SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Sharing Experiments Using Open Source Software

Adam Nelson, Tim Menzies, Gregory Gay*,*

Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, USA

KEY WORDS: open source, data mining

SUMMARY

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 OURMINE's succinctness. For example, this paper presents two experiments 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.

1. Introduction

Since 2004, the authors 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 [†].

The premise of this paper 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

*Correspondence to: Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, USA
*E-mail: rabituckman@gmail.com, tim@menzies.us, gregoryg@csee.wvu.edu
†http://promisedata.org/data

Received 1 October 2009
Revised ?? December 2009

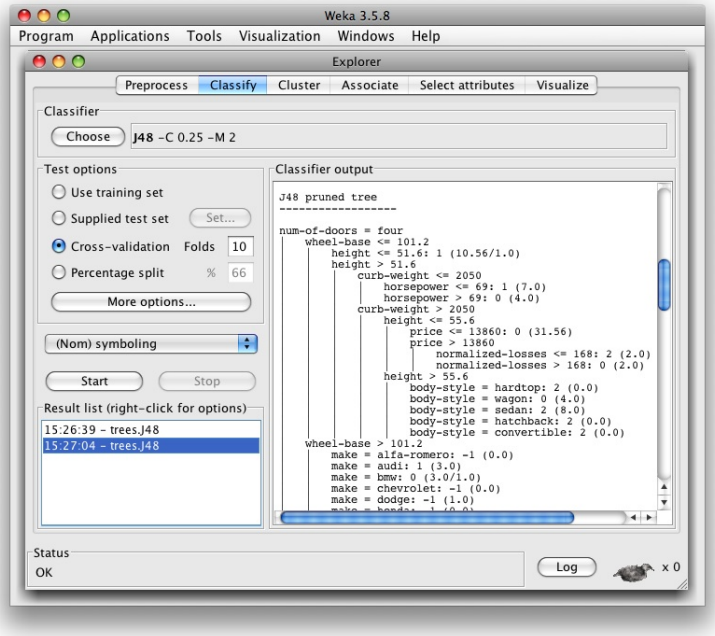

Figure 1. The WEKA toolkit running the J48 decision tree learner.

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.

Consequently, we have been assessing data mining toolkits such as the WEKA[‡] tool of Figure 1, "R"[§], the ORANGE[¶] tool of Figure 2, RAPID-I[‖], MLC++ [**], and MATLAB[††]. We find that they are are not suitable for publishing experiments. Our complaint with these tools is same as that offered by Ritthoff et al. [21]:

- Proprietary tools such as MATLAB are not freely available.
- Real-world experiments are more complex than running one algorithm.

[‡] http://www.cs.waikato.ac.nz/ml/weka/
[§] http://www.r-project.org/
[¶] http://magix.fri.uni-lj.si/orange/
[‖] http://rapid-i.com/
[**] http://www.sgi.com/tech/mlc/
[††] http://www.mathworks.com

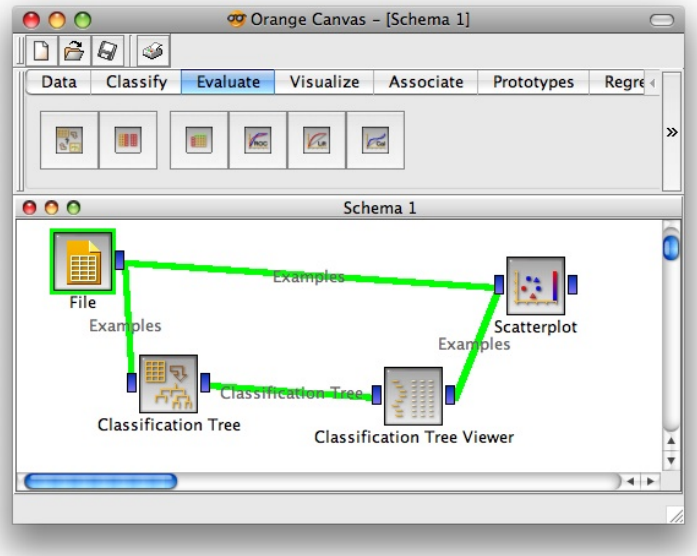

Figure 2. Orange's visual programming environment.

