Open Source Data Mining with OURMINE

September 1, 2009

Adam Nelson, Tim Menzies, Gregory Gay
CSEE, WVU, Morgantown, WV
anelson8@mix.wvu.edu, tim@menzies.us, gregoryg@csee.wvu.edu

Abstract

In this paper we discuss OURMINE as a data mining environment for the development and deployment of experiments, as well as its application in both data mining instruction and acquisition. Finally, we introduce two recent experiments conducted using OURMINE. The first experiment is to determine how defect predictors learned from one site can apply to another using PROMISE data. This is important because it shows that not only can this data be used for academic purposes, but also in industry; companies can use PROMISE data to build their own defect predictors when local data is not available. The second experiment conducted explores benefits of faster clustering and reduction algorithms over more rigorous methods using cluster inter/intra similarities and purity measures.

1 Introduction

Open source environments are abundantly available for data mining. Tools such as "R"[1], ORANGE[2] (see Figure 2), MATLAB [3], and even WEKA [4] [12] from the Waikato University's Computer Science Machine Learning Group, which was given the 2005 SIGKDD Data Mining and Knowledge Discovery Service Award, are among the most popular. While these tools are noteworthy, used by hundreds of data miners for both teaching and experimentation,

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

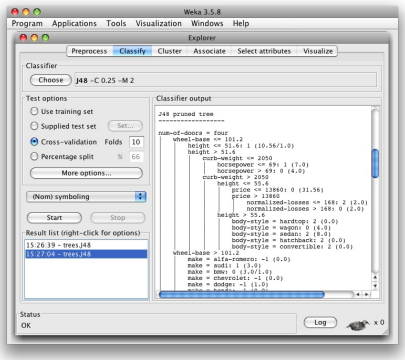


Figure 1: WEKA.

our issue with these tools is the same as raised by Ritthoff et al. [10]. Data mining in the real world is complex, and requires leveraging the combination of a multitude of tools including data pre-processors, data miners and report generators instead of simply running a single algorithm. We agree with Ritthoff et al. that an interface given by a tool such as WEKA does not support fast generation of these required combinations.

The demand for connectivity between data miners in other tools has yielded many results. For instance, WEKA introduced a visual programming environment where nodes denote data pre-processors and miners, etc. These nodes are then connected by arcs, representing data flows between them. Similarly, ORANGE provides a visual representation, as seen in Figure 2.

But while these visual environments are important, they are not always beneficial. In fact, according to Gay et al. [3], data mining students find these visual environments discouraging or distracting due to tool complexity or wasted time in the construction of these environments instead of actually *data mining*.

As we will discuss in this paper, in our experience, OURMINE provides a preferable environment for building and sharing publishable experiments over existing data mining toolkits. Thus, we present two experiments conducted using OURMINE whose results could be inserted into a research paper or thesis.

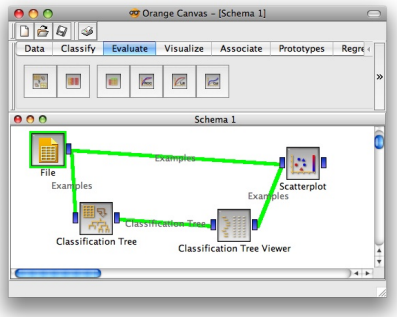


Figure 2: Orange.

2 OURMINE

OURMINE is a toolkit being used at West Virginia University to not only teach data mining through small examples, but also to provide an environment in which to conduct even large, time consuming experiments.

The toolkit operates on UNIX-based operating systems and as such uses shell scripting at its core. As a result, this allows any tool that can be executed from a command prompt to be seemlessly "linked" with other tools. As an example, you see in Figure 3 a simple bash function used in OURMINE to clean text data before conducting any experiments using it. Line 6 shows passing 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. The modules shown are written using BASH, awk and perl. Therefore, OURMINE allows connectivity between tools written in various languages as long as there is always a command-line API available for each tool.

The following sections describe OURMINE's functions and applications.

