2010; **00**:1–7

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

1. Introduction

OURMINE is a data mining scripting envrionment. 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 (Figure 1) must be written in JAVA.

Our preference for BASH [17]/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.

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,

*Correspondence to: Lane Department of Computer Science and Electrical Engineering, West Virginia Unviersity, Morgantown, WV, USA
*E-mail: anelson8@mix.wvu.edu, tim@menzies.us, gregoryg@csee.wvu.edu
Contract/grant sponsor: Publishing Arts Research Council; contract/grant number: 98–1846389

Received 1 October 2009
Revised ?? December 2009

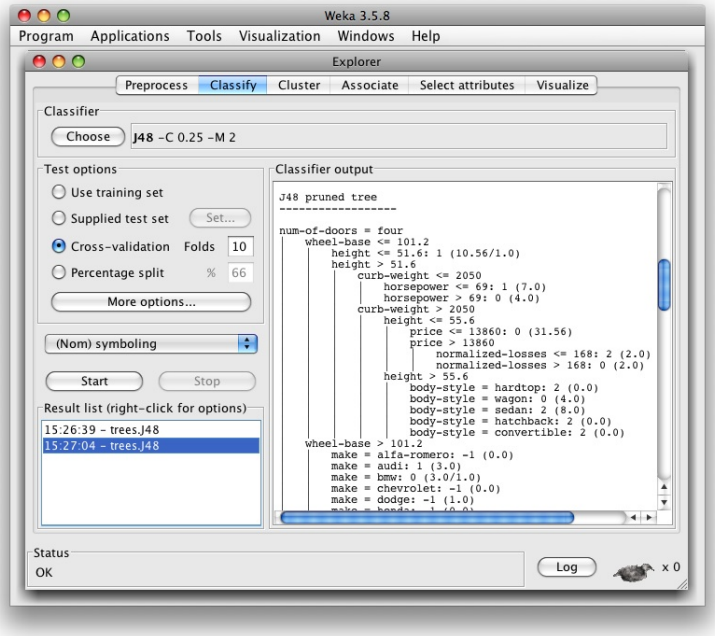

Figure 1. The WEKA toolit running the J48 decision tree learner.

any tool that can be executed from a command prompt can be seemlessly 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 2. 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 workds 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 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 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 repostitory*: 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 sample of these functions, see Figure 5. 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 6 shows a GAWK implementation used by OURMINE to determine the TF-IDF [18] values of each term in a document. This script is simple and concise, while a C++ or Java implementation would be large and overly complex. An additional example demonstrating the brevity of OURMINE

[†] http://mlg.ucd.ie/datasets

abcd provides analysis of experiments, such as *pd,pf,accuracy* and *precision* values;

clean clean text for further processing, removing tokens, capitalizations, stop words, etc.;

docsToSparff constructs a sparse arff file based on a directory of documents;

docsToTfidfSparff generates a sparse arff file of TF-IDF values based on a directory of documents;

funs shows a sorted list of all available functions in OURMINE;

logArff logs all numeric values in a data set ;

malign neatly aligns text into columns;

nb Runs Naive Bayes on the data given;

rankViaInfoGain ranks attributes by InfoGain values;

makeTrainAndTest splits a dataset into a test set and a training set as *train*.arff and *test*.arff, as well as *train*.lisp and *test*.lisp.

Figure 3. A small sample of the available OURMINE functions. Built-in functions give the user something with which to start and begin running demos and experiments immediately. For a detailed list of available tools, please see the appendix.

```
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 4. A GAWK implementation of TF-IDF.

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

script can be seen in Figure 10, 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 datasets such as those seen in the 2001 KDD cup and in [16]. 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

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 for 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 a anomaly detector.

Figure 6. Four Introductory OURMINE programming exercises.

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 afterwords compliment 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 artifical intelligence at Washington University in St. Louis. He writes:

> *There is no issue of user-interface. This forces the programmer to return to the question of what the program does, not how it looks. There is no time spent programming a binsort when the data can be shipped to /bin/sort in no time.* [8]

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

```
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 7. Function help in OURMINE.

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.

3. Using Ourmine for Research

OURMINE is not just a simple demonstration system for novice data miners. It can also be used to generate journal-level publishable results. In the last three years, the authors of this paper have published six papers in leading software engineering journals and conferences using OURMINE [20, 4, 14, 13, 2, 12].

In order to demonstate OURMINE's use in leading edge research, we present here a recent text mining result from an as-yet-unpublished WVU masters thesis.

Matheny [9] 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 leightweight methods perform nearly as well as the rigorous methods.

In a mature scientific discipline, it is standard practice to reproduce important results. The results need to be reproduced since they are important:

- If the results from the experiment are correct, then text mining methods can scale to much larget data sets.