- 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). The YALE tool (now RAPID-I) built by Ritthoff et al. formalizes these visual environments by generating them from XML schemas describing *operator trees* like Figure 3.

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

- These standard data mining toolkits may be overly elaborate. As shown throughout this paper, 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.

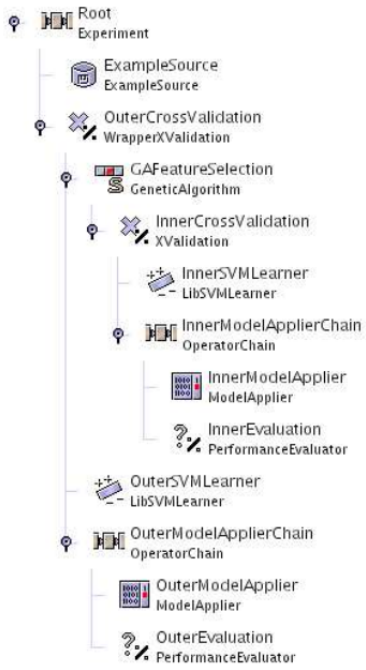

Figure 3. Rapid-I's operator trees. [17]. Internally, this tree is a nested XML expression that is traversed top-down to complete an experiment.

- 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 [12], 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 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, we 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.

Our preference for BASH [19]/GAWK [1] over, say, LISP is partially a matter of taste but we 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.

This paper introduces OURMINE as follows:

- First, we describe the base tool and offer some samples of coding in OURMINE;
- Next, we demonstrate OURMINE's ability to succinctly document even complex experiments.
  - OURMINE is a tool for both practitioners and researchers. Our first demonstration experiment is practitioner-focused and shows how OURMINE can be used in an industrial setting. Following experiments demonstrate OURMINE as an adequate tool for current research.

This paper is an extension of a prior publication [5]:

- This paper details functionality added since that last paper was published;
- This paper adds two new experiments to the description of OURMINE (see §3.1 and §3.3).

2. OURMINE

OURMINE was developed to help graduate students at West Virginia University document and execute 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.

For example, Figure 4 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 4 are written using BASH, Awk and Perl.

The following sections describe OURMINE's functions and applications.

2.1. Built-in Data and Functions

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

```
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 4. 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).

- *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][‡‡]
- *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, 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.

2.2. Learning and 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 5 shows a GAWK implementation used by OURMINE to determine the TF-IDF [20] (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

[‡‡] 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 5. A GAWK implementation of TF-IDF.

additional example demonstrating the brevity of OURMINE script can be seen in Figure 6, which is a complete experiment whose form can easily be taught and duplicated in future experiments.

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 7 shows a GAWK implementation of a Naive 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 [18]. WEKA cannot process these large data sets since it always loads its data into RAM. Figure 7, 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 7 is easily customizable. Figure 8 shows four warm-up exercises for novice data miners that (a) introduce them to basic data mining concepts and (b) show them how easy it is to script their own data miner: Each of these tasks requires changes to less than 10 lines from Figure 7. The simplicity of these 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 Naive 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 Naive 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.

We 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:

```
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 6. 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 7. A Naive Bayes classifier for a CSV file, where the class label is found in the last column.

> *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.* [9]

Function documentation provides a way for newcomers to OURMINE to not only get to know 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 9 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.

1. Modify Figure 7 so that there is no train/test data. Instead, make it an incremental learning. Hint: 1) call the functions *train*, then $classify$ on every line of input. 2) The order is important: always *train* before $classifying$ so the the results are always on unseen data.
2. Convert Figure 7 into HYPERPIPES [3]. Hint: 1) add globals $Max[h,i]$ and $Min[h,i]$ to keep the max/min values seen in every column "i"and every hypothesis class "h". 2) Test instance belongs to the class that most overlaps the attributes in the test instance. So, for all attributes in the test set, sum the returned values from *contains*1:
```
function contains1(h,i,val,numerip) {
  if(numericp)
      return Max[h,i] >= val && Min[h,i] <= val
  else return (h,i,value) in Seen
  }
```
3. Use Figure 7 to implement an anomaly detector. Hint: 1) make all training examples get the same class; 2) an anomalous test instance has a likelihood $\frac{1}{\alpha}$ of the mean likelihood seen during training (*alpha* needs tuning but $alpha = 50$ is often useful).
4. Using your solution to #1, create an incremental version of HYPERPIPES and an anomaly detector.