2.1 Built-in Data and Functions

OURMINE comes with the following data sets to begin conducting experiments right after installation (found in the appendix):

- Text, including STEP datasets (numeric): ap203, ap214, bbc, bbcsport, law, 20 Newsgroup subsets [sb-3-2, sb-8-2, ss-3-2, sl-8-2][5]
- UCI (discrete): anneal, colic, hepatitis, kr-vs-kp, mushroom, sick, waveform-5000, audiology, credit-a, glass, hypothyroid, labor, pcolic, sonar, vehi-

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

```
1 clean(){
2    local docdir=$1
3    local out=$2
4
5   for file in $docdir/*; do
6      cat $file | tokes | caps | stops $Lists/stops.txt > tmp
7      stems tmp >> $out
8      rm tmp
9   done
10 }
```

Figure 3: An OURMINE function to clean text documents and collect the results.

cle, 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
- PROMISE (discrete): 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. Just a few of these functions can be seen in Figure 7.

2.2 Adding/Modifying Code

Adding custom scripts to OURMINE is done by either supplying a new executable BASH file or by only adding to/modifying functions in the already existing code. Either method can be done quickly without a lot of knowledge of either BASH or of how OURMINE's file structure works. For example, if a user wished to add a *myFunctions.sh* to be used in the environment

- Create an executable *myFunctions.sh* containing any custom scripts
- Include this file in *$HOME/opt/ourmine/our/lib/sh/minerc.sh*

To add to or modify code in already existing files, an edit and save is all that is required.

```
#update counters for all words in the record
function train() {
  Documents++;
  for(I=1;I<NF;I++) {
    if( ++In[$I,Documents]==1)
    Document[$I]++
    Word[$I]++
    Words++
  }
}

#compute tfidf for one word
function tfidf(i) {
  return Word[i]/Words*log(Documents/Document[i])
}
```

Figure 4: A GAWK implementation of TF-IDF.

Once this basic structure of OURMINE is learned, however, more advanced additions can be made such as including multiple directories of BASH files, some of whose functions call code written in other files and other languages located elsewhere in the codebase. This gives a nearly limitless opportunity for customization.

2.3 Learning and Teaching with OURMINE

Standard data mining concepts can sometimes appear to be overly complex when implemented using an intricate, highly specific system such as those found in WEKA, etc. For this reason, OURMINE utilizes scripting using BASH [8] and GAWK [1] to better convey the inner-workings of these concepts. For instance, Figure 4 shows a GAWK implementation used by OURMINE to determine the TF-IDF [9] 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. GAWK is used in OURMINE for its simplicity and power. An associate professor of Computer Science at Washington University in St. Louis, R. Loui, encourages the use of GAWK in his artificial intelligence classes, and 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.* [6]

Another reason for OURMINE's scripting approach is due to its applicable generality. If a student learns a highly specific portion of a toolkit, those learned skills are generally confined to that environment. However, skills obtained from learning to use scripting within OURMINE can be used in a variety of future applications.

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, 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 typng *help X*, where *X* is the name of the function, information about that function is printed to the screen. See Figure 6 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.

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 the 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 a calculator, and then immediately afterwords, compliment this by running Naive Bayes on a small dataset 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.

Figure 5 shows a data mining experiment to be used as a demo. Once a student learns the basic mechanics of this demo experiment, its framework can be copied to another function within OURMINE to conduct a custom experiment using any number of tools.

3 Using Ourmine for Research

In order to demonstrate OURMINE as not only a tool used for data mining instruction and acquisition, or on only smaller experiments conducted in a classroom setting, the first author of this paper has reproduced variations of two recent, publishable experiments using OURMINE. The second experiment uses code written entirely by the first author, while the first uses scripts written by all authors of this paper.

The first experiment, based on those conducted by Turhan et al. [11],

```
1 demo004(){
2    demo004worker
3 }

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

6     local learners="nb j48"
7     local data="$Data/discrete/iris.arff"
8     local bins=10
9     local runs=5
10    local out=$Save/demo004-results.csv

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

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

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

20              for learner in $learners; do

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

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

33  less $out
34 }
```

