustrates the remaining setup of the experiment:

Lines 1 and 5 of Figure 7.1 illustrate the use of the 10X10-way cross validation used in the experimental process. The standard 10X10-way cross validation operates by selecting 90% of the data randomly for training, and the remaining 10% for testing. This process is then repeated 10 times for consistency. The experiment shown in Figure 7.1, however, handles this operation in a slightly different manner. Since the objective is to analyze the performance of training on modules in components containing a high number of defective modules while testing on all other components' modules, a minute alteration was made to the cross-validation of the experiment. A "pool" of training data was constructed by focusing on only those instances *within* a dense component, as in line 3 of the psuedocode. The available pool of testing instances, thus, are gathered from the *remaining* components in the data set. This is employed to prevent training, and then testing on modules within the same component. Lines 6 and 7 illustrate collecting 90% of the current dense component's instances as the final training set $Train'$, and 10% of the modules from the available instances in components not labeled *dense* as $Test'$.

While this represents a slight modification to the standard pratice of performing a cross-validation, it is within our engineering judgement to apply techniques that best mimick current methods in an area of experimentation still in its infancy. Thus, the recentness of this specific area of research invites further techniques to be discovered and implemented.

Line 8 of Figure 7.1 executes the classifier (in this case, Naïve Bayes) on the previously created training and testing sets $Train'$ and $Test'$. Naïve Bayes was used because of its speed, and also because it has been shown to perform well on PROMISE defect data against other learners [20].

As the overall goal is to determine if training our classifiers using fewer, but more densely-packed components is advantageous to the usual practice of learning on a pool of all components (and thus all modules), comparisons are made between the two approaches, as well as standard over and under sampling of the data. The results are shown in the following section.

7.1.2 Results

The metrics used in the analysis of comparing results from training on dense components over the traditional method of using all components in the data set are *pd* (Probability of Detection), *pf* (Probability of False Alarm) and *precision*. If A,B,C, and D denote the true negatives, false negatives, false positives and true positives (respectively) found by a classifier, then:

$$pd = Recall = \frac{D}{(B+D)} \tag{7.1}$$

and

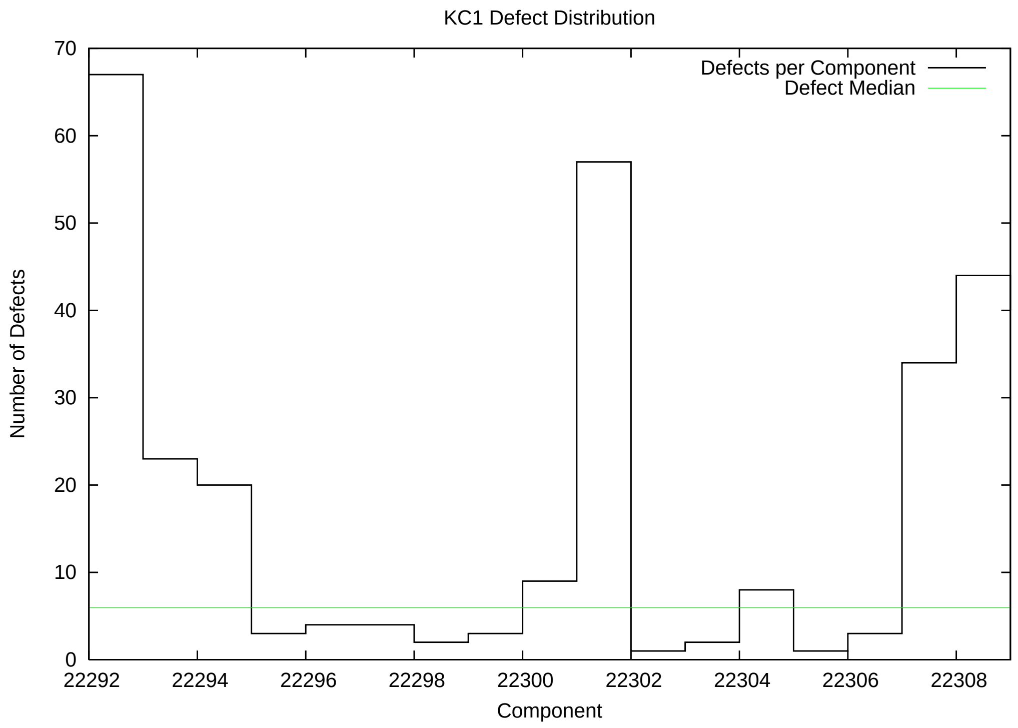


Figure 7.2: Defect distributions of components found in the KC1 data set. Note that only a small number of components contain a relatively high number of defects.

$$pf = \frac{C}{(A+C)} \tag{7.2}$$

and

$$precision = \frac{D}{(D+C)} \tag{7.3}$$

Therefore, *pd* and *precision* values are best if maximized, while pf results should be minimized.

