Bayesian Anomaly Detection (BAD v0.1)

Tim Menzies TIM@MENZIES.US
Lane Department of Computer Science and Electrical Engineering, West Virginia University, USA

David Allen DAVE@ANTIFORM.COM
Portland State University, Oregon, USA

Andres Orrego ANDRES.ORREGO@IVV.NASA.GOV
Global Science & Technology Inc, Fairmont, West Virginia

Abstract

Prior experiments with Bayesian rule generation produced a scalable anytime learner. At its core, that tool computes the likelihood of new events as the product of frequencies of old events. Orrego and Menzies applied that tool to logs of an F-15 flight simulator and showed that the same tool can detect anomalous events which have not been seen previously. This paper checks the external validity of that prior experiment. In twenty-five data sets, anomalous new situations could be identified with high probabilities of detection (average pd over 80%) and low probabilities of false alarm (usually, $pf \leq 5\%$). These results strongly suggest that we can detect anomalous events, even among very large data sets.

1. Introduction

Our goal is the development of a context-aware learning system that:

- $Goal_1$: Can understand what normal means for complex software;
- $Goal_2$: Can detect anomalous situations and alert astronauts or ground control that urgent actions is required;
- $Goal_3$: Can propose control actions to drive a system from dangerous anomalous situations to safe and known operational conditions.

The rest of this paper describes experiments with BAD (a Bayesian anomaly detector), our best current candidate for achieving $Goal_1$, $Goal_2$, and $Goal_3$. Our results will be quote promising, but need further work if they are to be applied in a NASA context. Hence, our next experiments will be to see if the effects reported here can be repeated in TRICK (a simulation framework) and ARES (a CEV instantiation of TRICK).

Appearing in Workshop on Machine Learning Algorithms for Surveillance and Event Detection at the 23rd ICML, Pittsburgh, PA, 2006. Copyright 2006 by the author(s).

2. Background

BAD relies on Bayes classifiers and the SPADE incremental discretizer.

2.1. Bayes Classifiers

NaïveBayes classifiers are based on Bayes' Theorem. Informally, the theorem says $next = old * new$ i.e. what we'll believe *next* comes from how *new* evidence effects *old* beliefs. More formally:

$$P(H \mid E) = \frac{P(H)}{P(E)} \prod_i P(E_i \mid H)$$

i.e. given fragments of evidence E_i and a prior probability for a class $P(H)$, the theorem lets us calculate a posteriori probability $P(H \mid E)$. Bayes classifiers are often called *naïve* since they assume that the frequencies of different attributes are independent. In practice [Z. Zheng & Webb, 2000], the absolute values of the classification probabilities computed by Bayes classifiers are often inaccurate. However, the relative ranking of classification probabilities is adequate for the purposes of classification. Many studies (e.g. [Hall & Holmes, 2003, Dougherty et al., 1995]) have reported that, in many domains, this NaïveBayes classifiers scheme exhibits excellent performance compared to other learners. Domingoes and Pazzini [Domingos & Pazzani, 1997], offer an extensive theoretical analysis which concludes, in the vast majority of cases, the independence assumption will only confuse a NaïveBayes classifier in a vanishingly small number of cases. Hence, Domingoes and

```
 # GLOBALS: ''F'': frequeny tables;    ''I'' : number of instances;
#            ''C'': how many classes?; ''N'': instances per class
function update(class,train)
    # OUTPUT: changes to the globals.
    # INPUT:  a ''train''ing example containing attribute/value pairs
   #          plus that case's ''class''
    I++;  if (++N[class]==1) then C++ fi
   for <attr,value> in train
       if (value != "?")  then
           F[class,attr,range]++ fi
function classify(test)
   # OUTPUT: ''what'' is the most likely hypothesis for the test case.
    # INPUT:  a ''test'' case containing attribute/value pairs.
    k=1; m=2                # Control for Laplace and M-estimates.
    like = -100000          # Initial, impossibly small likelihood.
    for H  in N             # Check all  hypotheses.
    { prior = (N[H]+k)/(I+(k*C))                    #⇐P(H)
      temp  = log(prior)
       for <attr,value> in attributes
      { if (value != "?")  then
          inc= F[H,attr,value]+(m*prior))/(N[H]+m)  #⇐P(E_i | H)
          temp += log(inc)  fi
       }
       if (temp >= like) then like = temp; what=class fi
    }
    return what
```

Figure 1. A simple Bayes Classifier. "?" denotes "missing values". Probabilities are multiplied together using logarithms to stop numeric errors when handling very small numbers. The m and k variables handle low frequencies counts in the manner recommended by Yang and Webb [Yang & Webb, 2003, §3.1].

