# Augmenting Network Intrusion Detection Systems (IDS)

**With Machine Learning**

1

# CS 590 Computer Network Defense

27 April 2011

# Alex Towell

# Abstract

Intrusion Detection Systems (IDS) of today are inherently prone to false positives and false negatives because types of attacks are all weighted equally and responded to with the same Computer Emergency Response tactics, techniques and policies while not associating the cost to business. The augmentation of the traditional IDS with machine learning to dynamically classify, weight and associate cost to a specific type of intrusion event can enable a business to “calibrate” their IDS and avoid costly reactionary measures to a low risk threat.

**IDS: Using A Cost Matrix To Tune Classifiers**

Intrusion detection systems (IDS) are a good second line of defense in network security, and arguably their importance is growing given the severity, sophistication, and accessibility (e.g., script-kiddies) of attacks. There are two primary ways to detect intrusions on a network. One approach is called signature-based detection. It consists of a database of signatures (patterns) for known attacks, and it tries to match these signatures to network traffic. Close matches (according to some distance measure and threshold) are flagged as a threat. The other approach to intrusion detection is called anomaly-based detection. It characterizes “normal” network traffic and flags any outliers (according to some distance measure and threshold) as a threat.

Both have their advantages and disadvantages with respect to each other. For instance, an advantage of signature-based detection is that it offers more discriminating alarms, e.g., it may sound the alarm on a specific kind of attack whereas anomaly-based detection will only tell you whether the traffic is normal or abnormal. The capacity to sound a specific kind of alarm can be useful. However, the cost of a false alarm may be greater than the cost of failing to detect minor attacks, whereas the cost of failing to detect major attacks may be much greater than the cost of a false alarm. An advantage of anomaly-based detection is that, since they flag “abnormal” traffic, they can in theory detect previously unknown attacks, e.g., zero day attacks. (It is worth noting that you can combine both approaches together.)

## False Positives and False Negatives

Unfortunately, both signature-based detection and anomaly-based IDS suffer from two types of errors. The first kind of error is called a false alarm, or a false positive. A false positive is when you flag network traffic as a threat when it is not. Arguably, false alarms are the limiting factor on the performance of an IDS system. After all, each alarm must potentially be investigated by human operators who are in limited supply and, therefore, relatively costly. The second kind of error is a false negative. A false negative happens when an alarm is not sounded for an attack that actually does take place, e.g., an undetected SQL injection attack. So, in both anomaly-based detection and signature-based detection, we must be worried about the false positive rate and the false negative rate. If we have too many false positives, then its results tend to be ignored or neglected. Conversely, if we have too many false negatives, then a lot of attacks will not be detected. Finding the right balance between these two extremes can be difficult.

## Expected Cost

There is also the issue of cost. In machine learning, optimizing classification rates without considering the cost of errors can lead to rather unexpected results. One example of this can be found in an early machine learning effort used to predict when each of the cows in a dairy herd were in heat (in modern dairy farms it is important to know when a cow is ready to be artificially inseminated) using observations like the volume of milk produced. However, since a cow is in heat only once every thirty days, it turns out to be a fairly good strategy to always predict “not in heat.” This leads to an impressive error rate of only ~3%. However, this is not what was intended: rather, they wanted to predict “in heat” far more accurately than “not in heat,” because the error costs are not uniform – that is, the cost of predicting “not in heat” when it is in heat is far greater than the cost of predicting “in heat” when it is not in heat. Likewise, with intrusion detection, the cost of predicting a “denial of service” attack when it is normal may be far greater than predicting “normal” when it is a denial of service attack. Of course, what they were interested in was when the cows were in heat, but because the machine learning classifier they used was trying to predict the class with the lowest error rate without respect to the actual cost of the different kinds of errors—false negatives or false positives—they instead got this strange result.