Figure 7.4, Figure 7.5 and Figure 7.6 show statistical rankings of each treatment, as well as quartile charts displaying the median and variance of each metric for the *combined* data sets, as a whole, used in the experiment. Note that training on components containing a higher number of defective modules maintains higher or tied ranks with the traditional method, and yields similar medians; while *precision* and *pd* medians lose 3% and 2% respectively, learning on dense areas provides much better pf medians – almost half.

Perhaps more interestingly are the analyses of data sets separately. Table 7.1 demonstrates the outcome of each treatment for each data set independently. A summary table of these results is shown in Figure 7.3, where"+" denotes a *win* for a particular treatment against the other, per data set. Conversely, a "-" indicates a *loss*, and "0" represents a *tie*. For example, the fifth row in Table 7.1 (data set PC3), shows that learning on dense components wins over all other methods for pf, but loses in recall and ties in precision. A win, loss or tie is assigned to a treatment by examining its statistical rank value (according to the Mann-Whitney test) in comparison to the opposing treatments.

The results from this table demonstrate that learning on only components containing higher numbers of defective modules is beneficial because

- defect prediction performance is improved significantly

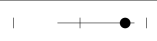
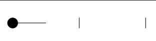
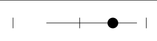
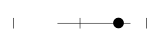
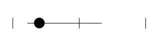
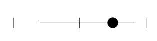
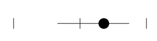
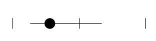
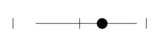
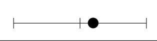
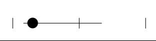
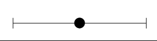
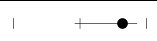
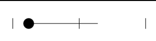
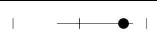
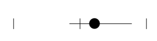
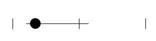
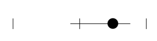
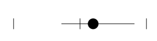
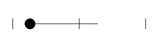
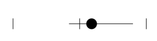
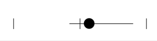
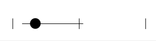
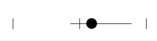
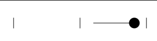
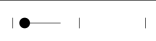
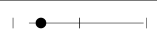
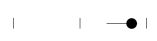
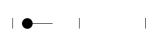
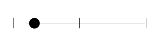
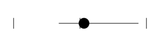
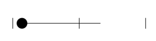
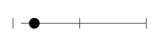
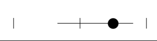
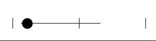
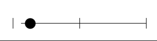
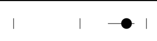
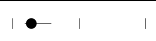
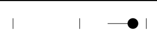
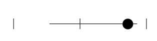
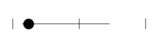
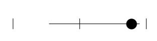
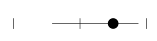
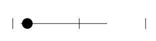
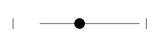
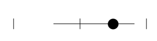
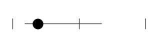
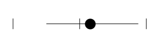
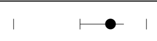
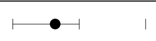
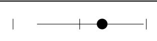
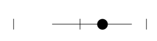
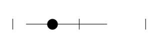
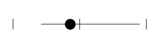
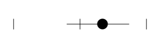
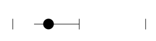
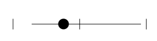
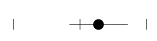
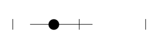
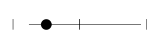
Project	**Recall**			**Prob. False Alarm (Pf)**			**Precision**		
			0% 50% 100%			0% 50% 100%			0% 50% 100%
CM1	1	All		1	Dense		1	All	
	2	Over		2	Over		1	Over	
	2	Under		2	Under		1	Under	
	3	Dense		2	All		1	Dense	
KC1	1	Dense		1	All		1	Dense	
	1	Under		1	Over		1	All	
	1	All		1	Under		1	Under	
	1	Over		1	Dense		1	Over	
MC1	1	Over		1	Over		1	Dense	
	1	Under		1	Under		1	All	
	2	All		2	All		1	Under	
	3	Dense		3	Dense		1	Over	
PC1	1	Dense		1	Dense		1	Dense	
	2	Over		2	Over		2	All	
	2	All		2	All		2	Under	
	3	Under		3	Under		3	Over	
PC3	1	Under		1	Dense		1	Dense	
	1	Over		2	Over		1	Under	
	2	All		2	Under		1	All	
	2	Dense		3	All		1	Over	

Table 7.1: Result statistics per data set. The numeric value next to each treatment represents its Mann-Whitney rank, to the right of each treatment lies the quartile chart for each. Each metric is either sorted by ranking, or in the case of a tie, descending *pd* and *prec* or ascending pf.