Figure 5: A demo OURMINE experiment. The worker function cycles through specified learners over a data set, and analyzes the results to find *a,b,c,d,accuracy,pd,pf,precision* and *balance* values.

```
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 6: Function help in OURMINE.

abcd provides analysis of experiments, such as *pd,pf,balance* 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 7: OURMINE functions give the user something with which to start and begin running demos and experiments.

as well as Gay et al., focuses on binary defect prediction. In [11], Turhan et al. conducted three experiments to rule in favor of cross-company (CC) data obtained from other sites, or within-company (WC) data gathered locally. The conclusions of those experiments show that CC data, when applied using

relevancy filtering, as explained below, can lead to defect predictors almost as effective as WC data. Thus, as stated by Gay et al., "...while local data is the preferred option, it is feasible to use imported data provided that it is selected by a relevancy filter."

The second experiment was a small-scale reproduction of that conducted by a graduate computer science student at West Virginia University to be used in a Master's thesis. The purpose of the experiment is to show that faster heuristic means of clustering and dimensionality reduction yield results comparable to slower, more rigorous methods when examining these methods' runtimes, cluster similarities, as well as cluster purities.

In the following sections we describe the experiments and show some of the scripts used to conduct them. Finally we analyze the results.

3.1 Experiment I

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. Then, duplicates are removed and the remaining instances are used as the new training set.

Recently in the paper [3], Gay et al. confirmed that using a relevancy filter on CC data is nearly as good as WC data, however in that experiment, a locally weighted scheme was used as the filter via *lwl* [2] instead of k-NN.

3.1.1 Building the Experiment

A sample of the script used to conduct this experiment is shown in Figure 8. The complete script can be located as per directions in the appendix.

To begin, the data in this study were the same as used by Gay et al.; 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, which is used as the training set for the cross-company (CC) data.

Next, as can be seen in line 15 of Figure 8, 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 25-33), the same test set is used, but with the newly created training set. Lines 34-41 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 [11], as well as because Naive Bayes has been shown to perform better on PROMISE defect data sets than other learners [5].

3.1.2 Results

Our results for this experiment can be found in Figure 14 and Figure 15. Figure 14 shows pd (probability of detection) values sorted in decreasing order, while Figure 15 shows pf (probability of false alarm) values sorted in increasing order. 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 their results of a Mann-Whitney test at 95% confidence. A rank is determined by how many times a learner or learner/filter loses compared to another. The method that lost the least number of times is given the highest rank.

The following are important conclusions derived from these results:

- When CC data is used, relevancy filtering is crucial. According to our results, cross-company data with no filtering (Naive Bayes alone) 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 WC 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 $goals; do
                cat produced.dat | abcd --prefix "$dat,$run,$bin,WC,$learner,$goal" \
                 --goal "$goal" \
                  --decimals 1
             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 $goals; do
                cat produced.dat | abcd --prefix "$dat,$run,$bin,WkNN,$learner,$goal" \
                --goal "$goal" \
                --decimals 1
             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 $goals; do
                cat produced.dat | abcd --prefix "$dat,$run,$bin,CC,$learner,$goal" \
                --goal "$goal" \
                --decimals 1
...
```

Figure 8: A sample of the script used in conducting the WC vs. CC experiment.

- 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 de-

velopment data is available. However, if local data is not available, relevancy-filtered cross-company data provides a feasible means to build defect predictors.

3.2 Experiment II

As stated above, the purpose of this experiment conducted for this paper is to verify if heuristic clustering/reduction methods outperform slower, more thorough and rigorous ones when comparing runtimes

The datasets 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][6]

3.2.1 Clustering

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 [7]

Thus the purpose of clustering is to "determine the intrinsic grouping in a set of unlabeled data" [7].

3.2.2 Reduction Methods

Data can sometimes be overwhelmingly large, containing a great number of attributes, or dimensions. In order to reduce this vast multidimensional space, there exist dimensionality reduction methods. Dimensionality reduction filters the attributes, and attempts to select the most valid ones. In "A Survey of Dimension Reduction Techniques" [8] by I.K. Fodor, Fodor writes:

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

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

[8] `https://e-reports-ext.llnl.gov/pdf/240921.pdf`

One of the problems with high-dimensional datasets is that, in many cases, not all the measured variables are "important" for understanding the underlying phenomena of interest.

In other words, in order to reduce the size (and inevitable number of computations) of our data, our goal is to extract the most relevant information and throw out the rest.

3.2.3 The Algorithms

While there are many clustering algorithms used today, this experiment focused on three: a naive K-Means implementation, GenIc [4], and clustering using canopies [7].

K-means, a special case of a class of EM algorithms, works as follows:

1. Select initial K centroids at random;
2. Assign each incoming point to its nearest centroid;
3. Adjusts each cluster's centroid to the mean of each cluster;
4. 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.

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. Repeat for many generations and return the highest scoring centroids.

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.