Pazzini suggest renaming NaïveBayes to *Simple* Bayes.

The function `update` in Figure 1 illustrates the simplicity of a Bayes classifier: just increment a frequency table F holding counts of the attribute values seen in the new training examples. Note that a simple Bayes classifier only needs the memory required for the `F` frequency counts, plus a buffer just large enough to hold the `test` instance passed to Figure 1's `classify` function. Hence, it can scale to very large data sets.

2.2. Discretization with SPADE

Bayes classifiers can be extended to numeric attributes in several ways. The standard kernel estimation method assumes the central limit theorem and models each numeric attribute using a single Gaussian. Other methods don't assume a single Gaussian; e.g. John and Langley's Gaussian kernel estimator models distributions of any shape as the sum of multiple Gaussians [John & Langley, 1995]. Other, more sophisticated methods are well-established [Fayyad & Irani, 1993], but several studies report that even simple *discretization methods* suffice for adapting Bayes classifiers to numeric variables [Dougherty et al., 1995, Yang & Webb, 2002].

John and Langley note that their method needs all the numeric values to build their kernel estimator. This is impractical for large data sets. Many discretization methods violate the *one pass* requirement of a data miner: i.e. the need to execute using only one scan (or less) of the data since there many not be time or memory to go back and look at a store of past instances. For example, Dougherty et.al.'s [Dougherty et al., 1995] *straw man* discretization method is *10-bins* which divides attribute a_i into bins of size $\frac{MAX(a_i)-MIN(a_i)}{10}$. If MAX and MIN are calculated incrementally along a stream of data, then instance data may have to be cached and re-discretized if the bin sizes change. An alternative is to calculate MAX and MIN after seeing *all* the data. Both cases require two scans through the data, with the second scan doing the actual binning. Many other discretization methods (e.g. all the methods discussed by Dougherty et.al. [Dougherty et al., 1995] and Yang and Webb [Yang & Webb, 2002]) suffer from this two-scan problem.

In order to process infinite streams of data, we developed a one-pass discretization method called SPADE (Single PAss Dynamic Enumeration). SPADE scans the input data once and, at anytime during the processing of X instances, SPADE's bins are available. SPADE adjusts bins (e.g. when merging bins with very small tallies), the information used for that merging comes from the bins themselves, and not some second scan of the instances. Hence, it can be used for the incremental processing of very large data sets.

Unlike standard NaïveBayes classifiers, SPADE makes no assumptions about the underlying numeric distributions. SPADE is similar to *10-bins* but the MIN and MAX change incrementally. The first value N creates one bin and sets {MIN=N, MAX=N}. If a subsequent new value arrives inside the current {MIN,MAX} range, the bins from MIN

to MAX are searched for an appropriate bin. Otherwise, a $SubBins$ number of new bins are created (default: SubBins=5) and MIN/MAX is extended to the new value. For example, here are four bins:

i	1	2	3	4	min	max
$border$	10	20	30	40	10	40

Each bin is specified by its lower *border* value. A variable N maps to the first/last bin if it is the current {MIN,MAX} value (respectively). Otherwise it maps to bin i where $border_i < N \leq border_{i+1}$. Assuming $SubBins = 5$, then if a new value $N = 50$ arrives, five new bins added above the old MAX to a new MAX=50:

i	1	2	3	4	5	6	7	8	9	min	max
$border$	10	20	30	40	42	44	46	48	50	10	50

If the newly created number of bins exceeds a *MaxBins* parameter (default=the square root of all the instances seen to date) then adjacent bins with a tally less than *MinInst* (default: same as *MaxBins*) are merged if the tally in the merged bins is less than a *MaxInst* parameter (default: 2**MinInst*). Preventing the creation of very few bins with big tallies is essential for a practical incremental discretizer. Hence, SPADE checks for merges only occasionally (at the end of each era), allowing for the generation of multiple bins before they are merged.