- less data is required during the training phase, meaning faster runtimes and results
- insight is provided for component types; software organizations can make informed decisions about how to approach certain problematic components

data set	performance measure	all components	dense components	over sampling	under sampling
CM1	precision	0	0	0	0
	recall	+	-	-	-
	pf	-	+	-	-
KC1	precision	0	0	0	0
	recall	0	0	0	0
	pf	0	0	0	0
MC1	precision	0	0	0	0
	recall	-	-	0	0
	pf	-	-	0	0
PC1	precision	-	+	-	-
	recall	-	+	-	-
	pf	-	+	-	-
PC3	precision	0	0	0	0
	recall	-	-	0	0
	pf	-	+	-	-
summary	+	1	5	0	0
	0	6	6	9	9
	-	8	4	6	6

Figure 7.3: Each treatment is assigned one of $\{+, 0, -\}$ depending on if it *won*, *tied*, *lost* in the statistical rankings of Table 1 (based on a Mann-Whitney test at 95% confidence). Note that *dense* won five times more often than *all*, and lost the least amount of times compared to all other treatments (4/15 vs. 8/15 and 6/15).

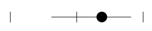
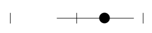
Rank	Treatment	pd percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	Train on Dense Components	31	69	91	
2	Train on All Components	35	71	93	
					0 50 100

Figure 7.4: *PD* values for learning on dense components compared to learning on all components across all data sets, sorted by statistical ranking via a Mann-Whitney test at 95% confidence.

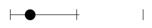
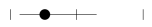
Rank	Treatment	pf percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	Train on Dense Components	0	15	52	
1	Train on All Components	0	26	65	
					0 50 100

Figure 7.5: *PF* values for learning on dense components compared to learning on all components across all data sets, sorted by statistical ranking via a Mann-Whitney test at 95% confidence.

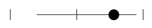
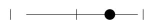
Rank	Treatment	precision percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
1	Train on All Components	20	78	95	
1	Train on Dense Components	12	75	96	
					0 50 100

Figure 7.6: *Precision* values for learning on dense components compared to learning on all components across all data sets, sorted by statistical ranking via a Mann-Whitney test at 95% confidence.

Chapter 8

Case Study 5: Analyzing the Scalability of Clustering Text Documents

8.1 Analyzing the Scalability of Clustering Text Documents

As stated above, the purpose of this experiment conducted for this thesis is to verify if lightweight data mining methods perform worse than more thorough and rigorous ones.

The data sets used in this experiment are:

- EXPRESS schemas: AP-203, AP-214
- Text mining datasets: BBC, Reuters, The Guardian (multi-view text datasets), 20 Newsgroup subsets: sb-3-2, sb-8-2, ss-3-2, sl-8-2

Classes of Methods

This experiment compares different *row* and *column* reduction methods. Given a table of data where each row is one example and each columns counts different features, then:

- Row reduction methods *cluster* related rows into the same group;

- Column reduction methods remove columns with little information.

Reduction methods are essential in text mining. For example:

- A standard text mining corpus may store information in tens of thousands of columns. For such data sets, column reduction is an essential first step before any other algorithm can execute
- The process of clustering data into similar groups can be used in a wide variety of applications, such as:
  - Marketing: finding groups of customers with similar behaviors given a large database of customer data
  - Biology: classification of plants and animals given their features
  - WWW: document classification and clustering weblog data to discover groups of similar access patterns.

The Algorithms

While there are many clustering algorithms used today, this experiment focused on three: a naive K-Means implementation, GenIc [16], and clustering using canopies [22]:

1. K-means, a special case of a class of EM algorithms, works as follows:
   (a) Select initial *K* centroids at random;
   (b) Assign each incoming point to its nearest centroid;
   (c) Adjusts each cluster's centroid to the mean of each cluster;
   (d) Repeat steps 2 and 3 until the centroids in all clusters stop moving by a noteworthy amount

[2] `http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/`

Here we use a naive implementation of K-means, requiring K*N *I comparisons, where N and I represent the total number of points and maximum iterations respectively.

2. GenIc is a single-pass, stochastic clustering algorithm. It begins by initially selecting K centroids at random from all instances in the data. At the beginning of each generation, set the centroid weight to one. When new instances arrive, nudge the nearest centroid to that instance and increase the score for that centroid. In this process, centroids become "fatter" and slow down the rate at which they move toward newer examples. When a generation ends, replace the centroids with less than X percent of the max weight with N more random centroids. Genic repeats for many generations, then returns the highest scoring centroids.

3. Canopy clustering, developed by Google, reduces the need for comparing all items in the data using an expensive distance measure, by first partitioning the data into overlapping subsets called *canopies*. Canopies are first built using a cheap, approximate distance measure. Then, more expensive distance measures are used inside of each canopy to cluster the data.

