How to Build Repeatable Experiments

Gregory Gay, Tim Menzies, Bojan Cukic
CS& EE, WVU, Morgantown, WVU
gregoryg@csee.wvu.edu,
tim@menzies.us, cukic@csee.wvu.edu

Burak Turhan
NRC Institute for Information Technology
Ottawa, Canada
Burak.Turhan@nrc-cnrc.gc.ca

ABSTRACT

The mantra of the PROMISE series is "repeatable, improvable, maybe refutable" software engineering experiments. This community has successfully created a library of reusable software engineering data sets. The next challenge in the PROMISE community will be to not only share data, but to share experiments. Our experience with existing data mining environments is that these tools are not suitable for publishing or sharing repeatable experiments.

OURMINE is an environment for the development of data mining experiments. OURMINE offers a succinct notation for describing experiments. Adding new tools to OURMINE, in a variety of languages, is a rapid and simple process. This makes it a useful research tool. Complicated graphical interfaces have been eschewed for simple command-line prompts. This simplifies the learning curve for data mining novices. The simplicity also encourages large scale modification and experimentation with the code.

In this paper, we show the OURMINE code required to reproduce a recent experiment checking how defect predictors learned from one site apply to another. This is an important result for the PROMISE community since it shows that our shared repository is not just a useful academic resource. Rather, it is a valuable resource industry: companies that lack the local data required to build those predictors can use PROMISE data to build defect predictors.

Categories and Subject Descriptors

i.5 [**learning**]: machine learning; d.2.8 [**software engineering**]: product metrics

Keywords

algorithms,experimentation, measurement

1. INTRODUCTION

With the recent rise of powerful open-source development platforms also came the development of data mining environments that offered seamless connection between a wide range of tools. The most famous of these tools is the WEKA[1] [11] from the Waikato University's Computer Science Machine Learning Group (see Figure 1). WEKA contains hundreds of data miners, has been widely used for teaching and experimentation, and won the the 2005 SIGKDD Data Mining and Knowledge Discovery Service Award. Other noteworthy tools are "R"[2], ORANGE[3] (see Figure 2) and MATLAB [4] (since our interest is in the ready availability of tools, the rest of this paper will ignore proprietary tools such as MATLAB).

Our complaint with these tools is the same issue raised by Ritthoff et al. [9]. Real-world data mining experiments (or applications) are more complex that 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. Ritthoff et al. argue (and we agree) that the standard interface of (say) WEKA does not support the rapid generation of these intricate combinations.

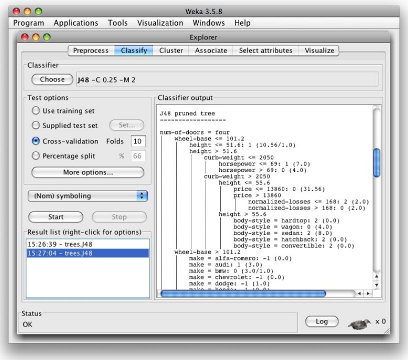
The need to "wire up" data miners within a multitude of other tools has been addressed in many ways. In WEKA's visual *knowledge flow* programming environment, for example, nodes represent different pre-processors/ data miners/ etc, and arcs represent data flows between them. A similar visual environment is offered by many other tools including ORANGE (see Figure 2). The YALE tool (now RAPID-I) built by Ritthoff et al. formalize these visual environments by generating them from XML schema describing *operator trees* like Figure 3.

In our experience in CS, students find these visual environments either discouraging or distracting:

- Some are discouraged by the tool complexity. These students shy away from extensive modification and experimentation.
- Others are so enamored with the impressive software engineering inside these tools that they waste an entire semester building environments to support data mining, but never get around to the data mining itself.

A similar experience is reported by our colleagues in the

[1]`http://www.cs.waikato.ac.nz/ml/weka/`
[2]`http://www.r-project.org/`
[3]`http://magix.fri.uni-lj.si/orange/`
[4]`http://www.mathworks.com`



Weka 3.5.8
Program Applications Tools Visualization Windows Help
Explorer
Preprocess Classify Cluster Associate Select attributes Visualize
Choose
Start
Stop
More options...
Log

Classifier: J48 -C 0.25 -M 2

Test options:
- Use training set
- Supplied test set
- Cross-validation Folds 10
- Percentage split % 66