However, if you have the relative costs of each type of error in hand, if a classifier produces a probability distribution rather than a single predicted class, the classifier can be calibrated to minimize the expected cost rather than simply choosing the most likely outcome. In fact, being able to specify the relative costs may address both the need to reduce the error rate (e.g., false positive rate) and the need to be able to make the classifier cost-sensitive, e.g., avoiding always predicting cows are “not in heat” when what is desired is to know when they actually are in heat

In the two class case, e.g., “normal network traffic” or “not normal network traffic,” we have only two kinds of errors, false positives and false negatives. If the costs are uniform, which is unlikely, then predicting the class that minimizes the expected cost is the same thing as predicting the most probable class. However, when more likely non-uniform costs are used, e.g. false alarms (false positives) are more costly than false negatives; therefore, the “best” class to predict depends on the class probability distribution and the relative associated costs of false negative and false positive. These costs can be represented by a 2x2 cost matrix, as depicted below in Table 1.

Table 1 2x2 Cost Matrix

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **Predicted class** |  |
|  |  | normal | not normal |
| **Actual class** | normal | 0 | 3 |
|  | not normal | 1 | 0 |

The diagonals (the 0s in this case) represent the correct classifications, true positives and true negatives, and the off-diagonals represent the two types of errors, false positives and false negatives. So, suppose the predicted probabilities for class “normal” is P(normal) = 0.3 and class “not normal” is P(not normal) = 0.7. To compute the expected costs for predicting normal and not normal, we simply compute the vector product of the class probability distribution and the associated matrix column for the predicted class. Thus, the expected cost for predicting the “normal” class is 0 \* P(normal) + 1 \* P(not normal) = 0.7, and the expected cost of predicting the “not normal” class is 3 \* P(normal) + 0 \* P(not normal) = 0.9. To minimize the expected cost, we choose the class with the smallest expected cost, which is “normal” in this case. This is despite the fact that, according to the classifier’s probability distribution, “not normal” is in fact more likely. However, since responding to false alarms may be very costly–after all, high false alarm rates are one of the limiting factors in adopting IDS–it makes sense to factor in the associated error costs to tune our classifiers to real-world needs. Evidently, this cost matrix suggests that false alarms about being “not normal” are typically more costly than failing to detect a “not normal” event.

The cost matrix can be generalized to an **n x n** square matrix when we have n classes instead of 2. What follows is a representation of the cost matrix we developed for the modified training and test sets.

**Table 2: 4x4 Cost Matrix**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **Predicted class** |  |  |  |
|  |  | normal | nmap | teardrop | warezclient |
| **Actual class** | normal | 0 | 30 | 1 | 4 |
|  | nmap | 1 | 0 | 1 | 1 |
|  | teardrop | 3 | 1 | 0 | 1 |
|  | warezclient | 1 | 1 | 1 | 0 |

In the above cost matrix, we have 4 classes, each representing a certain kind of attack, or normal. NMAP is network popular probing/scanning software, teardrop is a distributed denial of service attack, and warezclient is a method by which people can download “warez” through the system. As can be seen from the cost matrix, incorrectly classifying all but the teardrop attack when a normal attack is actually happening is heavily penalized (by a factor of 30 in the nmap case). Teardrop is not severely punished because it is considered, for this experiment, severe enough to warrant detection even at the expense of contributing higher false alarms. Indeed, predicting normal when the actual class is teardrop is penalized. To calculate the expected cost of making a certain prediction, the same technique used for the 2x2 matrix is used: take the dot product for each column (where the column header represents the predicted class) in the matrix and the class probability distribution. For instance, the expected cost of predicting nmap is P(normal) \* 30 + P(teardrop) + P(warezclient), the expected cost of predicting normal is P(nmap) + P(teardrop) \* 3 + P(warezclient), and so on for the other two columns. Then, you choose the class prediction that has the lowest expected cost.

So far, we have assumed that the classifier predicts a class probability distribution, rather than a single class. For instance, classifiers like NaïveBayes and BayesNet natively work with probability distributions. Therefore, they do not simply predict a class. Rather, for each instance, they predict the probability of it belonging to each class (where the total probability over the entire class probability distribution correctly sums to one). This is what is required to use the cost matrix.