As to column reduction, I will focus on two methods:

1. PCA, or Principal Components Analysis, is a reduction method that treats every instance in a dataset as a point in N-dimensional space. PCA looks for new dimensions that better fit these points– by mapping data points to these new dimensions where the variance is found to be maximized. Mathematically, this is conducted by utilizing eigenvalue decompositions of a data covariance matrix or singular value decomposition of a data matrix. Figure 8.1 shows an example of PCA. Before, on the left-hand-side, the data exists in a two-dimensional space, neither of which captures the distribution of the data. Afterwards, on the right-hand-side, a new dimension has been synthesized that is more relevant to the data distribution.

2. TF-IDF, or term frequency times inverse document frequency, reduces the number of terms (dimensions) by describing how important a term is in a document (or collection of docu-

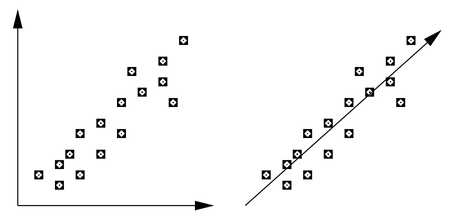


Figure 8.1: A PCA dimension feature.

ments) by incrementing its importance according to how many times the term appears in a document. However, this importance is also offset by the frequency of the term in the entire corpus. Thus, we are concerned with only terms that occur frequently in a small set of documents, and very infrequently everywhere else. To calculate the Tf*IDF value for each term in a document, I use the following equation:

$$Tf * df(t, D_j) = \frac{tf(t_i, D_j)}{|D_j|} log(\frac{|D|}{df(t_i)}) \tag{8.1}$$

where $tf(t_i, D_j)$ denotes the frequency of term i in document j, and $df(t_i)$ represents the number of documents containing term i. Here, $|D|$ denotes the number of documents in the corpus. To reduce all terms (and thus, dimensions), we must find the sum of the above

$$Tf * Ifd_{sum}(t) = \sum_{D_j \varepsilon D} Tf * Idf(t, D_j) \tag{8.2}$$

In theory, TF*IDF and GenIc should perform worse than K-Means, canopy clustering and PCA:

- Any single-pass algorithm like GenIc can be confused by "order effects"; i.e. if the data arrives in some confusing order then the single-pass algorithm can perform worse than other algorithms that are allowed to examine all the data.
- TF*IDF is a heuristic method while PCA is a well-founded mathematical technique

On the other hand, the more rigorous methods are slower to compute:

- Computing the correlation matrix used by PCA requires at least a $O(N^2)$ calculation.
- As shown below, K-means is much slower than the other methods studied here.

Building the Experiment

This experiment was conducted entirely with OURMINE using a collection of BASH scripts, as well as custom Java code. The framework was built as follows:

1. A command-line API was developed in Java for parsing the data, reducing/clustering the data, and outputting the data. Java was chosen due to its preferred speed for the execution of computationally expensive instructions.
2. The data was then iteratively loaded into this Java code via shell scripting. This provides many freedoms, such as allowing parameters to be altered as desired, as well as outputting any experimental results in any manner seen fit.

Figure 8.2 shows the OURMINE code for clustering data using the K-means algorithm. Shell scripting provides us with much leverage in this example. For instance, by looking at Lines 2-5, we can see that by passing the function four parameters, we can cluster data in the range from *minK* to *maxK* on all data in *dataDir*. This was a powerful feature used in this experiment, because it provides the opportunity to run the clusterer across multiple machines simultaneously. As a small example, suppose we wish to run K-means across three different machines with a minimum *K* of 2 and a maximum *K* of 256. Since larger values of *K* generally yield longer runtimes, we may wish to distribute the execution as follows:

```
Machine 1: clusterKmeansWorker 256 256 0 dataDir
Machine 2: clusterKmeansWorker 64 128 2 dataDir
Machine 3: clusterKmeansWorker 2 32 2 dataDir
```

```
1 clusterKmeansWorker(){
2     local minK=$1
3     local maxK=$2
4     local incVal=$3
5     local dataDir=$4
6     local stats="clusterer,k,dataset,time(seconds)"
7     local statsfile=$Save/kmeans_runtimes
8     echo $stats >> $statsfile
9     for((k=$minK;k<=$maxK;k*=$incVal)); do
10         for file in $dataDir/*.arff; do
11             filename=`basename $file`
12             filename=${filename%.*}
13             out=kmeans_k="$k"_$filename.arff
14             echo $out
15             start=$(date +%s.%N)
16             $Clusterers -k $k $file $Save/$out
17             end=$(date +%s.%N)
18             time=$(echo "$end - $start" | bc)
19             echo "kmeans,$k,$filename,$time" >> $statsfile
20         done
21     done
22 }
```

Figure 8.2: An OURMINE worker function to cluster data using the K-means algorithm. Note that experiments using other clustering methods (such as GenIc and Canopy), could be conducted by calling line 16 above in much the same way, but with varying flags to represent the clusterer.