Figure 8. Four Introductory OURMINE programming exercises.

```
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 9. Function help in OURMINE.

3. Using Ourmine for Industrial and Research Purposes

OURMINE is not just a simple demonstration system for novice data miners. It is also a tool for commercial practitioners and researchers alike. In order to demonstrate OURMINE's utility for practitioners, our first demonstration experiment is an example of selecting the right learner and sample size for a particular domain.

- In part A of this first experiment, we show a very simple example of how practitioners can "tune" learners to data sets in order to achieve better results. This is shown through the comparison of learners (Naive Bayes, J48, One-r, etc.) training on raw data, and on discretized data.

- In part B of this experiment, we demonstrate how *early* our predictors can be applied. This is accomplished through incremental, random sampling of the data, to which the winning method from Part A is applied. The objective here is to report the smallest number of cases required in order to learn an adequate theory.

As software engineering researchers, we use OURMINE to generate publications in leading software engineering journals and conferences. For example, in the last three years, we have used OURMINE in this way to generate [2,5,13,15,16,22]. In order to demonstrate OURMINE's application to research, we present here two further experiments. Experiment #2 reproduces an important recent result by Turhan et al. [22]. Kitchenham et al. report that there is no clear consensus on the relative value of effort estimation models learned from either local or imported data [7]. Since this matter has not been previously studied in the defect prediction literature, Turhan et al. conducted a series of experiments with ten data sets from different companies. In their *within-company* (WC) experiments, defect predictors were learned from 90% of the local data, then tested on the remaining 10%. In the *cross-company* (CC) experiments, defect predictors were learned from nine companies then tested on the tenth. Turhan et al. found that the CC predictions were useless since they suffered from very high false alarm rates. However, after applying a simple *relevancy filter*, they found that imported CC learning yielded predictors that were nearly as good as the local WC learning. That is, using these results, we can say that organizations can make quality predictions about their source code, even when they lack local historical logs.

Experiment #3 checks a recent text mining result from an as-yet-unpublished WVU masters thesis. Matheny [10] benchmarked various lightweight learning methods (TF*IDF, the GENIC stochastic clusterer) against other, slower, more rigorous learning methods (PCA, K-means). As expected, the rigorous learning methods ran much slower than the stochastic methods. But, unexpectedly, Matheny found that the lightweight methods perform nearly as well as the rigorous methods. That is, using these results, we can say that text mining methods can scale to much larger data sets.

The rest of this section describes the use of OURMINE to reproduce Experiment #1, #2 and #3.

3.1. Experiment I: Commissioning a Learner

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. [14]. 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.

3.1.1. Results

Our results for Part A and B of this experiment use pd (probability of detection) values sorted in decreasing order, and pf (probability of false alarm) values sorted in increasing order. These values assess the classification performance of a binary detector, and are calculated as follows. If $\{a, b, c, d\}$ are the true negatives, false negatives, false positives, and true positives (respectively) found by the 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.

Figure 10 and Figure 11 show the results from Part A of the experiment. As can be seen, ADTrees (Alternating Decision Trees) [4] 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 12 and Figure 13. 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

Rank	Treatment	pd percentiles 25%	50%	75%	2nd quartile median, 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	

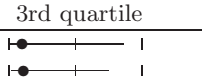

Figure 10. Experiment #1 - Part A - (Learner tuning). Probability of Detection (PD) results, sorted by rank then median values.

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

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.

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	

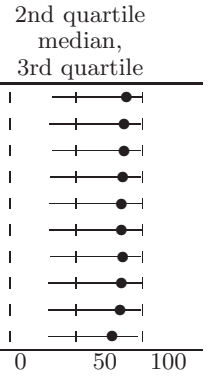

Figure 12. Experiment #1 - Part B - (Random Sampling). Probability of Detection (PD) results, sorted by rank then median values.

Rank	Training Size	pf percentiles 25%	50%	75%	2nd quartile median, 3rd quartile
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	

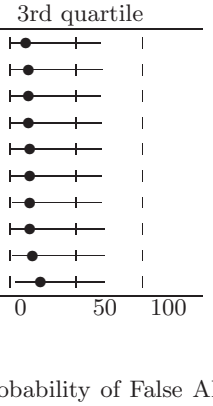

Figure 13. Experiment #1 - Part B - (Random Sampling). Probability of False Alarm (PF) results, sorted by rank then median values.

3.2. Experiment II: Within vs Cross-Company Data

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.

3.2.1. Building the Experiment

The entire script used to conduct this experiment is shown in Figure 14.

To begin, the data in this study were the same as used by Gay et al. [5]; 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 14, 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 $shared\_X.arff$.

We 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 [22], as well as because Naive Bayes has been shown to be competitive on PROMISE defect data against other learners [8].

3.2.2. Results

Our results for this experiment can be found in Figure 15 and Figure 16. 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.

