Running Data Mining Experiments

Tim Menzies, *Member, IEEE*

INTRODUCTION

Compare Figure 1 with Figure 2. The former shows the pseudo-code for a data mining experiment. The latter shows how the main driver of such an experiment. Not shown are the numerous sub-scripts used to (e.g.) divide the data into training and test sets; extract column subsets, etc, etc.

This paper explains why something as simple as Figure 1 gets as complicated as Figure 2. In summary, when coding data mining experiments, there are 16 important principles which are missed by Figure 1. Each one is simple enough to apply but all together, it took the author far too long to learn them. With these principles in hand, it is possible to run comprehensive data mining experiments. Without them, much time can be wasted in under-sampling a problem, losing results, running out of disk space, etc etc.

The rest of this article describes those principles.

1. RUN RANDOMIZED CROSS-VALIDATION STUDIES

In a *M*N-way cross-validation study* the available training data is divided into N buckets (often N=10 except for very small data sets where N=3). For each bucket in a 10-way cross-evaluation, a theory is learned on 90% of the data in the $N-1$ buckets then tested on 10% of the data from the remaining bucket.

It is considered good practice to repeat an N-way study M times, randomizing the order each time. Many algorithms exhibit *order effects* where certain orderings dramatically improve or degrade performance (e.g. insertion sort runs slowest if the inputs are already sorted in reverse order). Randomizing the order of the inputs defends against order effects.

In essence, such a M*N-way cross-evaluation study orchestrates multiple experiments where a learned theory is tested against data not seen during training. In all, a 10*10-way study generates 100 training sets $t_1, t_2, .., t_{100}$ and 100 disjoint test sets $t_1, t_2, .., T_{100}$ (i.e. $t_i \cup T_i = \emptyset$). In the data mining literature, this is the preferred evaluation method when the goal is to produce theories intended to predicting future as-yet-unseen events [1].

2. COMPARE APPLES WITH APPLES

(I'm ashamed to say the following issue cost me two weeks of experimentation. Read this section carefully!)

Dr. Menzies is with the Lane Department of Computer Science, West Virginia University and can be reached at tim@timmenzies.net

This research was conducted with funds from the NASA Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

Manuscript received January 20, 2001; revised November 18, 2002.

```
M        = 10
N        = 10
DATAS    = (cm1 kc3 kc4 mw1 pc1 pc2 pc3 pc4)
SUBSETS  = (loc locs m h hH locs+m locs+h locs+hH most)
FILTERS  = (none logNums)
LEARNERS = (oneR J48 Nb)

for data in DATAS
  for filter in FILTERS
      data' = filter(data)
    for attributes in SUBSETS
        some = just the attributes of data'
      repeat M times
          randomized order from "some"
          generate N bins from "some"
        for i in 1 to N
          tests  = bin[i]
          trainingData = some - tests
          for learner in LEARNERS
            METHOD = (filter attributes learner)
            theory = learner(trainingData)
            RESULT[METHOD] = use theory on tests
```

Fig. 1. This study.

Outer loops of an experimental rig generate test and training sets, perhaps after filtering them through various pre-processors. For the experiments to be valid, the learners have to be called on the same training and test sets; i.e. the call to the learner should be buried deep inside the inner-most loop of the experimental rig (see line 49 of Figure 2).

3. COLLECT MULTIPLE PERFORMANCE STATISTICS

A common method of assessing learners is via classification accuracy. This is most strange since classification accuracy can be remarkably uninformative.

A more comprehensive set of statistics can be collected using *ROC curves*. Formally, a *detector* hunts for a *signal*. Signal detection theory [2] offers *receiver operator characteristic* (ROC) curves as an analysis method for assessing different detectors. A typical ROC curve is shown in Figure 3. The y-axis shows probability of detection (pd) and the x-axis shows probability of false alarms (pf). By definition, the ROC curve must pass through the points $pf = pd = 0$ and $pf = pd = 1$ (a detector that never triggers never makes false alarms; a detector that always triggers always generates false alarms). Between these two points, the curve can take three interesting trajectories:

1) A straight line from (0,0) to (1,1) is of little interest since it offers *no information*: i.e. the probability of a detector firing is the same as it being silent.
2) Another trajectory is the *negative curve* that bends away from the ideal point. Our experience has been that these are detectors which, if their tests were negated, would transpose into a *preferred curve*.

```
. $HOME/.bashrc
. $HOME/etc/myweka
Data=${Data=$Arffs/mdp43}
Todo=${Todo="cm1 kc3 kc4 mw1 pc1 pc2 pc3 pc4"}
Learners=${Learners="J48 oneR Nb"}
Repeats=${Repeats=10}
Filters=${Filters="none logNums"}
Inc=${Inc=0} ; Bins=${Bins=10}
Subsets=${Subsets="1 2 3 4 5 6 7 8 9"}
Outputs=${Outputs="$HOME/var/pstats.103.out"}
Inputs=${Inputs="$HOME/var/pstats.103.in"}

setup() {
    rm -rf   $Inputs
    mkdir -p $Inputs $Outputs
    for what in $Todo; do
      blah "\n $what"
      for filter in $Filters; do
        blah "\n\t $filter"
        TrainTest=$Inputs/$what.$filter
        $filter $Data/$what.arff  > $TrainTest
        for subn in $Subsets;do
          sub=`mdp43subsets -v Name=$subn`
          mdp43subsets -v Want=$subn $TrainTest > $TrainTest.$subn.arff
          blah " $subn"
        done
      done
    done
    blah "\n"
}
nways() {
    pstats =prefix "data,filter,sub,learner,inc-nway,goal,training,testing,secs" -l  > $Outputs/00.stats
    for what in $Todo; do
      for filter in $Filters; do
        for subn in $Subsets; do
          TrainTest="$Inputs/$what.$filter.$subn.arff"
          goal=`nthClass $TrainTest 2`
          Nways=`nway =bins $Bins =repeats $Repeats $TrainTest` #:<-- split the data
          (   cd $Nways/test
              for arff in *.arff; do
                for learner in $Learners; do
                  Now="$arff,$filter,$subn,$learner,$Inc,$goal";
                  blah "$Inputs ==> $Now\n"
                  Test=$arff
                  Train="../train/$arff"
                  Size1=`instances $Train`
                  Size2=`instances $Test`
                  B4=`date +"%s"`
                  $learner $Train $Test > $Test.out
                  After=`date +"%s"`
                  let Time=After-B4
                  pstats =goal $goal =prefix "$Now,$Size1,$Size2,$Time" $Test.out >> $Outputs/$what.$filter.$subn.stats
                done
              done
          )
          blah "purging temps...\n";
          rm -rf $Nways
        done
      done
    done
}
save() {
   cd $Outputs
   zip ../pstats.103.`date +"%s"`.zip *
   cd ..
   ls -lsa pstats.103*zip
}
setup
nways
save
```

Fig. 2. Code

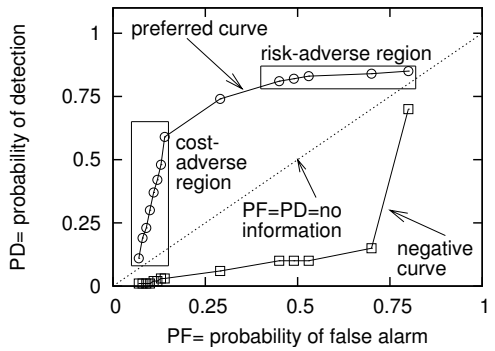

Fig. 3. Regions of a typical ROC curve.

		module found in defect logs?	
		no	yes
signal detected?	no (i.e. $v(g) < 10$)	A = 395	B = 67
	yes (i.e. $v(g) \geq 10$)	C = 19	D = 39

$$pd = \quad Prop.detected = \quad 37\%$$
$$pf = \quad Prob.falseAlarm = \quad 5\%$$
$$notPf = \quad 1 - pf = \quad 95\%$$
$$bal = \quad Balance = \quad 45\%$$
$$Acc = \quad accuracy = \quad 83\%$$

Fig. 4. A ROC sheet assessing a learned theory $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules that fall into each cell of this ROC sheet.

3) The point ($pf = 0$,$pf = 1$) is the ideal position (a.k.a. "sweet spot") on a ROC curve. This is where we recognize all errors and never make mistakes. *Preferred curves* bend up towards this ideal point.

In the ideal case, a detector has a high probability of detecting a genuine signal (pd) and a very low probability of false alarm (pd). This ideal case is very rare. The only way to achieve high probabilities of detection is to trigger the detector more often. This, in turn, incurs the cost of more false alarms.