The good news is that, in a sense, all classifiers can be viewed as predicting a class probability distribution–even classes that seemingly have nothing to do with probability distributions. As a result, the **cost matrix** can be generically applied to almost every popular classifier in use after a few tweaks. We will go over a few popular classifiers, suggesting possible tweaks so that they predict a class probability distribution for the purpose of using a cost matrix to minimize expected cost.

## Evaluating performance of a classifier

However, before we begin, some background information about how we collected data. Perhaps the biggest challenge in machine learning and data mining is finding enough quality data to train and test classifiers with. For instance, when doing supervised learning, one must find labeled data in sufficient quantity. This can be quite difficult, and it may require a human expert to manually classify instances. Unfortunately, the problem of limited data is compounded by the fact that, in order to evaluate the performance of various supervised classifiers, it is not sufficient to evaluate their performance based on how well they classify data in the training set – not only do we already have the class labels for instances in this data set (indeed, that is why we may use them for training), but since we constructed the model based on this data set, its performance measure on this data set (training set) will be aggressively optimistic.

To understand why this is so, consider the fact it is not difficult to construct a classifier that can classify all the instances in a training set with 100% accuracy (unless the training set has some instances which are alike in every respect except their class labels, which signifies the presence of noise in the training set). Indeed, simply storing all the instances in a table and then doing table lookups accomplishes this task. Unfortunately, this table would be unable to classify previously unseen instances (that come from the same distribution as the instances in the training set); that is, it does not generalize well. This is because the rules generated–in this case, storing all the instances in the training set–are too specific (i.e., the learned model does not generally apply).

This is called over-fitting (or over-training), where the classifier is too finely tuned to the distribution of instances in the training set, which makes it underperform on instances not represented in the training set. To learn useful models, we need to evaluate their performance on independent test sets which were not used during training to penalize over-fitting on the training set. (Other techniques, like learning biases which favor simplicity, like Occam’s razor, may also allow a classifier to generalize more effectively to instances not present in the training set). So, the bottom line is, unless all the instances in the population are known, one must avoid over-fitting on the training set by evaluating the learned model on an independent test set.

However, ideally the training set will be as large as possible; the larger the training set, the greater opportunity there is for the classifier to learn the nuances of the data. In addition, we would like the test set to be as large as possible; the more test data we have, the more confident we can be in its performance measures.