SPADE runs as a pre-processor to `update` to NaïveBayes. Newly arrived numerics get placed into bins and it is this bin number that is used as the $value$ passed to `update` or Figure 1. Also, when SPADE merges bins, this causes a similar merging in frequency tables entries (the F variable of Figure 1).

The opposite of merging would be to *split* bins with unusually large tallies. SPADE has no split operator since we did not know how to best divide up a bin *without* keeping per-bin kernel estimation data (which would be memory-expensive). Our early experiments suggested that adding $SubBins = 5$ new bins between old ranges and newly arrived out-of-range values was enough to adequately divide the range. Our subsequent experiments (see below) were so encouraging that we are not motivated to add a split operator.

Figure 2 compares results from SPADE and John and Langley's kernel estimation method using the display format proposed by Dougherty, Kohavi and Sahami [Dougherty et al., 1995]. In that figure, a 10*10-way cross validation used three learners:

1. NaïveBayes with a single Gaussian for every numeric;
2. NaïveBayes with s kernel estimation method
3. The Figure 1 classifier with data pre-discretized by SPADE.

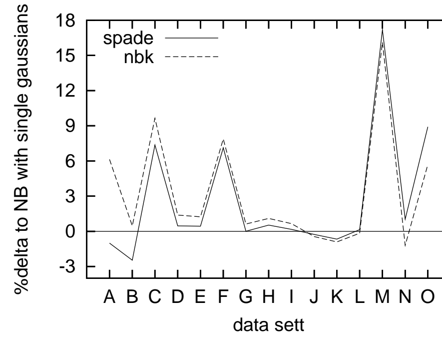

Figure 2. Comparing SPADE and kernel estimation. Data sets: {*A=vowel, B=iris, C=ionosphere, D=echo, E=horse-colic, F=anneal, G=hypothyroid, H=hepatitis, I=heart-c, J=diabetes, K=auto-mpg, L=waveform-5000, M=vehicle, N=labor, O=segment*}.

Mean classification accuracies were collected and shown in Figure 2, sorted by the means $(c-a)-(b-a)$; that is, by the difference in the improvement seen in SPADE *or* kernel estimation *over and above* a simple single Gaussian scheme. Hence, the left-hand-side data sets of Figure 2 show examples where kernel estimation worked better than SPADE, while the right-hand-side shows results where SPADE did comparatively better. In two cases, SPADE's one scan method lost information and performed worse than assuming a single Gaussian. In data set A, the loss was minimal (-1%), and in data set B SPADE's results were still within 3% of kernel estimation. In our view, the advantages of SPADE (incremental, one scan processing, distribution independent) compensate for its occasionally performing less well than state-of-the-art alternatives, which require far more memory.

3. BAD

BAD implements incremental anomaly detection by monitoring the internals of a simple Bayes classifier. The core of the `classify` function is the computation of the likelihood `like` value using the product of a set of frequency counts divided by the frequency of each class These numbers are normalized between one and zero. If new examples fall into poorly sample portions of the past behavior, then that product will include several unusually small frequency counts . The product of these reduced values will, in turn, be much smaller than the usual likelihoods. That is, anomalies can be detected when the likelihoods start dropping.

For example, Figure 3 shows an *unsupervised learning* experiment (where instances lack any class symbol). In Figure 3, the modes of an F-15 simulator were labelled with one of several symbols: nominal, errorA, errorB, etc. For this experiment, when this data was passed to an incre-

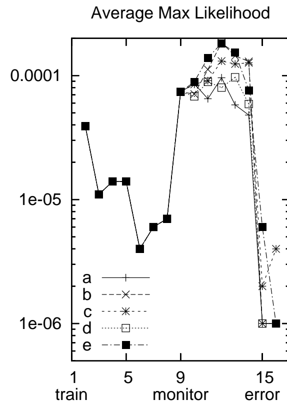

Figure 3. Learning normal flight (eras 1 to 8); monitoring five different flights a,b,..e (eras 9 to 16); injecting errors into eras 15,16.

mental simple Bayes classifier (running SPADE), the class of all instances were replaced with a single label: $class0$. The data was processed in *eras* of 100 instances. The first eight eras (800 instances) show the learner nominal flight simulator data. Updating of the frequency tables was then disabled and the system watched over five entirely different flights, each ending with one of our errors *a,b,c,d,e*. The `classify` routine of Figure 1 was modified to return the classification with the maximum likelihood, *as well as* that maximum likelihood value. Figure 3 shows the average maximum likelihood seen in each era. In all cases, the era 15,16 errors dramatically changed the likelihoods: they dropped by *two orders of magnitude* from the pre-error values, and dropped *below* the likelihoods seen during training (eras 1 to 8).