Lines 9-13 of Figure 8.2 load the data from *dataDir* for every *k*, and formats the name of the output file. Then, lines 15-19 begin the timer, cluster the data, and output statistical information such as *k*, the dataset, and runtime of the clusterer on that data set. This file will then be used later in the analysis of these clusters.

Similarly, the flags in line 16 can be changed to perform a different action, such as clustering using GenIc or Canopy, by changing *-k* to *-g* or *-c* respectively, as well as finding cluster similarities (as described below) and purities, by using *-sim* and *-purity* as inputs.

Since any number of variables can be set to represent different libraries elsewhere in OURMINE, the variable

```
$Reducers
```

is used for the dimensionality reduction of the raw dataset, as seen in Figure 8.3, whose overall structure is very similar to Figure 8.2.

```
reduceWorkerTfidf(){
    local datadir=$1
    local minN=$2
    local maxN=$3
    local incVal=$4
    local outdir=$5
    local runtimes=$outdir/tfidf_runtimes

   for((n=$minN;n<=$maxN;n+=$incVal)); do
       for file in $datadir/*.arff; do
            out=`basename $file`
            out=${out%.*}
            dataset=$out
            out=tfidf_n="$n"_$out.arff
            echo $out
            start=$(date +%s)
            $Reducers -tfidf $file $n $outdir/$out
            end=$(date +%s)
            time=$((end - start))
            echo "tfidf,$n,$dataset,$time" >> $runtimes
        done
    done
}
```

Figure 8.3: An OURMINE worker function to reduce the data using TF-IDF.

8.1.1 Results

To determine the overall benefits of each clustering method, this experiment used both cluster similarities, as well as the runtimes of each method.

Similarities

Cluster similarities tell us how similar points are, either within a cluster (*Intra*-similarity), or with members of other clusters (*Inter*-similarity). The idea here is simple: gauge how well a clustering algorithm groups similar documents, and how well it separates different documents. Therefore, intra-cluster similarity values should be maximized, while minimizing inter-cluster similarities.

Similarities are obtained by using the cosine similarity between two documents. The cosine similarity measure defines the cosine of the angle between two documents, each containing vectors of terms. The similarity measure is represented as

$$sim(D_i, D_j) = \frac{D_i \cdot D_j}{||D_i|| \, ||D_j||} = cos(\theta) \tag{8.3}$$

Reducer and Clusterer	Time	InterSim	IntraSim	Gain
TF-IDF*K-means	17.52	-0.085	141.73	141.82
TF-IDF*GenIc	3.75	-0.14	141.22	141.36
PCA*K-means	100.0	0.0	100.0	100.0
PCA*Canopy	117.49	0.00	99.87	99.87
PCA*GenIc	11.71	-0.07	99.74	99.81
TF-IDF*Canopy	6.58	5.02	93.42	88.4

Figure 8.4: Experiment #3 (Text mining). Similarity values normalized according to the combination of most rigorous reducer and clusterer. Note that *Gain* is a value representing the difference in cluster intrasimilarity and intersimilarity.

where D_i and D_j denote two term frequency vectors for documents i and j, and where the denominator contains magnitudes of these vectors.

Cluster similarities are determined as follows:

- Cluster intra-similarity: For each document d in cluster C_i, find the cosine similarity between d and all documents belonging to C_i
- Cluster inter-similarity: For each document d in cluster C_i, find the cosine similarity between d and all documents belonging to all other clusters

Thus the resulting sum of these values represents the overall similarities of a clustering solution. Figure 8.4 shows the results from the similarity tests conducted in this experiment. The slowest clustering and reduction methods were set as a baseline, because it was assumed that these methods would perform the best. With intra-similarity and inter-similarity values normalized to 100 and 0 respectively, we can see that surprisingly, faster heuristic clustering and reduction methods perform just as well or better than more rigorous methods. *Gain* represents the overall score used in the assessment of each method, and is computed as a method's cluster $intra$-similarity value minus its $inter$-similarity value. Thus, the conclusions from this experiment shows that fast heuristic methods are sufficient for large data sets due to their scalability and performance.

Chapter 9

Conclusion

In this thesis, I have shown OURMINE to be an adequate tool for use in a variety of ways.

In terms of research, the ability to succinctly represent complex experiments assists in reproducibility and verification. Thus, the next challenge in the empirical SE community will be to not only share data, but to share experiments. My colleagues and I look forward to the day when it is routine for conference and journal submissions to come not just with supporting data but also with a fully executable version of the experimental rig used in the paper. Ideally, when reviewing papers, program committee members could run the rig and check if the results recorded by the authors are reproducible.