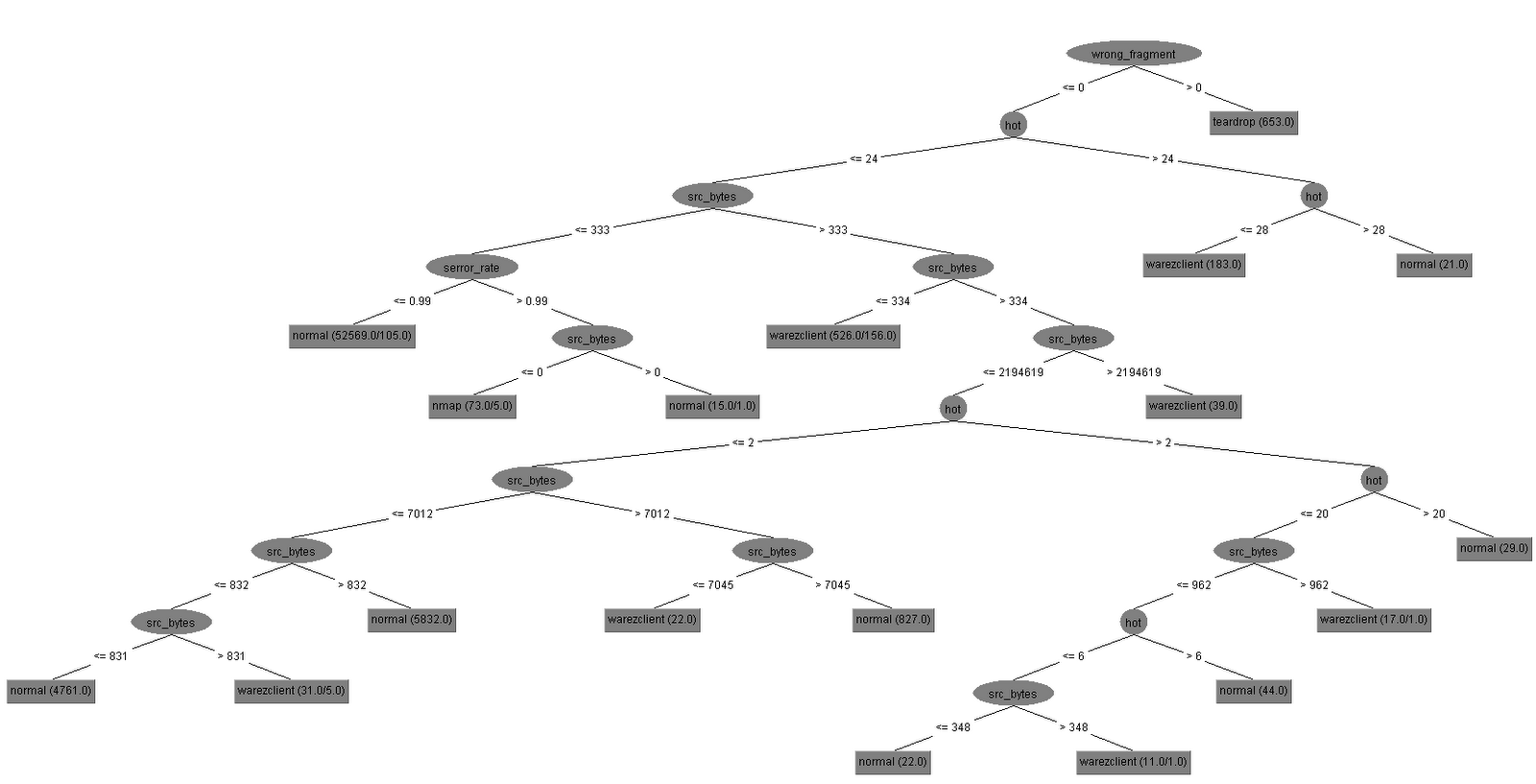
So, in light of this, how much data should be set aside for testing? We have several options available to us. If our data set is limited, we could use N-fold cross validation. To do N-fold cross-validation, we partition the data into N parts. Then, we hold out one of the parts as the test set, and use the remaining n-1 parts as the training set. Then, we run the classifier on the training set and evaluate its model on the test set. We do this for each of the n parts, such that we use the entire data set for both training and testing. Finally, we average the results for each part to arrive at a final performance measure. This addresses the desire to maximize the size of the training set and the test set.

However, if the data set is sufficiently large, one can simply use a large sample as the training set and another large sample as the test set. This is in fact our approach, since we were able to find a very large data set of labeled network traffic (KDD-99\_CUP)2. In fact, our data set was so large that most classifiers (any classifiers which needed to have the entire training set available to it for learning and classification, like instance-based classifiers that need to store the entire training set in main memory) were unable to cope with it.