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. In more mathematical terms, it maps possibly correlated variables into a smaller set of uncorrelated variables, which are called principal components. Figure 9 shows an example of how two dimensions can be approximated in a single new dimension feature, as seen by the dashed line.

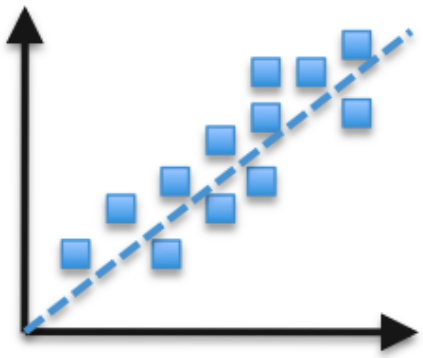


Figure 9: A PCA dimension feature.

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:

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

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

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

3.2.4 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 10 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
```

Lines 9-13 of Figure 10 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 dataset. 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 and purities (as described below), 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 11, whose overall structure is very similar to Figure 10.

```
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.

3.3 Results

To determine the overall benefits of each clustering method, this experiment used both cluster similarities and cluster purities.

3.3.1 Purities

When we cluster data that already has classes available to us, we can determine cluster purity. A cluster is considered relatively "pure" if it contains a high percentage of points that share the same class. Thus, we can say that this is a measure of the "*quality* of a clustering solution"[9].

To determine the purity of individual clusters, the equation

$$purity(C_a) = \frac{1}{|C_a|} * max(|C_a|_{cl=b}) \tag{3}$$

was used, where a cluster C_a contains $|C_a|_{cl=b}$ number of points assigned to