Pf and pd can be calculated using the ROC sheet of Figure 4. Consider a detector which, when presented with some signal, either triggers or is silent. If some oracle knows whether or not the signal is actually present, then Figure 4 shows four interesting situations. The detector may be silent when the signal is absent (cell A) or present (cell B). Alternatively, if the detector registers a signal, sometimes the signal is actually absent (cell C) and sometimes it is present (cell D).

If the detector registers a signal, there are three cases of interest. In one case, the detector has correctly recognized the signal. This probability of this detection is the ratio of detected signals, true positives, to all signals:

$$\text{probability detection} = pd = recall = \frac{D}{B+D} \tag{1}$$

(Note that pd is also called $recall$.) In another case, the probability of a false alarm is the ratio of detections when no signal was present to all non-signals:

$$\text{probability false alarm} = pf = \frac{C}{A+C} \tag{2}$$

For convenience, we define $notPf$ to be the complement of pf:

$$\text{notPf} = 1 - \frac{C}{A+C} \tag{3}$$

Figure 4 also lets us define the *accuracy*, or acc, of a detector as the percentage of true negatives and true positives:

$$\text{accuracy} = acc = \frac{A+D}{A+B+C+D} \tag{4}$$

All these measures range 0 to 1 and, if reported as percentages, have the range

$$0 \leq \ acc\%, \ pd\%, \ , notPf\% \leq 100$$

Ideally, we seek detectors that maximize *acc%, pd%, notPf%*.

Note that maximizing any one of these does not imply high values for the others. For example Figure 4 shows an example with a high accuracy (83%) but a low probability of detection (37%). Accuracy is a good measure of a learner's performance when the classes occur with similar frequencies.

In practice, engineers *balance* between pf and pd. One way to operationalize this notion of *balance*, we define bal to be the Euclidean distance from the sweet spot $pf = 0, pd = 1$ to a pair of $< pf, pd >$. For convenience, we (a) normalize bal by the maximum possible distance across the ROC square ($\sqrt{2}$); (b) subtract this from 1; and (c) express it as a percentage; i.e.

$$balance = bal = 1 - \frac{\sqrt{\left(\sqrt{0-pf}\right)^2 + \left(\sqrt{1-pd}\right)^2}}{\sqrt{2}} \tag{5}$$

Hence, better and *higher* balances fall *closer* to the desired sweet spot of $pf = 0, pd = 1$.

The above definitions of the pd, pf measures apply to two-class systems. For n-class systems, the above statistics have to be either:

- Repeated for each class in turn.
- Restricted to just binary; e.g. some target class and not the target class. The code of Figure 2 uses this second option (see line 37). The second class name of the data sets used in that study was "yes" denoting that a defective software cost module had been found.

4. Add a "Straw Man"

In his text *Experimental Methods for Artificial Intelligence*, Cohen advises comparing the performance of a supposedly more sophisticated approach against a seemingly stupider "straw man" method [3, p81]. After that comparison, the merits of the more sophisticated methods would be doubted if its performance was close to that of the "straw man". Holte's OneR algorithm is the standard "straw man" data miner and has been shown to perform remarkable well compared to standard data miners on many commonly used data sets [4].

Accordingly, line 5 of Figure 2 includes OneR in the list of learners used in this study.

5. USE SCRIPTING

Comprehensive data mining experiments can't be run via click-and-point interfaces without risking wrist damage. Applying the above principles means conducting 10*10-way experiments on multiple learners, collecting multiple performance numbers all the while. Without automatic scripts, the odds of making minor clerical errors during all that work is very high.

Another reason to use scripting is that often experiments have to repeated. There are many reasons for needing such repeat are numerous. Paper reviewers or supervisors or co-workers might suggest important minor revisions to your current experimental regime. More commonly, you realize some small error in the current rig that means all the current results are wrong, must be deleted, and recollected.

But the most important reason is that, without scripting, the researcher is hampered in their range of inquiries. Modern data mining tools come with impressive interfaces and those interfaces are both useful and a trap. If the researcher ever wants to do something *different* to what is offered by that interface, they can't. For example:

- The standard WEKA [1] toolkit has an "Experimenter" panel that can be used for conducting experiments. Sadly, the WEKA's experiments don't measures on "PRED(N)"- a value used frequently in the software cost estimation literature.
- WEKA's experimental rig can't do incremental cross-validation- which is a useful method for determining the minimum number of instances to reach stable conclusions in a domain [3].