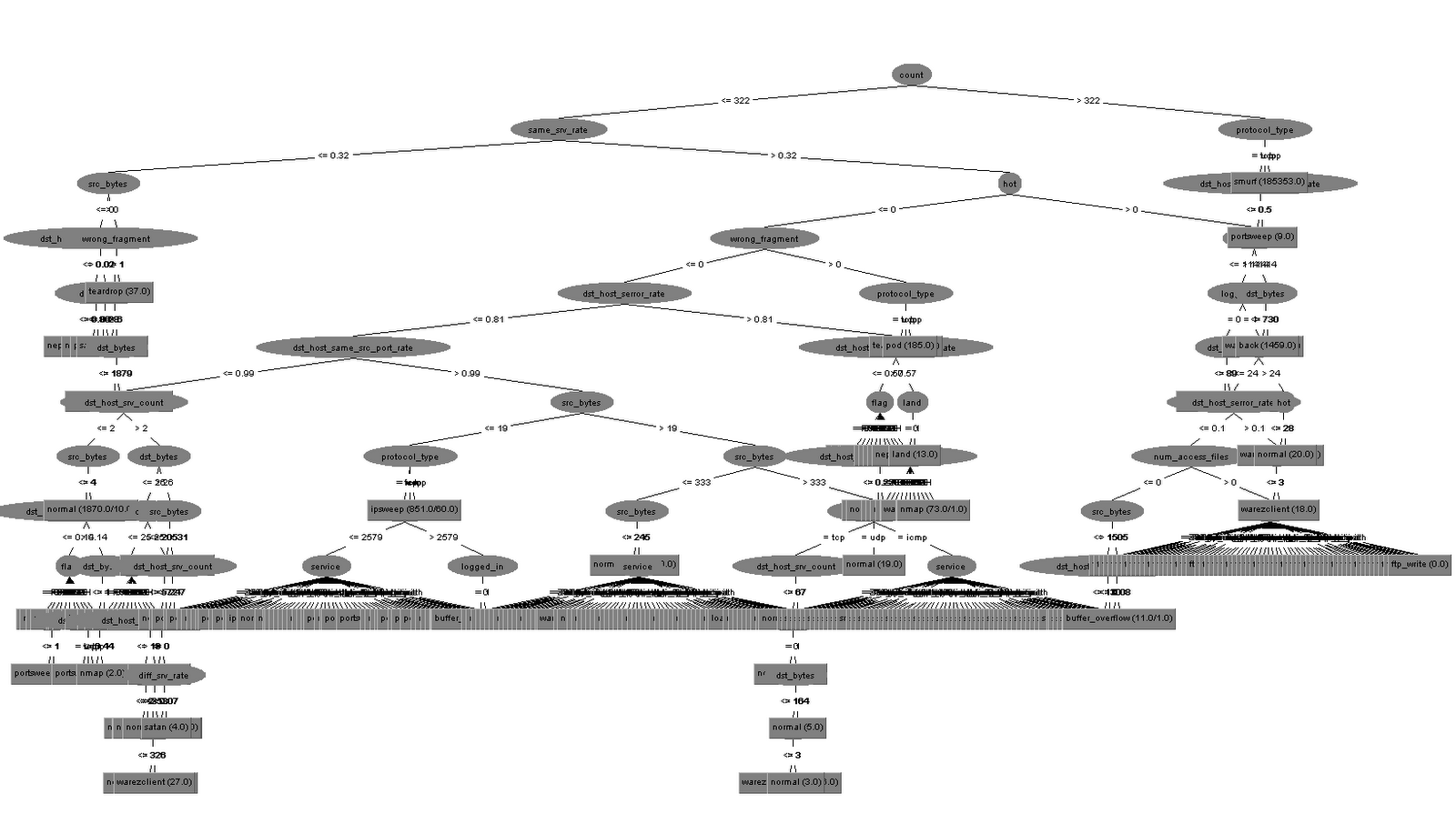
To address this problem, and also to improve the performance and comprehension of our results, we performed the following preprocessing steps. First, we used a 10 percent stratified re-sampling of the original data set. This is legitimate due to the law of diminishing returns. Most classifiers flatten out on performance long before seeing every instance in large data sets. Second, for the sake of making the data more understandable and digestible, we eliminated most of the nominal class values; we went from 23 classes to only 4. Third, for the sake of reducing time and space complexity, we eliminated the least “useful” attributes using greedy subset attribute selection, productively reducing the number of attributes from an excessive 43 attributes to a more reasonable 5 attributes. This is justified as a way to improve the real-world performance of most classifiers, even though they are, in theory (with a few exceptions), supposed to ignore useless or redundant attributes. Fourth, we partitioned the data set into a training set and a test set, where each set had the same class proportions (of those classes remaining) as the original data set. Ultimately, the training set consists of 60% of the data, and the test set consists of 40% of the data.

## J48 decision tree classifier

To demonstrate the simplifications graphically, below are two graphs. Figure 1 is for the simplified data set modified as previously described and the bottom, Figure 2, is the re-sampled data set (using stratified re-sampling to reduce its size so that the J48 tree inducer could work with it without running out of memory).