[9] http://www.cse.iitm.ac.in/~cs672/purity.pdf

```
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.

class b. However, a high purity using this equation always is easy to obtain. For instance, if a cluster contains only one instance from the dataset, the purity of that cluster is always 1. Therefore, this study uses a weighted sum of these purities

$$purity = \sum_{a=1}^{k} \frac{|C_a|}{|D|} * purity(C_a) \tag{4}$$

where $|D|$ denotes the number of documents in the dataset.

To determine cluster purities, the following text datasets were used in conjunction with their natural classes:

- BBC: 5 natural classes (business, entertainment, politics, sport, tech)
- BBCSPORT: 5 natural classes (athletics, cricket, football, rugby, tennis)[10]

The results of the cluster purity experiments are shown in Figure 12. By examining these results, we can see that the more complete methods (in this

[10] `http://mlg.ucd.ie/datasets/bbc.html`

case K-means) yields the highest overall weighted purity. However, since each data set and its class distribution can drastically alter the outcome of these values, it is sometimes beneficial to use fast heuristic methods, as seen in the results for the BBC data set.

3.3.2 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.

A cluster's intra-similarity is given by

$$intraSim_a = \sum_{d \epsilon C_a} \sum_{d' \epsilon C_a} \frac{cos(d, d')}{n_a^2} \tag{5}$$

where d is a specific document in cluster C_a, n represents the number of documents in the cluster, and $cos(d, d')$ is the cosine similarity between document d and d'.

A cluster's inter-similarity to other clusters is given by

$$interSim_{ab} = \sum_{d \epsilon C_a} \sum_{d' \epsilon C_b} \frac{cos(d, d')}{n_a n_b} \tag{6}$$

where C_a and C_b represent different clusters, and n_a and n_b denote the number of documents contained in these clusters.

Finally, the overall similarities of a clustering solution are

$$intraSim = \sum_i \frac{n_i}{N} intraSim_a \tag{7}$$

$$interSim = \sum_i \frac{n_i}{N} interSim_{ab} \tag{8}$$

Figure 13 shows the results from the cluster inter/intra similarity tests.

4 Conclusions

In this paper we have discussed OURMINE has an important tool in the development and deployment of data mining experiments, as well as its use in data mining instruction. We have also used OURMINE to reproduce two publishable experiments that yielded noteworthy results. In our first 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*, because...
- Imported data that uses *relevency filtering* performs nearly as well as using local data to build defect predictors

In our second experiment, we learned:

- When examining cluster purities and 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 while the slower solutions are more rigorous, they do not maintain *scalability*.

We prefer OURMINE to other visual tools such as WEKA, etc., because much like Ritthoff et al., we acknowledge the standard interface's ability to represent only a small subset of possible experiments. Complex visual environments can result in the data mining novice to become discouraged or distracted from learning to focus on actually *data mining* and conducting meaningful experiments.

References

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

[2] 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.

[3] Gregory Gay, Tim Menzies, Bojan Cukic, and Burak Turhan. 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.

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

[5] 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.

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

[7] 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.

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

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

[10] 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`.

[11] 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`.

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

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`.

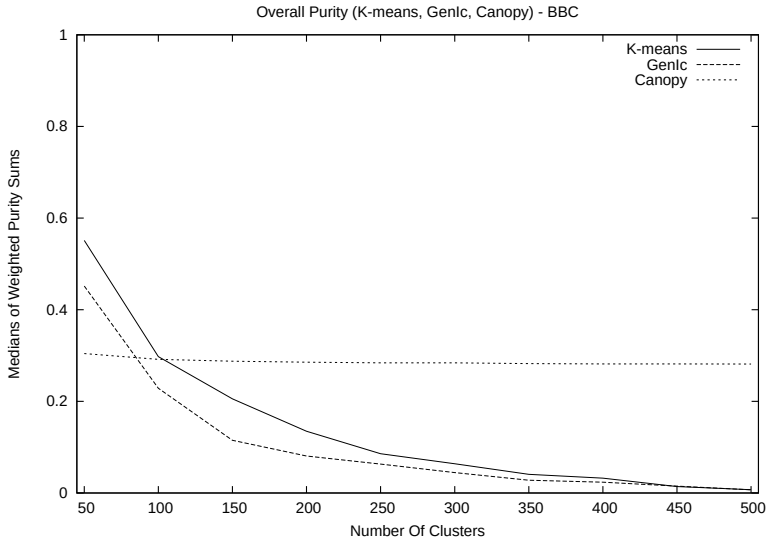
OURMINE is a command-line environment, and as such, system requirements are minimal. However, in order to use OURMINE three things must be in place:

- 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.

Locating OURMINE Code

OURMINE code can be located from the install url given above, or by downloading the toolkit.



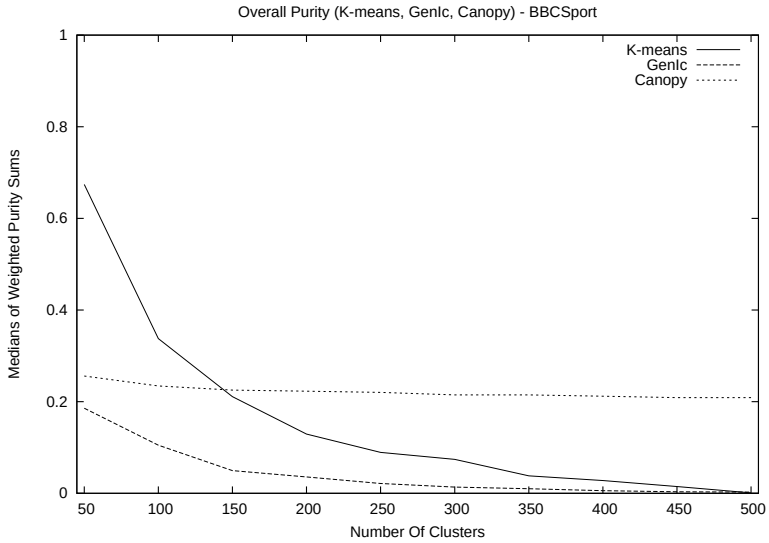


Figure 12: Normalized weighted purity sums of clusters, separated by data set. Note that using data set BBCSport, K-means' clusters clearly contain a higher weighted purity. However, for BBC data, GenIc follows K-means closely, while Canopy remains relatively low, but steady, as a result of documents being shared by neighboring canopies.

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

		pd percentiles			2nd quartile median, 3rd quartile
Rank	Treatment	25%	50%	75%	
1	WC kNN/nb	66	73	80	
2	CC kNN/nb	57	71	83	
2	WC nb	59	69	83	
3	CC nb	49	66	87	
					0 50 100

Figure 14: Probability of Detection (PD) results, sorted by median values.

		pf percentiles			2nd quartile median, 3rd quartile
Rank	Treatment	25%	50%	75%	
1	CC kNN/nb	17	29	43	
1	CC nb	13	34	51	
2	WC nb	17	30	41	
3	WC kNN/nb	20	27	34	
					0 50 100

Figure 15: Probability of False Alarm (PF) results, sorted by median values.