With knowledge of scripting, however, researchers have no such limitations. Novel experiments can defined with minimal coding. For example, *over-sampling* is a method that can improve learner accuracy in data sets where the target class is in the extreme minority. To over-sample:

- First repeatedly print each instance at the inverse of its class frequency (i.e. rarer classes are printed more often than more common classes).
- Next, to generate a data set of size M, we print M randomly selected instances from that over sample.

In the `oversample` script of Figure 5, `balance.awk` handles the repeated printing and `some.awk` prints the right number of instances: This code is succinct and handles numerous tedious details such as printing the data dictionary at the top of the data files; skipping lines of blanks and comments; and controlling the random number generator.

For a good set of scripting tools, use any LINUX system and most UNIX systems. For good scripting languages, pick one of GAWK, PERL, PYTHON, RUBY,.... For the reader's information, the author has spent months to years coding in PERL, PYTHON, and RUBY but keeps coming back to GAWK.

6. PLAY NICE WITH OTHERS

Running long scripts with many calls to many learners can be very CPU-intensive. In environments where computers are a shared resource, it is polite to run your script at low priorities (lest the SS cancel your job).

```
#---- oversample --------
# usage: bash oversample File N
Opts="-v Seed=$RANDOM IGNORECASE=1 FS=, "
gawk   -f balance.awk $Opts Pass=1 $1 Pass=2 $1 \
| gawk -f some.awk    $Opts N=$2

#---- balance.awk --------
/^[ \t]*$/          { next} # skip blank lines
/^[ \t]*%/          { next} # skip comment lines
/@relation/,/@data/ { if (Pass==2) print; next}
Pass==1             { Class[$NF]++;
                      N++ }
Pass==2             { R= int(N/Class[$NF]+0.5);
                      while(R-- >0)
                         print $0}

#---- some.awk --------
BEGIN               { Seed ? srand(Seed) : srand() }
/@relation/,/@data/ { print; next}
                    { Line[rand()]= $0 } # put at random index
END                 { while(N) # print till we get enough
                         for(I in Line) {
                             if (N<1) exit
                              print Line[I];
                              N--  }}
```

Fig. 5. Oversampling

Running low-priority jobs is very simple, at least in a UNIX environment. The `nice` command controls runs a script with an adjusted scheduling priority. Priorities range goes from -20 (highest priority) to 19 (lowest- when you are being most "nice" to other users). All the scripts shown here were run as follows:

```
nice -n 19 script
```

7. USE A SANDBOX (FOR TEMPORARIES)

The case for randomizing the input space was discussed above. The code of Figure 5 shows a simple randomization method. With small changes, it can be used to output the bins required for a 10*10-way. Line 38 of Figure 2 shows where such a sampler can be called.

A 10*10-way generates 200 temporary data files (100 training and test files). A *sandbox* is a temporary directory created to store such temporaries. If the right naming conventions are used, programmers can ensure that no other such sandbox exists on the current systems:

```
#---- sandbox --------
# usage: sandbox name
Root=/tmp/$USER
Root=''$Root/$1''
subdir=''$Root/$$.$RANDOM''
mkdir -p $subdir
[ -d ``$Root/last'' ]  && unlink ``$Root/last''
ln -sf ``$subdir'' ``$Root/last''
echo $subdir
```

A useful convention is to create a test and training directory with the same file names in each and $train/x$ is the training set associated with $test/x$. Lines 38 to 49 illustrate that idiom which can be summarized as follows:

```
Sandbox=`sandbox myExperiments`
# send training/test files to $Sandbox/train, $Sandbox/test
( cd $Sandbox/test
  for i in *
  do
     Test=$i
     Train=../train/$i
     run $Train $Test> $Test.out
  done
)
rm -rf $Sandbox
```

8. Conduct Interium Purges

Note the last line of the above code where the temporaries are deleted. Such intra-run purges are important: real data mining experiments can generate 100,000s of files that can consume all disk space.

9. Cache Interium Results

The above purge actually adds a bug to this code: all the results of the learning are lost! Therefore, prior to deletions, performance statistics should be copied to some safe place.

```
Cache=$USER/var/myExperiments
mkdir -p $Cache
Sandbox=`sandbox myExperiments`
# send training/test files to $Sandbox/train, $Sandbox/test
( cd $Sandbox/test
  for i in *
  do
     Test=$i
     Train=../train/$i
     run $Train $Test> $Test.out
     collectPerformanceStats $Test.out > $Cache/$Test.out
  done
)
rm -rf $Sandbox
```

10. Use a Safe Place (for Results)

The *Cache* directory used above is a place to store results generated via very long data mining runs. Such caches contain hard-won data and so should be treated very carefully. Automatic scripts should never delete files in those caches. Indeed, researchers are advised to *never* clean their cache lest they remove important files.