(Nom) symboling

Result list (right-click for options):
15:26:39 - trees.J48
15:27:04 - trees.J48

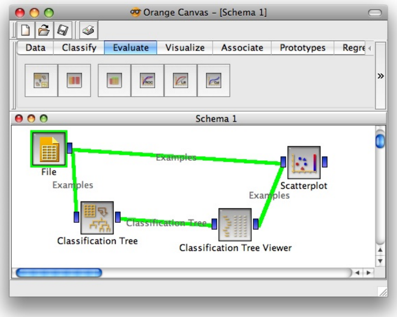
Status: OK

Classifier output:

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

num-of-doors = four
|   wheel-base <= 101.2
|   |   height <= 51.6: 1 (10.56/1.0)
|   |   height > 51.6
|   |   |   curb-weight <= 2050
|   |   |   |   horsepower <= 69: 1 (7.0)
|   |   |   |   horsepower > 69: 0 (4.0)
|   |   |   curb-weight > 2050
|   |   |   |   height <= 55.6
|   |   |   |   |   price <= 13860: 0 (31.56)
|   |   |   |   |   price > 13860
|   |   |   |   |   |   normalized-losses <= 168: 2 (2.0)
|   |   |   |   |   |   normalized-losses > 168: 0 (2.0)
|   |   |   |   height > 55.6
|   |   |   |   |   body-style = hardtop: 2 (0.0)
|   |   |   |   |   body-style = wagon: 0 (4.0)
|   |   |   |   |   body-style = sedan: 2 (8.0)
|   |   |   |   |   body-style = hatchback: 2 (0.0)
|   |   |   |   |   body-style = convertible: 2 (0.0)
|   wheel-base > 101.2
|   |   make = alfa-romero: -1 (0.0)
|   |   make = audi: 1 (3.0)
|   |   make = bmw: 0 (3.0/1.0)
|   |   make = chevrolet: -1 (0.0)
|   |   make = dodge: -1 (1.0)
```

Figure 1: WEKA.



Orange Canvas - [Schema 1]
Data Classify Evaluate Visualize Associate Prototypes Regre
Schema 1
File
Examples
Examples
Scatterplot
Examples
Classification Tree
Classification Tree
Classification Tree Viewer

Figure 2: Orange.

WVU statistics department. In order to support the learning process, they bury these data mining tools inside high-level scripting environments. Their scripting environments shields the user from the complexities of "R" by defining some high-level LISP code that, internally, calls "R."

Our experience with existing data mining toolkits is that these tools are not suitable for publishing or sharing repeatable experiments. This paper discusses why we believe this to be so and offers OURMINE as an alternate format for sharing executable experiments.

OURMINE is a data mining environment used at West Virginia University to teach data mining as well as to support the generation of our research papers. OURMINE does not replace WEKA, "R", etc. Rather, it takes a UNIX shell scripting approach that allows for the "duct taping" together of various tools. For example, OURMINE currently calls WEKA learners as sub-routines, but it is not limited to that tool. This is no requirement to code in a single language (e.g. the JAVA used in WEKA) so data miners written in "C" can be run along side those written in JAVA or any other language. Any program with a command-line API can be combined with other programs within OURMINE. In this way, we can lever useful components from WEKA, R, etc, while quickly adding in scripts to handle any missing functionality.

Based on three years experience with OURMINE, we assert that the tool has certain pedagogical advantages. The simplicity of the tool encourages experimentation and modification, even by data mining novices. We also argue that OURMINE is a productive data mining research environment. For example $\frac{9}{10}$ of the first and second authors' recent papers used OURMINE.

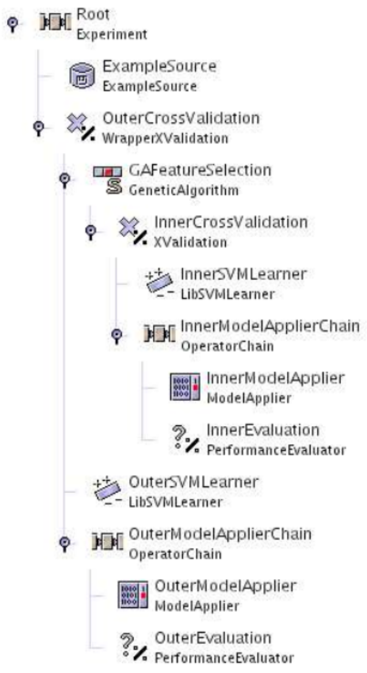


Figure 3: Data mining operator trees. From [7].

More importantly, OURMINE offers a succinct executable representation of an experiment that can be included in a research paper. To demonstrate that, we present:

- The OURMINE script reproducing a recent SE experiment [10].
- The results obtained from that script.

The experiment selected is of particular importance to the PROMISE community since it shows that PROMISE data sets collected from one source can be applied elsewhere to create good defect predictors. This in turn means that the PROMISE repository is not just a useful academic resource, but is also a valuable source of data for industry. If a company wants to build defect predictors, but lacks the local data required to build those predictors, then by using the right *relevancy filters* (described below) it is possible to use data imported from the PROMISE repository to generate defect predictors nearly as good as those that might be built from local data.

2. OURMINE

While a powerful language, we find that LISP can be arcane to many audiences. Hence, OURMINE's script are a combination of BASH [8] and GAWK [1]. Our choice of BASH/GAWK over, say, LISP is partially a matter of taste but we defend that selection as follows. Once a student learns RAPID-I's XML configuration tricks, then those learned skills are highly specific to that particular tool. On the other hand, once a student learns BASH/GAWK meth-