**Figure 1 Simplified Decision Tree**



**Figure 2 Original Decision Tree**

As you may be able to see, the decision tree generated in figure 2 is vastly more complex than the decision tree generated in figure 1; actually, it is hard to see, but there are hundreds of leaf nodes smashed together in Figure 2. This is not necessarily a bad thing (although most data sets should be preprocessed to eliminate useless attributes), but it does serve our expository purpose.

To help the reader to see the tree structure in Figure 1, here it is again shown in text format:

wrong\_fragment <= 0

| hot <= 24

| | src\_bytes <= 333

| | | serror\_rate <= 0.99: normal (52569.0/105.0)

| | | serror\_rate > 0.99

| | | | src\_bytes <= 0: nmap (73.0/5.0)

| | | | src\_bytes > 0: normal (15.0/1.0)

| | src\_bytes > 333

| | | src\_bytes <= 334: warezclient (526.0/156.0)

| | | src\_bytes > 334

| | | | src\_bytes <= 2194619

| | | | | hot <= 2

| | | | | | src\_bytes <= 7012

| | | | | | | src\_bytes <= 832

| | | | | | | | src\_bytes <= 831: normal (4761.0)

| | | | | | | | src\_bytes > 831: warezclient (31.0/5.0)

| | | | | | | src\_bytes > 832: normal (5832.0)

| | | | | | src\_bytes > 7012

| | | | | | | src\_bytes <= 7045: warezclient (22.0)

| | | | | | | src\_bytes > 7045: normal (827.0)

| | | | | hot > 2

| | | | | | hot <= 20

| | | | | | | src\_bytes <= 962

| | | | | | | | hot <= 6

| | | | | | | | | src\_bytes <= 348: normal (22.0)

| | | | | | | | | src\_bytes > 348: warezclient (11.0/1.0)

| | | | | | | | hot > 6: normal (44.0)

| | | | | | | src\_bytes > 962: warezclient (17.0/1.0)

| | | | | | hot > 20: normal (29.0)

| | | | src\_bytes > 2194619: warezclient (39.0)

| hot > 24

| | hot <= 28: warezclient (183.0)

| | hot > 28: normal (21.0)

wrong\_fragment > 0: teardrop (653.0)

Initially, the J48 tree inducer uses a recursive algorithm to divide and conquer the instances, preferring to branch on attributes that maximize the information gain until it (usually) perfectly classifies every instance in the training set. However, this level of accuracy is almost certainly over-fitting the training set, and so pruning operations are performed to reduce the size of the tree. To see if a pruning operation is warranted, a test set can be used to evaluate the performance of the pre-pruned and post-pruned tree, and then choose the most accurate one on the test set. After a series of pruning operations, the tree will be more general (and simpler).

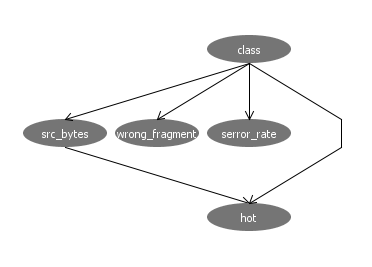
At this point, one can argue that the tree essentially represents a class probability distribution. That is, most of its leaf nodes, due to the pruning operations, have distributions of multiple classes in it. A reasonably simple tweak, therefore, can be used to make the decision tree provide explicit class probability distributions: for each class, count its frequency in the leaf node and then divide by the total number of instances (possibly with a Laplace estimator). This will give you a proper probability distribution that sums to one over the class distribution. Consequently, it would be trivial to use a cost matrix on it to have the classifier be cost-sensitive (e.g., minimize expected cost).

So, for instance, the tree path wrong\_fragment -> hot -> src\_bytes -> serror\_rate -> srcbytes < 0 leads to a leaf node that has mostly nmap class classes (73 out of 78 instances). Since they are not all nmap classes, other classes like normal or teardrop may be represented in this leaf node. One can then calculate their probabilities, and use the cost matrix as previously described to predict the class that minimizes the expected cost. (While the nmap class is by far the most probable in this instance, the cost of a false positive may be exceptionally high.)