11. Add Seed Control

Randomization, while important, introduces debugging problems. In short, it can be hard to reproduce error conditions if those error conditions occur infrequently.

Hence, it is *strongly* advised that all random number seed be controllable via command-line parameters. We saw an example of such control above in `some.awk`:

```
BEGIN              { Seed ? srand(Seed) : srand() }
```

With this convention, it is possible to control and replay problematic code.

12. Debug with Small Examples

Building 10*10-way runs with multiple learners and multiple pre-processing steps can be complex. Before scaling up to large runs, it is advisable to debug on very small runs.

```
# ---- etc/myweka --------
Weka=''java -cp $HOME/unbox.org/data/weka.jar''
Arffs=''$HOME/unbox.org/data/arff''

j48()    { $Weka weka.classifiers.trees.J48  -C 0.25 -M 2 -t $1 -T $2; }
oneR()   { $Weka weka.classifiers.rules.OneR -B 6         -t $1 -T $2; }
bayes()  { $Weka weka.classifiers.bayes.NaiveBayes        -t $1 -T $2; }
nb()     { $Weka weka.classifiers.bayes.NaiveBayes        -t $1 -T $2; }
nbk()    { $Weka weka.classifiers.bayes.NaiveBayes -K     -t $1 -T $2; }
m5P()    { $Weka weka.classifiers.trees.M5P               -t $1 -T $2; }

# ---- J48 --------
weka j48 $1 $2

# ---- NB -------
weka nb $1 $2
```

Fig. 6. Definition of WEKA-based learners

13. Seperate "Set up" From "Run"

The `setup` function pre-computes and caches all the pre-processed files to be executed via this code. This separates the fast part of the code (pre-processing) from the slow part of the code (running the 10*10-ways). This is good practice since it lets a researcher to browse and debug any pre-processing errors.

14. Let the Master be the Servant

Given a CPU farm, it is possible to spread out long runs over multiple machines. For example, on a UNIX box where the researcher has accounts on multiple machines, then command

```
ssh machine.name nice -n 19 Learners=J48 codeOfFigure2
```

would unleash Figure 2 for just the $J48$ experiments.

For that to work, scripts like Figure 2 need to be run as either *masters* where they control the runs, or *slaves* where they run some portion of the experiment as controlled by others. Given the $bash$ scripting language, this is simple to do. For example, the command on line 5 of Figure 2 defines the list of learners to execute *unless* it is already defined:

```
Learners=${Learners="J48 oneR Nb"}
```

Note how, in the above `ssh` call, `Learners` was set on the command line. So Figure 2 would then run, just for this learner.

Of course, this implies much repeated processing of the setup command. This is another reason to separate "set up" from "run": the "set up" could be moved to another script and run once em before calling all the nways on different machines.

15. Add Hooks

Note the use on lines 21 and 49 of Figure 2 calls to functions with variable names:

```
21  $filter $Data/$what.arff  > $TrainTest
49 $learner $Train $Test      > $Test.out
```

This allows for the simple addition of *hooks* into the current system. Now, any script that is executable on the current system can be called in your experimental rig. For example, Figure 6 shows the definition of several scripts used by Figure 2. Note that they are defined outside of Figure 2 and can be used for smaller, one-off learning tasks.

16. Allow Local Environments

It is useful to separate the script from the environment and then make the script call environment-specific set-up code. This is particularly useful when running scripts on multiple CPUs: every time your scripts log into a new machine, the first they need to do is access the local environment details.

Lines 1 and 2 of Figure 2 shows that script accessing the local environment. The user's own `.bashrc` file is executed and the local definition of the `weka` scripts are defined. This author's WEKA environment was shown in Figure 6. According to the UNIX convention, these environment files are either stored in the user's root or in a `etc` directory (in this case, $USER/etc$).

References

[1] I. H. Witten and E. Frank, *Data mining. 2nd edition*. Los Altos, US: Morgan Kaufmann, 2005.
[2] D. Heeger, "Signal detection theory," 1998, available from http://white.stanford.edu/~heeger/sdt/sdt.html.
[3] P. Cohen, *Empirical Methods for Artificial Intelligence*. MIT Press, 1995.
[4] R. Holte, "Very simple classification rules perform well on most commonly used datasets," *Machine Learning*, vol. 11, p. 63, 1993.