```
#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)
          Attributes[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]+Attributes[NF])) }
     if ( temp >= like ) { # we've found a better hypothesis
        like = temp
        what=h}
   }
   return what;
}
```

Figure 4: A Naive Bayes classier for data sets in csv format where the last column is the class.

ods for data pre-processing and reporting, they can apply those scripting tricks to any number of future applications.

Another reason to prefer scripting 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 4 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 datasets such as those seen in the 2001 KDD cup. WEKA, on the other hand, cannot process these large data sets as it always loads its data into RAM. Figure 4, on the other hand, only requires enough memory to store one instance as well as the frequency counts in the hash table "F".

More importantly, in terms of teaching, Figure 4 is easily customizable. For example, Figure 5 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 fewer than 10 lines in Figure 4. The ease 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.

2.1 Built-in Data/Functions

The appendix of this paper describes the download and install instructions for OURMINE. The standard install comes with the following public domain data sets (found in the $OURMINE/lib/arffs directory):

- PROMISE: cm1, kc1, kc2, kc3, mc1, mc2, mw1, pc1, pc2, pc3, pc4, pc5, ar3, ar4, ar5

1. Modify Figure 4 so that there is no train/test data. Instead, make it an incremental learner. 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 4 into HYPERPIPES [2]. 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 $contains1$:

```
function contains1(h,i,val,numerip) {
  return numericp ?
        Max[h,i] >= val && Min[h,i] <= val :
        (h,i,value) in Seen }
```

3. Use Figure 4 for anomaly detector. Hint: 1) make all training examples classify as 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 5: Four Introductory OURMINE programming exercises.

- UCI (discrete): 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
- UCI (numeric): 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

OURMINE also comes with a library of common functions used in machine learning experiments. All learners are the WEKA implementations. The functions that interface OURMINE with the WEKA set certain command-line parameters. To see those flag settings, type *show <learner name>* at the command prompt. That library includes:

- The data manipulation functions of Figure 6;
- The statistical functions of Figure 7;
- Some reporting functions such as code to auto-generate the latex files required to report quartile charts. For examples of those charts, see Figure 12 and Figure 13 (later in this article).
- Learners: oner, jRip, jRip10, part, aode, aode10, nb (naive bayes), nb10, nbk, lwl, j48, j4810, j4810c, lsr, m5p, 1Bkx, 1Bk, apriori

2.2 Extending the Functionality

OURMINE's library contains the functions used by a team of 12 graduate students and one faculty student researching data mining over a three year period. This library is hardly complete and one of the goals of this paper is to create a community of OURMINE programmers so that the library

buildIncrementalSet builds a set containing a set number of instances;

classes find all values for a class attribute;

combineFilesRandom combines two data sets, randomizing the lines;

combineFiles combines two data sets with common attributes

gains runs InfoGain on the data

intersectAttributes finds the intersection of attributes over several data sets;

logNumbers logs all numeric values in a data set ;

randomizeFiles randomizes lines in a data set;

rankViaInfoGain ranks attributes by InfoGain values;

removeAttributes performs column pruning;

sample creates an over or under-sampled dataset;

shared find shared attributes;

someArff splits a data set into $train.arff$ and $test$.arff where the test fail contains the B-th division of $\frac{1}{B}$-the data and train file contains the rest.

some generates an ARFF file containing certain attributes;

Figure 6: OURMINE data manipulation functions.

of functions can be extended.

Adding new functionality to OURMINE is not difficult. OURMINE's library of scripts are contained in a file called *minerc* which is just a collection of functions written in the BASH scripting language. Hence, to add functionality:

- Create a separate BASH file (i.e. *functions.sh*).
- Write all new functions in this file.
- At the end of *minerc*, add the line "*. functions.sh*".

Very little knowledge of BASH is required to add functions to OURMINE. If the new function is actually in a JAVA file, the BASH function could just instantiate the Java file. In Figure 8, for example, the BASH function *nb* uses a series of commands to instantiate a JAVA classifier. The *blab* is a built-in function that prints text to the screen. The second line is the command to launch an external Java application (-Xmx1024M is a flag that gives the Java application a certain

abcd: finds a,b,c,d values and reports pd, pf, accuracy, precision, and balance. If $\{A, B, C, D\}$ are the true negatives, false negatives, false positives, and true positives (respectively) found by a defect predictor, then:

$$\begin{aligned} pd = recall = & \quad D/(B+D) \\ pf = & \quad C/(A+C) \\ balance = & \quad 1 - \frac{\sqrt{(0-pf)^2+(1-pd)^2}}{\sqrt{2}} \end{aligned}$$

columnStats: find column min, max, mean, and std.deviations.

medians: finds medians;

quartiles: generates box plots showing quartiles;

uniques: finds the unique entries in a column;

winLossTie: runs the Mann-Whitney test

Figure 7: OURMINE statistical functions.