Thus, I have used OURMINE to reproduce or check several important result. In Chapter 8, I conducted a massive text mining experiment and showed that:

- When examining cluster inter/intra similarities resulting from each clustering/reduction solution, we found that faster heuristic methods can outperform more rigorous ones when observing decreases in runtimes.
- This means that faster solutions are suitable on large data sets due to *scalability*, as well as performance.

In Chapter 5, I reproduced a software defect prediction experiment and concluded that:

- When local data is available, that data should be used to build defect predictors
- If local data is not available, however, imported data can be used to build defect predictors when using *relevancy filtering*
- Imported data that uses *relevancy filtering* performs nearly as well as using local data to build defect predictors

In Chapter 7, I demonstrated that by learning on highly defective components:

- Defect prediction performance is improved significantly
- Less data is required during the training phase, meaning faster runtimes and results
- Insight is provided for component types; software organizations can make informed decisions about how to approach certain problematic components

OURMINE also provides a way for practitioners to utilize the environment to better suit industrial needs. This is important because it demonstrates OURMINE's well-roundedness as a competitive data mining toolkit. More specifically, it illustrates the usability of the scripting-only environment for simple as well as complex tasks. To convey its usability in industry, an industrial type experiment was conducted in Chapter 4 using software defect data. From that experiment it was shown how practitioners can:

- First, "tune" our predictors to best fit a data set or corpus, and to select the winning method based on results.
- Secondly, understand how *early* we can apply these winning methods to our data by determining the fewest number of examples required in order to learn an adequate theory.

My data mining colleagues and I at WVU prefer OURMINE to other tools. Four features are worthy of mention:

1. OURMINE is very succinct. As seen above, a few lines can describe even complex experiments.

2. OURMINE's experimental descriptions are complete. There is nothing hidden in Figure 5.1; it is not the pseudocode of an experiment, it <u>is</u> the experiment.

3. OURMINE code like in Figure 8.2, Figure 8.3 and Figure 5.1 is executable and can be executed by other researchers directly.

4. Lastly, the execution environment of OURMINE is readily available. Unlike RAPID-I, WEKA, "R", etc, there is nothing to debug or install. Many machines already have the support tools required for OURMINE. For example, we have run OURMINE on Linux, Mac, and Windows machines (with Cygwin installed).

Like Ritthol et al., I doubt that the standard interfaces of tools like WEKA, etc, are adequate for representing the space of possible experiments. Impressive visual programming environments are not the answer: their sophistication can either distract or discourage novice data miners from extensive modification and experimentation. Also, I find that the functionality of the visual environments can be achieved with a few BASH and GAWK scripts, with a fraction of the development effort and a greatly increased chance that novices will modify the environment.

OURMINE is hence a candidate format for research, education, industry and personal use.

Appendix A

Installing and Running Ourmine

OURMINE is an open source toolkit licensed under GPL 3.0. It can be downloaded and installed from `http://code.google.com/p/ourmine`.

OURMINE is a command-line environment, and as such, system requirements are minimal. However, in order to use OURMINE three things must be in place:

- A Unix-based environment. This does not include Windows. Any machine with OSX or Linux installed will do.
- The Java Runtime Environment. This is required in order to use the WEKA, as well as any other Java code written for OURMINE.
- The GAWK Programming Language. GAWK will already be installed with up-to-date Linux versions. However, OSX users will need to install this.

To install and run OURMINE, navigate to `http://code.google.com/p/ourmine` and follow the instructions.

Appendix B

Built-in Functions and Data

Utility Functions I

Function Name	Description	Usage
abcd	Performs confusion matrix computations on any classifier output. This includes statistics such as $pd, $pf, $accuracy, $balance and $f-measure	— *abcd –prefix –goal*, where $prefix$ refers to a string to be inserted before the result of the *abcd* function, and *goal* is the desired class of a specific instance.
arffToLisp	Converts a single .arff file into an equivalent .lisp file	*arffToLisp $dataset.arff*
blab	Prints to the screen using a separate environment. This provides the ability to print to the screen without the output interfering with the results of an experiment	*blab $message*
blabln	The same as blab, except this will print a new line after the given output	*blabln $message*

docsToSparff	Converts a directory of document files into a sparse .arff file. Prior to building the file, however, the text is cleaned	*docsToSparff $docDirectory $output.sparff*
docsToTfidfSparff	Builds a sparse .arff file from a directory of documents, as above, but instead constructs the file based on TF-IDF values for each term in the entire corpus.	*docsToTfidfSparff $docDirectory $numberOfAttributes $output.sparff*
formatGotWant	Formats an association list returned from any custom LISP classifier containing actual and predicted class values in order to work properly with existing OURMINE functions	*formatGotWant*
funs	Prints a sorted list of all available OURMINE functions	*funs*
getClasses	Obtains a list of all class values from a specific data set	*getClasses*
getDataDefun	Returns the name of a .arff relation to be used to construct a LISP function that acts as a data set	*getDataDefun*
gotwant	Returns a comma separated list of actual and predicted class values from the output of a WEKA classifier	*gotwant*
help	When given with an OURMINE function, prints helpful information about the function, such as a description of the function, how to use it, etc.	*help $function*, where $function is the name of the function