While an encouraging result, the results of Figure 3 come from a single data set. The rest of this paper explores the external validity of Figure 3. We will see that in 25 data sets, BAD could identify the point at which a previously unseen class appeared in a dataset.

4. Experiments

In the following experiments, data sets from the UCI database are sorted such that a jumbled set of N classes are presented to BAD followed by a new class $N+1$. It will be shown that BAD can recognize the moment when the anomaly that has not been seen before (New) presents itself. To make the experiment interesting, the rig runs in a one-pass incremental learning mode (this is the mode required when processing very large data streams).

4.1. Data Filtering

A supervised batch learner can search all the data before drawing any conclusions. An unsupervised incremental learner, on the other hand, makes it conclusions using just part of the data:

- i.e. just the data seen to date
- i.e. just the independent attributes without any knowledge of the dependent class values.

Hence, by definition, an incremental unsupervised learner *must* perform worse than a supervised batch learner.

A corollary of this is that we should not ask an unsupervised incremental learner to find what can't be found by a supervised batch learner. Consequently, we remove from the data sets classes with PDs (as found by supervised batch learning) below a certain threshold $T\%$. In our experiments we used $T \in \{80, 60, 40, 20\}$ and, for reasons of space, present only $T \in \{80, 20\}$ Note that at $T = 80\%$, we are searching for classes that are very easy to find while at $T = 20\%$, we are searching classes that are very hard to find.

Phase one of our experiments was to remove classes with a supervised batch $PD < T$. The filtering was performed iteratively as follows. WEKA's [Witten & Frank, 1999] simple Bayes Classifier was run on the data. After each iteration, the class with the lowest probability of detection lower than I was removed. This was repeated until all remaining classes had a PD higher than or equal to the threshold.

Any data sets that ended up with zero or one class were removed. At low T values, all the data sets were used but at high T values, 40% of the classes were pruned. The results of pruning at $PD < 0.8$ are shown in Figure 4. Ten data sets had no classes with $PD \geq 0.8$ (*breast-cancer, colic, credit-a, credit-g, diabetes, heart-c, heart-h, sick, sonar, vehcile*) so these were not used in the $T < 80\%$ experiments.

4.2. Training/Test Set Generation

Having filtered the classes, the data within each data set was sorted ten times as follows. All our subsequent results are averages over the ten trial.s

If a data set contained (say) classes A, B, etc and C, then three experiments were performed: one each for A / not-A, B / not-B, etc. For each class in a data set, a pair of files were created, one for the training set, and one for the test set.

For a training set, if the target class was A, then the set was created by sampling instances from the filtered data that were *not* classified as A. The number of instances per class was chosen randomly between 300 and 3000 and the instance sampling from each class was done randomly with replacement. In the final file, the instances were grouped by class, with the class order randomly chosen.

dataset	Classes	Attributes	Instances
anneal.arff	5 / 5	38	898 / 898
audiology	24 / 4	69	226 / 134
autos	6 / 3	25	205 / 103
breast-cancer	2 / 0	9	286 / 0
colic	2 / 0	22	368 / 0
credit-a	2 / 0	15	690 / 0
credit-g	2 / 0	20	1000 / 0
diabetes	2 / 0	8	768 / 0
heart-c	2 / 0	13	303 / 0
heart-h	2 / 0	13	294 / 0
hypothyroid	4 / 2	29	3772 / 3576
ionosphere	2 / 2	34	351 / 351
kr-vs-kp	2 / 2	36	3196 / 3196
letter	26 / 11	16	20000 / 8498
mushroom	2 / 2	22	8124 / 8124
primary-tumor	21 / 3	17	339 / 133
segment	7 / 5	19	2310 / 1650
sick	2 / 0	29	3772 / 0
sonar	2 / 0	60	208 / 0
soybean	19 / 18	35	683 / 592
splice	3 / 3	61	3190 / 3190
vehicle	4 / 0	18	846 / 0
vote	2 / 2	16	435 / 435
vowel	11 / 5	13	990 / 450
waveform-5000	3 / 2	40	5000 / 3308

Figure 4. Data sets before/after pruning classes with $PD < 0.8$.