```
nb() {
        blab "n"
        java -Xmx1024M -cp $Tmp/weka.jar \
                 weka.classifiers.bayes.NaiveBayes \
                -p 0 -t  $1 -T $2
}
```

Figure 8: OURMINE calling the WEKA

```
 1 someData() {
 2     cat $1 | gawk ’
 3     BEGIN           { IGNORECASE =1
 4                       srand(’$RANDOM’)
 5                       N=’$2’ }
 6     gsub(/%.*/,"")
 7     /^[ \t]*$/      { next }
 8     In              { Data[rand()]=$0 }
 9     /@data/         { In=1 }
10     /@/             { next }
11     END             { for(D in Data) {
12                          print Data[D]
13                          if ((--N) <= 0) exit }} ’
14 }
```

Figure 9: SomeData: An OURMINE function

amount of memory). The process to call external programs is similar for any language.

The majority of the functions in OURMINE are BASH scripts that glue together GAWK code. GAWK is simple to learn and encourages short, easily modifiable, programs. R. Loui, an associate professor in Computer Science at Washington University in St. Louis, strongly supports the use of GAWK in his artificial intelligence classes. 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.* [5]

For example, the GAWK script of Figure 9 extracts N randomly selected data lines from an arff file. It uses two parameters as input: *$1=input file*; *$2=N* (how many lines of data we require).

- Lines 4 and 5 initializes random numbers and N.
- Lines 6 and 7 are the standard idiom for "skip lines that are all blanks or comments". If, after deleting all characters after comment, the line contains only white space then we skip that line.
- Line 8 stores a data line at a random location in the $Data$ array, but only if Line 9 ran before and recognized the start of the data section in this file (marked with "@data").
- Line 10 skips all the meta lines in the file.
- Lines 10 through 13 print out N random data items.

2.3 Writing Experiments

The OURMINE user writes experiments in a BASH notation into a file loaded by *minerc*. Running an experiment is as simple as typing the name of its function at the OURMINE command prompt (i.e. if your function is called *exp1*, it will run when you type *exp1* at the command prompt).