### Alternative Approaches To The Cost Matrix

This is a fairly straightforward way to make decision tree classifiers cost sensitive. However, another way presents perhaps a simpler, though arguably more ad hoc way: increase the number of classes which you want to make more costly to misclassify or fail to classify. For instance, in a two class situation, if you want to make the classifier predict “no” classes more often (with higher false alarms), duplicate a portion of the instances that have class “no” assigned to them. This will bias the classifier towards making more “no” classifications without needing a cost matrix. A similar effect can also be achieved by weighing the instances differently. However, the cost matrix seems like a more flexible and timely way to approach this problem of making classifiers cost-sensitive, as it does not require any retraining. Furthermore, the cost matrix can be more easily understood, especially if the various costs are already known, since it is all contained in one simple matrix.

## BayesNet classifier



**Figure 3 Bayesian network**

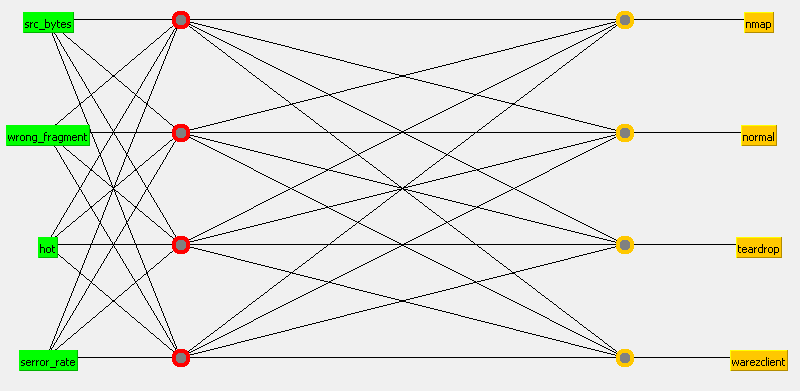
As previously mentioned, the BayesNet (and other simpler variants, like NaïveBayes) does not need to be tweaked to allow them to work with the cost matrix. They are designed from the very start to explicitly generate class probability distributions; in fact, the above Bayesian network represents a joint probability distribution, where some of the attributes are dependent upon other attributes, e.g., hot depends on both class and src\_bytes attributes.

It was able to correctly classify 33682 (99.5537%) of the test instances, misclassifying only 151 (0.4463%). This is a fairly impressive statistic, especially in light of the seeming complexity of the original data set.

## Feed-forward neural network w/back-propagation

Next, we consider how to modify the traditional neural network to output class probability distributions instead of a single class value, i.e., instead of using a “winner-takes-all” scheme on the activation level on the output nodes, treat the output nodes as representing a class probability distribution. Technically, as in the J48 decision tree inducer, they do not represent probability distributions in the same sense that the BayesNet rigorously does, but it is a reasonable approximation. Furthermore, to have the output nodes sum to one (to make it a more proper probability distribution), one can treat the output neurons as a vector and normalize it to unit distance.

After performing the above tweaks, it should be straightforward to use a cost matrix with it to minimize expect costs, rather than simply choosing the most probable outcome.



**Figure 4 Feed-forward neural network w/backprop**

Figure 4 shows our neural network’s topology. Specifically, it is a multi-layer (with one hidden layer) perception. It took awhile to stabilize, and is also relatively worse than the results from the BayesNet. Specifically, it correctly classified 99.0837% of the instances, misclassifying the remaining 0.9163%. This is respectable, perhaps, but not competitive with the BayesNet or decision tree inducer. In fact, it actually fairs worse than 1R, which is a tree structure that branches off from only one attribute.