The test sets contained two parts:

Test1: If the target class was A, then the first part contained only instances that were *not* classified as A. These instances were chosen first by randomly selecting a not-A class, then by randomly sampling an instance of that class with replacement.

Test2: The other part contained only instances of class A, selected randomly with replacement from the data set.

The number of instances for both sections were chosen independently, randomly selecting a size between 600 and 3000. The values for minimum train and test size (multiples of 300) come from the SAWTOOTH research. That work showed that the performance of many learners plateaued after a few hundred instances. Very few needed more than 300 instances to learn all that they could from the source data. Note also that the way our rig is defined, if a data set has less than 300 (or 600) instances, this experiment just over-sampled (at random, with replacement) the available data.

Figure 5 details the training and test sets generation method.

4.3. Training

Training was performed on the training sets. For a particular pair of $trainingSet, testSet$ files, the training set contains all classes from the data set except one.

Training on these sets was performed exactly as it was done in the original SAWTOOTH [Menzies & Orrego, 2005]. Training was performed on instances with a window size of 150, in a supervised mode using a simple Bayes classifier.

```
trainMin = 300
trainMax = 3000
testMin  = 600
testMax  = 3000

foreach targetClass in dataSet
{  #----------- generate training data -----------
   tempClassList = Classes( dataSet );        # all classes
   RemoveClass( tempClassList, targetClass ); # one class
   classOrder = RandomOrder( tempClassList ); # randomize
   foreach class in classOrder
   {  instanceArray = GetInstanceArray( dataSet, class );
      instanceCount = random number in trainMin ... testMin;
      foreach ( 1 ... instanceCount )
      {  # randomly select one instance
         instance = SelectRandomEntry( instanceArray );
         # write instance to training file
         WriteTrainingFileInstance( instance );
   }  }
   #----------- generate test data -----------
   instanceCount = random number in testMin ... testMax;
   # write theses  instances from all classes but target
   foreach ( 1 .. instanceCount )
   {  class         = SelectRandomEntry( classOrder );
      instanceArray = GetInstanceArray( dataSet, class );
      instance      = SelectRandomEntry( instanceArray );
      WriteTestFileInstance( instance );
   }
   instanceArray = GetInstanceArray( dataSet, targetClass );
   instanceCount = SelectRandomNumber( 600, 3000 );
   foreach ( 1 .. instanceCount )
   {  # randomly select one instance
      instance = SelectRandomEntry( instanceArray );
      WriteTestFileInstance( instance );
}  }
```

Figure 5. Generating training/test data.

Once stability was achieved updates to the theory are disabled. If instability returned, learning was re-enabled until stability returned. SPADE used to incrementally discretize numeric attributes for the Bayes Classifier.

4.4. Testing

Testing followed the same outline as training, but instead of training in an supervised mode, we attempted to detect when the new class appeared in an unsupervised mode. The code is shown in Figure 6. Just like during training, the era window size was set to 150, stability was tested with the statistical z-test, and learning was re-enabled when the behavior became unstable.

The statistical tables for the simple Bayes classifier and the bins for the SPADE discretizer were initialized during training. At the beginning of testing, the first era was considered to be stable by default.