```
 1 demo15() {
 2  cd $Tmp
 3  (echo "#data,bin, a,b,c,d,acc,pd,pf,prec,bal"
 4   seed=$RANDOM;
 5   for((bin=1;bin<=10;bin++)); do
 6      blab "$bin"
 7      auto93discreteClass  |
 8              someArff --seed $seed --bins $Bins --bin $bin
 9      nb train.arff test.arff |
10      gotwant                |
11      abcd --goal "_20"  --prefix "auto93,$bin" --decimals 1
12   done |
13   sort -t, -n -k 11,11
14   ) |
15   malign > demo15.csv
16   blabln " "
17   echo ""; cat demo15.csv
18   cp demo15.csv $Safe/demo15.csv
19   cd $Here
20 }
```

Figure 10: An OURMINE experiment.

Figure 10 shows a basic experiment from OURMINE. When reading BASH scripts like Figure 10, one thing to note is that the pipe character "|" links the output of one process into the input of another. Also, a "|" at the end of line means that the next line inputs the piped output of the last line.

Line 5 of Figure 10 shows that this a 10-way cross-validation experiment, executed over one file (see line 7: auto93 from the UCI collection). At line 8, the *someArff* function splits the data into "train.arff" (containing 90% of the data) and "test.arff" (containing 10% of the data).

At line 9, a Naive Bayes learner of Figure 8 is called. A more general construct, seen in the experiment described below is to loop through, a set of learners using

for learner in $learners; do

in which case, line 9 would become

$learner train.arff test.arff

The code in lines 10 and 11 collect the performance statistics and print out $a, b, c, d, acc, pred, pf, prec, bal$. Some of these statistics are class-dependent. Line 11 shows the idiom *abcd –goal "_20"* which instructs OURMINE to only report statistics for the class "_20". If statistics for all classes are required, then the OURMINE idiom is to cache the result of the learner, then loop through all classes calling *abcd* on each class in turn. When using the OURMINE tools, that idiom can be see in lines 20 through 29 of Figure 11.

The sort function of line 13 of Figure 8 sorts these results on accuracy (column 11) and the results are saved to a safety box $\$Safe$. This final saving step is required because, by default, OURMINE cleans up after itself as it exits. Unless result files are saved to some safe place, they will be wiped.

This is a simple experiment, and it should be fairly easy to see the flow between statements. If the settings used for the parameters of a function are confusing, students learning OURMINE can run that function separately at the BASH prompt or look at its source code using the *show <function name>* command within OURMINE.

One nuance, not readily apparent in Figure 10, is that these scripts are naturally parallizable. The idiom

ssh me@someMachine.edu runThis

logs into a machine and runs the script *runThis*. The script executes in some default, rather restricted, environment so *runThis* must now how to create the paths and temp directories needed for execution. OURMINE is designed to automatically create its execution environment so it is easy to write *runThis* scripts based on OURMINE. In a standard university environment, where students have the same password on multiple machines, then very large experiments can be farmed out over those CPUs by running:

ssh me@someMachine1.edu runThis1
ssh me@someMachine2.edu runThis2
ssh me@someMachine3.edu runThis3
etc

This lets researchers complete lengthy data mining experiments in minutes to hours, rather than days to weeks.

3. OURMINE & RESEARCH

To illustrate the use of OURMINE, the rest of this paper uses the tool to reproduce a recent *Empirical Software Engineering* paper written by the second and fourth authors [10]. Note that the following code was written by the first author with minimal assistance from the others: for the most part, the first author worked straight from the text of [10].

In their paper, Turhan et al. [10] focus on binary defect prediction (defective or non-defective) and perform three experiments to reach a conclusion in favor of either:

- Cross-company (CC) data imported from other sites;
- or within-company (WC) data collected locally.

That paper made several conclusions:

- **Turhan#1:** Using local WC data produces significantly better defect predictors than using imported CC data.
- **Turhan#2:** However, when *relevancy filtering* (see below) is applied to the CC data, then the imported CC data leads to defect predictors nearly as effective as using local WC data.
- **Turhan#3:** Hence, while local data is the preferred option, it is feasible to use imported data provided that it is selected by a relevancy filter.

This experiment is of great importance to the PROMISE community. It shows that PROMISE data sets collected from one source can be applied elsewhere to learn good defect predictors; i.e. the PROMISE repository is not just a useful academic resource, but is also a valuable source of data for industry. Companies wishing to build defect predictors can do so, even if they lack the local data required to build those predictors.

Such an important result needs confirmation. Tools like OURMINE are useful for such replication studies.

3.1 Experiment

Many research papers in the field of data mining present results but bury the exact details of the implementation. After years of experience, it is clear that the majority of time is spent in the tedium of pre and post-processing. The main contribution of Ourmine is to provide functions to handle these tasks. The script shown in Figure 11 is a complete structured experiment that is run in the Ourmine environment. What is not shown are the highly customized tools that are frequently built and thrown away to deal with tasks like splitting a data set into training and test subsets. An Ourmine function, *someArff* does this for the researcher. Many of the lines in this figure are actually references to internal Ourmine functions.

As a preliminary to this experiment, we took seven PROMISE defect data sets (PC1,CM1,KC1,KC2,KC3,MW1,MC2) and built seven combined data sets, each containing $\frac{6}{7}$-th of the data. For example, the file:

$OURMINE/lib/arffs/mdp/combined_PC1.arff

contains all seven data sets *except* PC1. This is the training set used for the CC (cross-company) data.