```
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 14. The OURMINE script used in conducting the WC vs. CC experiment.

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 15. 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	WC (local data) + relevancy filter	20	27	34	
2	WC (local data)	17	30	41	
3	CC (imported data) + relevancy filter	17	29	43	
3	CC (imported data)	13	34	51	
					0 50 100

Figure 16. Experiment #2 (WC vs. CC). Probability of False Alarm (PF) results, sorted by rank then median values.

- 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.

3.3. Experiment III: Scaling Up Text Miners

As stated above, the purpose of this experiment conducted for this paper 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

3.3.1. 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.

3.3.2. The Algorithms

While there are many clustering algorithms used today, this experiment focused on three: a naive K-Means implementation, GenIc [6], and clustering using canopies [11]:

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

   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.

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

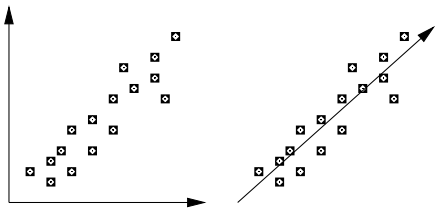

Figure 17. A PCA dimension feature.

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, we 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 17 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 documents) 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, we use the following equation:

$$Tf * Idf(t, D_j) = \frac{tf(t_i, D_j)}{|D_j|} log(\frac{|D|}{df(t_i)}) \quad (1)$$

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, and $|D_j|$ is the total number of terms in document j. To reduce all terms (and thus, dimensions), we must find the sum of the above in order to assign values to terms across *all* documents

$$Tf * Idf_{sum}(t) = \sum_{D_j \epsilon D} Tf * Idf(t, D_j) \quad (2)$$

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.

3.3.3. 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 18 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
```

Lines 9-13 of Figure 18 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
```

```
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 18. 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.

is used for the dimensionality redu ction of the raw dataset, as seen in Figure 19, whose overall structure is very similar to Figure 18.

3.4. Results

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

3.4.1. 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

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

8    for((n=$minN;n<=$maxN;n+=$incVal)); do
9        for file in $datadir/*.arff; do
10            out=‘basename $file‘
11            out=${out%.*}
12            dataset=$out
13            out=tfidf_n="$n"_$out.arff
14            echo $out
15            start=$(date +%s)
16            $Reducers -tfidf $file $n $outdir/$out
17            end=$(date +%s)
18            time=$((end - start))
19            echo "tfidf,$n,$dataset,$time" >> $runtimes
20        done
21    done
22 }
```

Figure 19. An OURMINE worker function to reduce the data using TF-IDF.

$$sim(D_i, D_j) = \frac{D_i \cdot D_j}{||D_i||||D_j||} = cos(\theta) \tag{3}$$

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 20 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

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 20. 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.

from this experiment shows that fast heuristic methods are sufficient for large data sets due to their scalability and performance.