Since testing was run in an unsupervised mode, the classes of the instances in the test data were ignored. Instead of running the z-test on statistics of successful instance classification, it was run on statistics of the of the maximum likelihood returned by classification. In general, if an instance belongs to a class that the simple Bayes classifier has been trained on, the maximum likelihood of that classification would be relatively high. If the instance belongs

```
# Globals:
#   testSet: the test data set
#   zThresh: the threshold for the z-test
#   expectedEra: the era where the new class appeared in the test data
#                only used for scoring
        # ---------- initialize ----------
        eraNum = 1;          # current era number
        stableCount = 3;     # number of stable eras
        detectedClass = -1;  # number of the detected classes
        totalCount = 0;      # running total of instances
        totalSum = 0;        # running sum of log max likelihoods

        # ---------- run over test set eras --------
        foreach era in testSet
        {       instanceCount = 0;
                logMaxLikelihoodSum = 0;
                foreach instance in era
                {       # classify instance, get the log of the max likelihood
                        logMaxLikelihood                 = log( Classify( instance ));
                        logMaxLikelihoods[instanceCount]  = logMaxLikelihood;
                        logMaxLikelihoodSum              += logMaxLikelihood;
                        if ( detectedClass >= 0 )
                                SetClass( instance, "__" + detectedClass + "__" );
                        instanceCache[ instanceCount ] = instance; # cache
                        ++instanceCount;                           # increment
                }

                eraMean  = logMaxLikelihoodSum / instanceCount;
                eraStdev = CalculateStdDev( logMaxLikelihoods );
                # handle the first era, this forces the first era to be treated as stable
                if ( eraNum == 1 ) totalMean = eraMean;
                # standardized z-test
                zTest = ZTest( eraMean, totalMean, eraStdev, instanceCount );

                if ( zTest < zThresh )   # if unstable
                {       # only count as a new class if we are currently stable, otherwise
                        # a double detection could result
                        if ( stableCount >= 3 )
                        {       ++detectedClass;                          # new class number
                                detectedClassEra[detectedClass] = eraNum; # record where found
                        }
                        stableCount = 0;                                  # mark unstable
                }
                else                                                      # else, keep tracking
                {       totalCount += instanceCount;
                        totalSum   += logMaxLikelihoodSum;
                        totalMean   = totalSum / totalCount;
                }

                if ( stableCount < 3 )                                    # train if recently unstable
                {       Train( instanceCache );
                        # reset total values while unstable
                        totalCount = instanceCount;
                        totalSum = logMaxLikelihoodSum;
                        totalMean = eraMean;
                }

                ++eraNum;                   # count eras
                ++stableCount;              # count stable eras
                delete instanceCache;       # prep for next era
                delete logMaxLikelihoods;   # prep for next era
        }
    <PD, FP> = ScoreResult( expectedEra, detectedClassEra );
```

Figure 6. Training.

Data Set	Classes	PD%	FP%
segment	5	88.0	0.0
soybean	18	94.4	0.6
letter	11	99.1	0.0
audiology	4	100.0	0.0
ionosphere	2	100.0	0.0
kr-vs-kp	2	100.0	0.0
mushroom	2	100.0	0.0
primary-tumor	3	100.0	0.0
splice	3	100.0	0.0
vote	2	100.0	0.0
vowel	5	100.0	0.0
waveform-5000	2	100.0	0.0
anneal	5	100.0	2.0
autos	3	100.0	3.3
hypothyroid	2	100.0	5.0
	average:	98.8	0.7

Figure 7. Minumum PD = 0.8, z-test α = 0.00001.

to a new class, the maximum likelihood will be relatively low. The z-test was used to recognize when the test data switched from instances of known classes, to instances of the new class.

The threshold for the z-test comparison was passed into the algorithm. When the result of the z-test dropped below this threshold, the era was marked as unstable and learning was re-enabled. Each time this occurred, a new class was created and later instances were marked as this class. Detection of new classes was then turned off until stability was achieved to prevent duplicate detection of the new class. After three eras of stable behavior the data was marked as stable and learning was disabled.

To score the results of testing, the code stored the era where each new class was detected. Depending on where within an era the new class appeared, the new class might be detected within the era where it first appeared, or it might be detected in the next era. In either case this class detection was counted as a positive detection (PD). Any other class detections were counted as false positives (FP).

4.5. Results

Initially, large false alarm rates were were seen until the z-statistic α threshold was adjusted from 0.01 to 0.00001. The results for $\{\alpha = 0.00001, T = 80\%\}$ are shown in Figure 7. Since $T = 80\%$ selects for the most detectable classes so it is hardly surprising that these results are quite good: mean $PD \approx 99\%$ and mean $PF < 1\%$.

The $T = 20\%$ results are shown in Figure 8. Note as T decreases, we search for a larger number of less detectable classes (so the number of rows in the results tables increase). The hardest test of our rig is Figure 8 where BAD searches for anything down to $T = 20\%$. Even here, the results are promising: mean $PD \approx 80\%$; mean $PF < 5\%$.