The rest of this section describes the use of OURMINE to reproduce the experiment.

```
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 8. 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.

3.1. The Experiment

As stated above, the purpose of this experiment conducted for this paper is to verify if lightweight data mining methods perform slower or 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.

3.1.1. 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.

3.1.2. The Algorithms

While there are many clustering algorithms used today, this experiment focused on three: a naive K-Means implementation, GenIc [5], and clustering using canopies [10]:

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;

*http://mlg.ucd.ie/datasets
†http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/

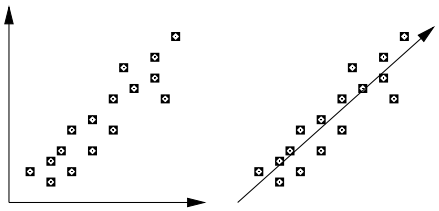

Figure 9. A PCA dimension feature.

(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 intially selecting K centroids at random from all instances in the data. At the beginning of each generation, set the centroid weight to one. When new instances arrive, nudge the nearest centroid to that instance and increase the score for that centroid. In this process, centroids become "fatter" and slow down the rate at which they move toward newer examples. When a generation ends, replace the centroids with less than X percent of the max weight with N more random centroids. Genic repeats for many generations, then returns the highest scoring centroids.
3. Canopy clustering, developed by Google, reduces the need for comparing all items in the data using an expensive distance measure, by first partitioning the data into overlapping subsets called *canopies*. Canopies are first built using a cheap, approximate distance measure. Then, more expensive distance measures are used inside of each canopy to cluster the data.

As to column reduction, 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. PCA involves the calculation of the eigenvalue decomposition of a data covariance matrix or singular value decomposition of a data matrix. Figure 14 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 syntethesized 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:

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

To reduce all terms (and thus, dimensions), we must find the sum of the above

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

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.

3.1.3. 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 computationaly 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 15 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 oppurtunity to run the clusterer across multiple machines simultaneously. As a small example, suppose we wish to run K-means across three different machines with a minimum K of 2 and a maximum K of 256. Since larger values of K generally yield longer runtimes, we may wish to distribute the execution as follows:

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

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

Figure 10. An OURMINE worker function to cluster data using the K-means algorithm. Note that experiments using other clustering methods (such as GenIc and Canopy), could be conducted by calling line 16 above in much the same way, but with varying flags to represent the clusterer.

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

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

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

```
$Reducers
```

is used for the dimensionality reduction of the raw dataset, as seen in Figure 16, whose overall structure is very similar to Figure 15.

3.2. Results

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

```
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 11. An OURMINE worker function to reduce the data using TF-IDF.

3.2.1. 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

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

where D_i and D_j denote two specific documents.

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 17 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. Thus, the conclusions from this experiment shows that fast heuristic methods are sufficient for large data sets due to their scalability.

4. Conclusions

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 an important result. From the experiment, we concluded that:

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

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 code like in Figure 15 and Figure 16 is executable and can be executed by other researchers directly.
3. 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 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.

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. 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.
5. Chetan Gupta and Robert Grossman. Genic: A single pass generalized incremental algorithm for clustering. In *In SIAM Int. Conf. on Data Mining*. SIAM, 2004.
6. 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.
7. 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.
8. R. Loui. Gawk for ai. *Class Lecture*. Available from http://menzies.us/cs591o/?lecture=gawk.
9. A. Matheny. Scaling up text mining, 2009. Masters thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University.
10. 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.
11. T. Menzies. Evaluation issues for visual programming languages, 2002. Available from http://menzies.us/pdf/00vp.pdf.
12. 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.
13. 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.
14. 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.
15. I. Mierswa, M. Wurst, and R. Klinkenberg. Yale: Rapid prototyping for complex data mining tasks. In *KDD'06*, 1996.
16. A.S. Orrego. Sawtooth: Learning from huge amounts of data, 2004.
17. Chet Ramey. Bash, the bourne-again shell. 1994. Available from http://tiswww.case.edu/php/chet/bash/rose94.pdf.
18. 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.
19. 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.
20. 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.

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 12. Similarity values normalized according to the combination of most rigorous reducer and clusterer.

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 constuction of the sets, and $bin is the bin to select as the test set. For instance, if 10 is chosen as the number of bins, and 1 is chosen as the test set bin, then the resulting training set would consist of 90% of the data, and the test set would consist of 10%.
malign	Neatly aligns any comma-separated format into an easily readable format	*malign*
medians	Computes median values given a list of numbers	*medians*
quartile	Generates a quartile chart along with min/max/median values, as well as second and third quartile values given a specific column	*quartile*
show	Prints an entire OURMINE function so that the script can be seen in its entirety	*show $functionName*
winLossTie	Generates win-loss-tie tables given a data set. Win-loss-tie tables, in this case, depict results after a statstical analysis test on treatments. These tests include the Mann-Whitney-U test, as well as the Ranked Wilcoxon test	*winLossTie –input $input.csv –fields $numOfFields –perform $performanceField –key $keyField –$confidence*, where $input.csv refers to the saved output from the *abcd* function described above, $numOfFields represents the number of fields in the input file, $performanceField is the field on which to determine performance, such as *pd*, *pf*, *acc*, $keyField is the field of the key, which could be a learner/treatment, etc., and $confidence is the percentage of confidence when running the test. The default confidence value is 95%

Learners

Function Name	Description	Usage
adtree	Calls WEKA's Alternating Decision Tree	*adtree $train $test*
bnet	Calls WEKA's Bayes Net	*bnet $train $test*
j48	Calls WEKA's J48	*j48 $train $test*
nb	Calls WEKA's 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 specfic fields on which to perform log calculations
stems	Performs Porter's stemming algorithm on incoming text data	*stems $inputFile*
stops	Removes any terms from incoming text data that are in a stored stop list	*stops*
tfidf	Computes TF*IDF values for terms in a document	*tfidf $file*
tokes	Removes unimportant tokens or whitespace from incoming textual data	*tokes*

Feature Subset Selectors

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

Clusterers

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