4. Discussion

Productivity of a development environment depends on many factors. In our opinion, among the most important of these are simplicity, power and motivation provided by the environment. There exists an intricate balance between simplicity and power. If a tool is too simple, a user's actions are confined to a smaller space, thus decreasing the overall power and intricacy of the system. Also, exceedingly simple environments can restrict our learning process because we are not forced to learn or implement new ideas. However, even though a tool yields a great number of features, the options presented to a user can be overly complex and thus discourage further experimentation. We find that quick, yet careful experiment building promotes more motivation and interest to work on and finish the task.

Practitioners and researchers can both benefit from using OURMINE's syntax based on widely recognized languages. While practitioners are offered extensive freedom in modifying and extending the environment to be used as necessary, researchers can duplicate previously published experiments whose results could change the way we view data mining and software engineering.

5. Conclusions

The next challenge in the empirical SE community will be to not only share data, but to share experiments. We 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.

This paper has reviewed a UNIX scripting tool called OURMINE as a method of documenting, executing, and sharing data mining experiments. We have used OURMINE to reproduce and check several important result. In our first experiment, we learned that:

- 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.

In our second experiment, we 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 our third experiment, we learned:

- 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 datasets due to *scalability*, as well as performance.

We 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 14; it is not the pseudocode of an experiment, it is the experiment.
3. OURMINE code like in Figure 14, Figure 18 and Figure 19 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., we 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, we 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 sharing descriptions of experiments. The data mining community might find this format unacceptable but discussions about the drawbacks (or strengths) of OURMINE would help evolve not just OURMINE, but also the discussion on how to represent data mining experiments for software engineering.

REFERENCES

1. Brian W. Kernighan Alfred V. Aho and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
2. Zhihao Chen, Tim Menzies, Dan Port, and Barry Boehm. Finding the right data for software cost modeling. *IEEE Software*, Nov 2005.
3. Jacob Eisenstein and Randall Davis. Visual and linguistic information in gesture classification. In *ICMI*, pages 113–120, 2004. Avaliable from http://iccle.googlecode.com/svn/trunk/share/pdf/eisenstein04.pdf.
4. 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.
5. 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.
6. Chetan Gupta and Robert Grossman. Genic: A single pass generalized incremental algorithm for clustering. In *In SIAM Int. Conf. on Data Mining*. SIAM, 2004.
7. B. A. Kitchenham, E. Mendes, and G. H. Travassos. Cross- vs. within-company cost estimation studies: A systematic review. *IEEE Transactions on Software Engineering*, pages 316–329, May 2007.
8. 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.
9. R. Loui. Gawk for ai. *Class Lecture*. Available from http://menzies.us/cs591o/?lecture=gawk.
10. A. Matheny. Scaling up text mining, 2009. Masters thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University.
11. 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.
12. T. Menzies. Evaluation issues for visual programming languages, 2002. Available from http://menzies.us/pdf/00vp.pdf.
13. T. Menzies, D. Port, Z. Chen, J. Hihn, and S. Stukes. Specialization and extrapolation of induced domain models: Case studies in software effort estimation. 2005. IEEE ASE, 2005, Available from http://menzies.us/pdf/05learncost.pdf.
14. 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.
15. Tim Menzies, Zhihao Chen, Jairus Hihn, and Karen Lum. Selecting best practices for effort estimation. *IEEE Transactions on Software Engineering*, November 2006. Available from http://menzies.us/pdf/06coseekmo.pdf.
16. Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, January 2007. Available from http://menzies.us/pdf/06learnPredict.pdf.
17. I. Mierswa, M. Wurst, and R. Klinkenberg. Yale: Rapid prototyping for complex data mining tasks. In *KDD'06*, 1996.
18. A.S. Orrego. Sawtooth: Learning from huge amounts of data, 2004.
19. Chet Ramey. Bash, the bourne-again shell. 1994. Available from http://tiswww.case.edu/php/chet/bash/rose94.pdf.
20. 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.pdf.
21. 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.
22. 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.

Installing 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.

Built-in OURMINE Functions

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 construction 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 statistical 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 Naive 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 specific 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*