Data Set	Classes	PD%	FP%
credit-g	2	30.0	35.0
breast-cancer	2	50.0	0.0
sick	2	50.0	0.0
waveform-5000	3	53.3	10.0
diabetes	2	55.0	10.0
primary-tumor	10	56.0	10.0
vehicle	3	56.7	0.0
vowel	11	74.5	4.5
colic	2	75.0	20.0
letter	26	82.7	0.8
autos	6	86.7	0.0
splice	3	93.3	6.7
soybean	19	94.7	0.5
kr-vs-kp	2	95.0	0.0
segment	6	95.0	0.0
hypothyroid	3	96.7	0.0
anneal	5	100.0	0.0
audiology	8	100.0	0.0
credit-a	2	100.0	0.0
heart-c	2	100.0	0.0
heart-h	2	100.0	0.0
ionosphere	2	100.0	0.0
mushroom	2	100.0	0.0
sonar	2	100.0	0.0
vote	2	100.0	0.0
	average:	81.8	3.9

Figure 8. Minimum PD = 0.2, z-test α = 0.00001.

5. Conclusion

Repeating our introduction, our goal is the development of a context-aware learning system that:

- $Goal_1$: Can learn "normal" for complex software;
- $Goal_2$: Can detect anomalous situations and alert astronauts or ground control that urgent actions is required;
- $Goal_3$: Can propose control actions to drive a system from dangerous anomalous situations to safe and known operational conditions.

It turns out that $Goal_1$ and $Goal_2$ are complimentary: if a system understands "normal" it can recognize "anomaly" and vice versa. Hence, this paper has explored the speculation that an incremental simple Bayes classifier can also be used for $Goal_1$ and $Goal_2$. There are many systems engineering advantages to using one tool for multiple purposes, not the least of which is that optimizations of that single tool can benefit many purposes. For example, the particular data miner use here has a very small memory footprint and hence should scale to very large data sets.

Our results are very promising: even for hard to detect classes, this rig had over an 80% probability of detection of the arrival of a class that has never been seen before. Hence, we are motivated to persist with this approach.

Nevertheless, while promising, these results are not based based on NASA flight systems. Our next experiments plans to use TRICK (a simulation framework) and ARES (a CEV instantiation of TRICK) to attempt the construction of a real-time flight monitor and repair agent for CEV.

References

Domingos, P., & Pazzani, M. J. (1997). On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, *29*, 103–130.

Dougherty, J., Kohavi, R., & Sahami, M. (1995). Supervised and unsupervised discretization of continuous features. *International Conference on Machine Learning* (pp. 194–202).

Fayyad, U. M., & Irani, I. H. (1993). Multi-interval discretization of continuous-valued attributes for classification learning. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence* (pp. 1022–1027).

Hall, M., & Holmes, G. (2003). Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions On Knowledge And Data Engineering*, *15*, 1437– 1447.

John, G., & Langley, P. (1995). Estimating continuous distributions in bayesian classifiers. *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence Montreal, Quebec: Morgan Kaufmann* (pp. 338–345). Available from `http://citeseer.ist.psu.edu/john95estimating.html`.

Menzies, T., & Orrego, A. (2005). Incremental discreatization and bayes classifiers handles concept drift and scaled very well. . Submitted, IEEE TKDE, Available from `http://menzies.us/pdf/05sawtooth.pdf`.

Witten, I. H., & Frank, E. (1999). *Data mining: Practical machine learning tools and techniques with java implementations*. Morgan Kaufmann.

Yang, Y., & Webb, G. (2003). Weighted proportional k-interval discretization for naive-bayes classifiers. *Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003)*. Available from `http://www.cs.uvm.edu/~yyang/wpkid.pdf`.

Yang, Y., & Webb, G. I. (2002). A comparative study of discretization methods for naive-bayes classifiers. *Proceedings of PKAW 2002: The 2002 Pacific Rim Knowledge Acquisition Workshop* (pp. 159–173).

Z. Zheng, Z., & Webb, G. (2000). Lazy learning of bayesian rules. *Machine Learning*, *41*, 53–84. Available from `http://www.csse.monash.edu/~webb/Files/ZhengWebb00.pdf`.