```
 1 an2() {
 2    local me="promise_an2"
 3    local bins=10
 4    local repeats=10
 5    local learners="nb lwl"
 6    local datas="PC1 CM1 KC1 KC2 KC3 MW1 MC2"
 7    cd $Tmp
 8    (echo "#data,repeat,bin,treatment,learner,goal,a,b,c,d \
 9       ,acc,pd,pf,prec,bal"
10    for((r=1;r<=$repeats;r++)); do
11       for data in $datas; do
12          arff=$OURMINE/lib/arffs/mdp/shared_$data.arff
13          combined=$OURMINE/lib/arffs/mdp/combined_$data.arff
14          blab "data=$data repeat=$r "
15          seed=$RANDOM;
16
17          for((bin=1;bin<=$bins;bin++)); do
18             blab "$bin"
19             cat $arff |
20             someArff --seed $seed --bins $bins --bin $bin
21             goals=‘cat $arff | classes --brief‘
22             for learner in $learners; do
23                $learner train.arff test.arff |
24                gotwant > results.dat
25                for goal in $goals; do
26                   cat results.dat |
27                   abcd --goal "$goal" \
28                     -prefix "$data,$r,$bin,WC,$learner,$goal" \
29                     --decimals 1
30                done
31             done
32             goals=‘cat $arff | classes --brief‘
33             blab "CC"
34             local N=$[ ( $RANDOM % 10 ) +1 ]
35             blab "($N)"
36             cat $arff |
37             someArff --seed $seed --bins $bins --bin $N
38             for learner in $learners; do
39                makeTrainCombined $combined > trainCom.arff
40                $learner trainCom.arff test.arff |
41                gotwant > results.dat
42                for goal in $goals; do
43                   cat results.dat |
44                   abcd --goal "$goal" \
45                     --prefix "$data,$r,CC,CC,$learner,$goal" \
46                     --decimals 1
47                 done
48              done
49
50           done
51           blabln
52        done
53     done
54    ) | sort -t, -n -k 12,12 | malign > $me.csv #each bin
55    blabln " "
56    echo ""; cat $me.csv
57    cp $me.csv $Safe/$me.csv
58    cd $Here
59 }
```

Figure 11: WC or CC defect prediction. From [10].

Next, we conducted a 10*10-way cross validation experiment; i.e. 10 times we asked *someArff* to randomly sort each data set, then generate 10 train and test sets containing 90% and 10% of each data set. The 90% sets are the training set of the WC (within-company) data. Lines 17 to 31 of Figure 11 shows that standard cross-validation study.

The CC study is shown on lines 32 to 46 of Figure 11. This is the same as the WC study but this time the training set is one the of the *combined_X.arff* files described above.

The learners in this study were two Naive Bayes classifiers. *Nb* (from Figure 8) and Frank et al.'s locally weighted Bayes classifier called *lwl* [3]. Naive Bayes classifiers were used since:

- Previously [6], we have offered evidence that they do better than other commonly used learners;
- Lessmann et al. report that many other learners do no better than Naive Bayes on the PROMISE defect data sets [4].

Lwl applies *relevancy filtering* to the training set. The idea behind relevancy filtering is to collect similar instances together in order to construct a learning set that is homogeneous with the testing set. Specifically, the aim of this filter is to try to introduce a bias in the model by using training data that shares similar characteristics with the testing data. Turhan et al. implemented this with a combination of k-Nearest Neighbor (k-NN) method and Naive Bayes. For each test instance in the test set, the k=10 nearest neighbors in the training set are collected. Duplicates are removed, and the remaining examples are used to train a Naive Bayes Classifier.

For reasons of repeatability, this study did not use Turhan et al.'s relevancy filter. Rather, we used the *lwl* from the WEKA. For each test instance, *lwl* constructs a new naive Bayes model using a weighted subset of training instances in the locale of the test instance. In this approach, the size of the training subset is based on the distance of the kth nearest-neighbor to the instance being classified. The training instances in this neighborhood are weighted, with more weight assigned to the closest instances. An instance identical to the test instance is given a weight of one, with the weight decreasing linearly to zero as the distance increases. Higher values for the number of neighbors (k) will result in models that are less sensitive to data anomalies, while smaller models will conform more closely to the data. The authors caution against using too small of a value for k, as it will be sensitive to the noise in the data. At the suggestion of Frank et al. we ran Figure 11 using $k \in \{10, 25, 50, 100\}$.

3.2 Results

Our results, shown in Figure 12 and Figure 13, are divided into reports of probability of detection (pd) and probability of false alarms (pf). When a method uses k nearest neighbors, the k value is shown (in brackets). For example, the first line of Figure 12 reports WC data when lwl used 100 nearest neighbors.