Utility Functions II

Function Name	Description	Usage
makeQuartiles	Builds quartile charts using any key and performance value from the abcd results (see above)	*makeQuartiles $csv $keyField $performanceField*, where $keyField can be a learner/treatment, etc., and $performanceField can be any value desired, such as *pd*, *accuract*, etc.
makeTrainAndTest	Constructs a training set and a test set given an input data set. The outputs of the function are train.arff, test.arff and also train.lisp and test.lisp	*makeTrainAndTest $dataset $bins $bin*, where $dataset refers to any data set in correct .arff format, $bins refers to the number of bins desired in the constuction of the sets, and $bin is the bin to select as the test set. For instance, if 10 is chosen as the number of bins, and 1 is chosen as the test set bin, then the resulting training set would consist of 90% of the data, and the test set would consist of 10%.
malign	Neatly aligns any comma-separated format into an easily readable format	*malign*
medians	Computes median values given a list of numbers	*medians*

quartile	Generates a quartile chart along with min/max/median values, as well as second and third quartile values given a specific column	*quartile*
show	Prints an entire OURMINE function so that the script can be seen in its entirety	*show $functionName*
winLossTie	Generates win-loss-tie tables given a data set. Win-loss-tie tables, in this case, depict results after a statstical analysis test on treatments. These tests include the Mann-Whitney-U test, as well as the Ranked Wilcoxon test	*winLossTie –input $input.csv –fields $numOfFields –perform $performanceField –key $keyField –$confidence*, where $input.csv refers to the saved output from the *abcd* function described above, $numOfFields represents the number of fields in the input file, $performanceField is the field on which to determine performance, such as *pd*, pf, *acc*, $keyField is the field of the key, which could be a learner/treatment, etc., and $confidence is the percentage of confidence when running the test. The default confidence value is 95%

Learners

Function Name	Description	Usage
adtree	Calls WEKA's Alternating Decision Tree	*adtree $train $test*
bnet	Calls WEKA's Bayes Net	*bnet $train $test*
j48	Calls WEKA's J48	*j48 $train $test*
nb	Calls WEKA's Naïve Bayes	*nb $train $test*
oner	Calls WEKA's One-R	*oner $train $test*
rbfnet	Calls WEKA's RBFNet	*rbfnet $train $test*
ridor	Calls WEKA's RIDOR	*ridor $train $test*
zeror	Calls WEKA's Zero-R	*zeror $train $test*

Preprocessors

Function Name	Description	Usage
caps	Reduces capitalization to lowercase from an input text	*caps*
clean	Cleans text data by removing capitals, words in a stop list, special tokens, and performing Porter's stemming algorithm	*clean*
discretize	Discretizes the incoming data via WEKA's discretizer	*discretize $input.darff $output.arff*
logArff	Logs numeric data in incoming data	*logArff $minVal $fields*, where $minVal denotes the minimum value to be passed to the log function, and $fields is the specfic fields on which to perform log calculations
stems	Performs Porter's stemming algorithm on incoming text data	*stems $inputFile*
stops	Removes any terms from incoming text data that are in a stored stop list	*stops*
tfidf	Computes TF*IDF values for terms in a document	*tfidf $file*
tokes	Removes unimportant tokens or whitespace from incoming textual data	*tokes*

Feature Subset Selectors

Function Name	Description	Usage
cfs	Calls WEKA's Correlation-based Feature Selector	*cfs $input.arff $numAttributes $out.arff*
chisquared	Calls WEKA's Chi-Squared Feature Selector	*chisquared $input.arff $numAttributes $out.arff*
infogain	Calls WEKA's Infogain Feature Selector	*infogain $input.arff $numAttributes $out.arff*
oneR	Calls WEKA's One-R Feature Selector	*oneR $input.arff $numAttributes $out.arff*
pca	Calls WEKA's Principal Components Analysis Feature Selector	*pca $input.arff $numAttributes $out.arff*
relief	Calls WEKA's RELIEF Feature Selector	*relief $input.arff $numAttributes $out.arff*

Clusterers

Function Name	Description	Usage
K-means	Calls custom Java K-means	*$Clusterers -k $k $input.arff $out.arff, where $k is the initial number of centroids*
Genic	Calls custom Java GeNic	*$Clusterers -g $k $n $input.arff $out.arff, where $k is the initial number of centroids, and $n is the size of a generation*
Canopy	Calls custom Java Canopy Clustering	*$Clusterers -c $k $p1 $p2 $input.arff $out.arff, where k is the initial number of centroids, $p1 is a similarity percentage value for the outer threshold, and $p2 is a similarity percentage value for the inner threshold. If these percentages are not desired, a value of 1 should be provided for both*
EM	Calls WEKA's Expectation-Maximization Clusterer	*em $input.arff $k, where $k is the initial number of centroids*

Bibliography

[1] Adam: Algorithm development and mining system. Available from http://datamining.itsc.uah.edu/adam/index.html.