The results are sorted, top to bottom, from best to worse (higher pd is better; lower pf is better). Quartile plots are shown on the right-hand side of each row. The black dot in those plots shows the median value and the two "arms" on either side of the median show the second and third quartile respectively. The three vertical bars on each quartile chart mark the (0%,50%,100%) points.

Column one of each row shows the results of a Mann-Whitney test (95% confidence): learners are ranked by how many times they lose compared to every other learner (so the top-ranked learner loses the least). In Figure 12 and Figure 13, row i has a different rank to row $i + 1$ if Mann-Whitney reports a statistically significant difference between them.

Several aspects of these results are noteworthy:

- Measured in terms of *pd* and pf, the learners are ranked the same. That is, methods that result in higher *pd* values also generate lower pf values. This result means that, in this experiment, we can make clear recommendations about the value of different learners.
- When performing relevancy filtering on CC data, extreme locality is not recommended. Observe how $k \in \{10, 25\}$ are ranked second and third worst in terms of both *pd* and pf.
- When using imported CC data, some degree of relevancy filtering is essential. The last line of our results shows the worst pd, pf results and on that line we see that Naive Bayes, using all the imported data, performs worst.
- The locally-weighted scheme used by *lwl* improved WC results as well as CC results. This implies that a certain level of noise still exists within local data sets and it is useful to remove extraneous factors from the local data.

These results confirm the Turhan et al. results:

		pd percentiles			2nd quartile median, 3rd quartile
Rank	Treatment	25%	50%	75%	
1	WC lwl(100)	20	76	97	
1	WC lwl(50)	25	75	95	
2	CC lwl(100)	20	73	93	
3	WC lwl(25)	33	73	93	
3	WC lwl(10)	33	73	92	
4	CC lwl(50)	25	70	91	
5	WC nb	57	68	83	
6	CC lwl(10)	25	67	89	
7	CC lwl(25)	25	67	88	
8	CC nb	47	65	88	

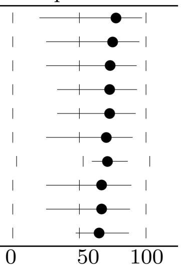


Figure 12: Probability of Detection (PD) results, sorted by median values.

		pf percentiles			2nd quartile median, 3rd quartile
Rank	Treatment	25%	50%	75%	
1	WC lwl(100)	3	22	80	
1	WC lwl(50)	5	22	75	
2	CC lwl(100)	7	26	80	
3	WC lwl(25)	7	25	67	
4	WC lwl(10)	7	27	67	
5	CC lwl(50)	9	29	75	
6	WC nb	18	32	43	
7	CC lwl(10)	11	33	74	
8	CC lwl(25)	11	33	75	
9	CC nb	11	35	53	

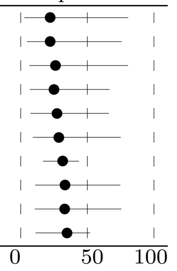


Figure 13: Probability of False Alarm (PF) result, sorted by median values.

- In a result consistent with **Turhan#1**, best results were seen using WC data (see the top two rows of each table of results).
- In a result consistent with **Turhan#2**, after relevancy filtering with $k = 100$, the CC results are nearly as good as the WC result: a loss of only 3% in the median *pd* and a loss of 4% in the median pf.

Our conclusions from this study are the same as **Turhan#3**: while local data is the preferred option, it is feasible to use imported data provided it is selected by a relevancy filter. If a company has a large collection of local development data, they should use that to develop defect predictors. If no such local repository exists, then a cross-company data collection filtered by a locally-weighted classifier will yield usable results. Repositories like PROMISE can be used to obtain that CC data.

4. DISCUSSION

As Figure 14 illustrates, the replication for this experiment was completed within a matter of hours. Original research would undoubtedly take more time, but the tools in OURMINE clearly cut the necessary time to set up an experiment, or to reproduce an experiment, by a large amount.

Note that the slowest task in reproducing the experiment was the 200 minutes for *experiment alteration*; i.e. finding and fixing conceptual bugs in version i of the functions, then improve them with i+1. In the past, we considered this work wasted time since it seemed that we were just making dumb mistakes and correcting them. Now, we have a different view. It is this process of misunderstanding, then correcting the details of an experiment, that is the real point of the experimentation:

- If repeating an experiment takes too long, we may abandon the exercise and learn nothing from the process.
- If repeating an experiment is too easy (e.g. just editing a RAPID-I operator tree) then repeating the experiment does little to change our views on software engineering and data mining.
- But if repeating the experiment takes the *right* amount of time (not too long, not too short), and it challenges just enough of our understanding of some topic, then we find that we "engage" with the problem; i,e, we think hard about:
  - The base assumptions of the experiment;
  - The interesting properties of the data;
  - and the alternative methods that could achieve the same or better results.

That is, we advocate OURMINE and the ugly syntax of Figure 11 over beautiful visual programming environments like WEKA's knowledge flow or the RAPID-I operator tree

Step	Time spent (minutes)
Prep work	38
Data manipulation	152
Coding	163
Experiment alteration (bugs, rework)	210
Total	563 ≈ 10 hours

Figure 14: Time spent on each step