[2] Databionic esom tool. Available from http://databionic-esom.sourceforge.net/index.html.

[3] The gnome data mine. Available from http://www.togaware.com/datamining/gdatamine/.

[4] Knime: Konstanz information miner. Available from http://www.knime.org/.

[5] Orange. Available from http://www.ailab.si/orange/.

[6] R: A freely available software environment for statistical computing and graphics. Available from http://www.r-project.org/.

[7] Rapidminer. Available from http://rapid-i.com/content/view/181/190/lang,en/.

[8] Rattle: the r analytical tool to learn easily. Available from http://rattle.togaware.com/.

[9] Weka: Waikato environment for knowledge analysis. Available from http://www.cs.waikato.ac.nz/ml/weka/.

[10] Fabian Morchen Alfred Ultsch. Esom-maps: tools for clustering, visualization, and classification with emergent som.

[11] Brian W. Kernighan Alfred V. Aho and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[12] Adrian Bors. *Introduction of the Radial Basis Function (RBF) Networks.*

[13] William Notz David Moore. *Statistics: concepts and controversies*. 2006.

[14] Yoav Freund and Llew Mason. The alternating decision tree learning algorithm. In *In Machine Learning: Proceedings of the Sixteenth International Conference*, pages 124–133. Morgan Kaufmann, 1999.

[15] Greg Gay, Tim Menzies, and Bojan Cukic. How to build repeatable experiments. In *PROMISE '09: Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, pages 1–9, New York, NY, USA, 2009. ACM.

[16] Chetan Gupta and Robert Grossman. Genic: A single pass generalized incremental algorithm for clustering. In *In SIAM Int. Conf. on Data Mining*. SIAM, 2004.

[17] Mark A. Hall. Correlation-based feature selection for discrete and numeric class machine learning. pages 359–366. Morgan Kaufmann, 2000.

[18] David Heckerman. A tutorial on learning with bayesian networks. 1996.

[19] R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.

[20] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, May 2008. Available from `http://iccle.googlecode.com/svn/trunk/share/pdf/lessmann08.pdf`.

[21] R. Loui. Gawk for ai. *Class Lecture*. Available from `http://menzies.us/cs591o/?lecture=gawk`.

[22] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178, New York, NY, USA, 2000. ACM.

[23] T. Menzies. Evaluation issues for visual programming languages, 2002. Available from `http://menzies.us/pdf/00vp.pdf`.

[24] T. Menzies, B. Turhan, A. Bener, and J. Distefano. Cross- vs within-company defect prediction studies. 2007. Available from `http://menzies.us/pdf/07ccwc.pdf`.

[25] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[26] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[27] A.S. Orrego. Sawtooth: Learning from huge amounts of data, 2004.

[28] J. Ross Quinlan. *Induction of decision trees*. 1 edition, 1986.

[29] J. Ross Quinlan. *C4.5: Programs for Machine Learning (Morgan Kaufmann Series in Machine Learning)*. Morgan Kaufmann, 1 edition, January 1993.

[30] Chet Ramey. Bash, the bourne-again shell. 1994. Available from `http://tiswww.case.edu/php/chet/bash/rose94.pdf`.

[31] Juan Ramos. Using tf-idf to determine word relevance in document queries. In *Proceedings of the First Instructional Conference on Machine Learning*, 2003. Available from `http://www.cs.rutgers.edu/~mlittman/courses/ml03/iCML03/papers/ramos.pd%f`.

[32] E. Rish. An empirical study of the naive bayes classifier. In *IJCAI-01 workshop on Empirical Methods in AI*, 2001. Available from `www.intellektik.informatik.tu-darmstadt.de/~tom/IJCAI01/Rish.pdf`.

[33] Irina Rish. An empirical study of the naive bayes classifier. In *IJCAI-01 workshop on "Empirical Methods in AI"*.

[34] O. Ritthoff, R. Klinkenberg, S. Fischer, I. Mierswa, and S. Felske. Yale: Yet another learning environment. In *LLWA 01 - Tagungsband der GI-Workshop-Woche, Dortmund, Germany*, pages 84–92, October 2001. Available from `http://ls2-www.cs.uni-dortmund.de/~fischer/publications/YaleLLWA01.pdf`.

[35] Burak Turhan, Tim Menzies, Ayse B. Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 2009. Available from `http://menzies.us/pdf/08ccwc.pdf`.