because the visual environments are too easy to use (so we are not forced into learning new ideas).

On the other hand, we still want a productive development environment for developing our experiments. It should be possible to update a data mining environment and get new insights from the tools, fast enough to keep ups motivated and excited to work on the task. In our experience, the high-level scripting of BASH and GAWK lets us find that middle ground between too awkward and too easy.

5. CONCLUSIONS

The mantra of the PROMISE series is "repeatable, improvable, maybe refutable" software engineering experiments. This community has successfully created a library of reusable software engineering data sets that is growing in size and which have been used in multiple PROMISE papers:

- 2006: 23 data sets
- 2007: 40 data sets
- 2008: 67 data sets
- 2009: 85 data sets (at the time of this writing)

The next challenge in the PROMISE community will be to not only share data, but to share experiments; i.e. the PROMISE repository should grow to include not just data, but the code required to run experiments over that data. We look forward to the day when it is routine for PROMISE submissions to come not just with supporting data but also with a full executable version of the experimental rig used in the paper. This would simplify the goal of letting other researchers repeat, and maybe even improve, the experimental results of others.

In the paper, we have used OURMINE to reproduce an important result:

- It is best to use local data to build local defect predictors;
- However, if imported data is selected using a *relevancy filtering*, then....
- ... imported data can build defect predictors that function nearly as well as those built from local data.

This means that repositories like PROMISE are actually an important source of data for industrial applications.

We prefer OURMINE to other tools. For example, four features of Figure 8 and Figure 11 are worthy of mention:

1. OURMINE is very succinct, a few lines can describe even complex experiments.
2. OURMINE's experimental descriptions are complete. There is nothing hidden in Figure 11; it is not the pseudocode of the our experiments, it is the experiment.
3. OURMINE code is executable and can be executed by other researchers directly.
4. Lastly, the execution environment of OURMINE is readily available. 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 environment can be achieved with little 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 PROMISE 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.

6. REFERENCES

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

[2] 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`.

[3] Eibe Frank, Mark Hall, and Bernhard Pfahringer. Locally weighted naive bayes. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 249–256. Morgan Kaufmann, 2003.

[4] 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.

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

[6] 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`.

[7] I. Mierswa, M. Wurst, and R. Klinkenberg. Yale: Rapid prototyping for complex data mining tasks. In *KDD'06*, 1996.

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

[9] 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`.

[10] 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`.

[11] Ian H. Witten and Eibe Frank. *Data mining. 2nd edition.* Morgan Kaufmann, Los Altos, US, 2005.

APPENDIX

Installing OURMINE

OURMINE is an open source tool licenses under GPL 3.0. It can be downloaded from `http://unbox.org/wisp/trunk/`

`our/INSTALL`.

As OURMINE is a command-line tool, the system requirements are insignificant. However, there are a few things that are necessary before installing OURMINE.

- A Unix-based platform. OURMINE is designed to work in the BASH shell, and will not operate on a Windows system. If no other platform is available, a BASH emulator like Cygwin will need to be installed before using OURMINE. Users running any Linux distribution, BSD, or Mac OS X can run OURMINE natively.
- The Java Runtime Environment. Most computers will already have this installed. The ability to run Java programs is required for the WEKA learners.
- The GAWK Programming Language. Many of the scripts within OURMINE are written using GAWK. Any up-to-date Linux system will already have GAWK installed. Cygwin or Mac OS X users will need to install it themselves.

To install OURMINE, follow these instructions:

- Go to a temporary directory
- wget -q -O INSTALL http://unbox.org/wisp/trunk/our/INSTALL
- bash INSTALL

OURMINE is installed to the $HOME/opt directory by default. To run OURMINE, simply move into that directory and type *bash our minerc*. This will launch the OURMINE shell. OURMINE has several demos included to familiarize you with the built-in functionality. These demos may be run by typing *demoX* into the command prompt, where *X* is a number between 3 and 19. To look at the source code for that demo, type *show demoX*. It is highly recommended that new users run each demo and take a close look at its source code. This *show* command works for any function in OURMINE, not just the demos.