## 1R classifier

src\_bytes:

< 27.5 -> normal

< 28.5 -> teardrop

< 333.5 -> normal

< 334.5 -> warezclient

< 831.5 -> normal

< 832.5 -> warezclient

< 1220.5 -> normal

< 1222.5 -> warezclient

< 1223.5 -> normal

< 1225.5 -> warezclient

< 1236.5 -> normal

< 1239.5 -> warezclient

< 7028.5 -> normal

< 7061.0 -> warezclient

< 2347338.5 -> normal

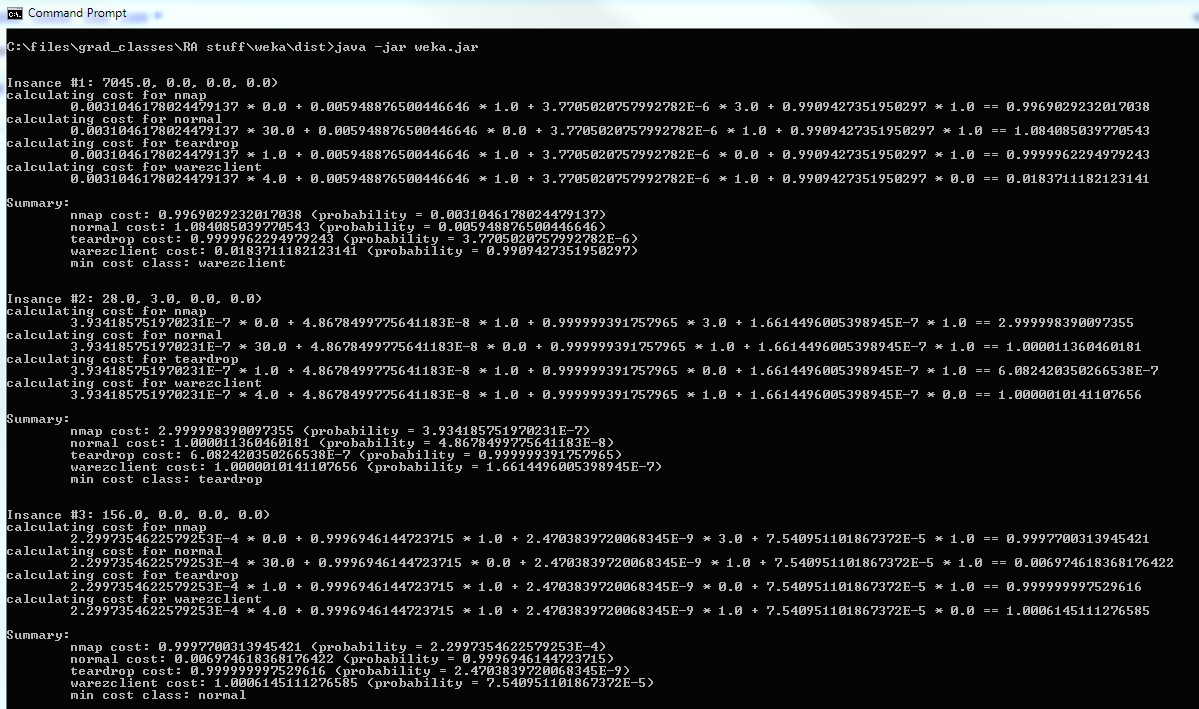
>= 2347338.5 -> warezclient

1R, remarkably, achieves 99.1488% accuracy, which is not much worse than BayesNet. Indeed, the src\_bytes attribute by itself is sufficient to classify the vast majority of instances. It just goes to show you that many data sets that seem extremely complicated turn out to be fairly one-dimensional.

To tune 1R to generate a class probability distribution instead of a simple class prediction, we note that 1R works by selecting the attribute that causes the least amount of error in the training set. Therefore, a simple tweak would be to record the various class frequencies in each of the branches from src\_bytes, and use those frequencies to generate the probabilities. For example, if src\_bytes < 27.5, then "normal" is the majority class. Suppose there are 100 instances that reach this leaf node, and 60 of them are classified as normal, 30 of them are classified as warezclient, and 10 of them are classified as teardrop. Therefore, the class probability distribution is, simply enough, 0.6, 0.3, and 0.1. As such, it too can be tweaked to work with a cost matrix.

## Notes on Implementation

We were able to implement a basic version of the min expected cost classifier using a cost matrix, as described in this paper. See the README.TXT for more information. However, here is some output generated by the program.



## Final thoughts

We ran this data set again on many other machine learning classifiers, unfortunately we did not have time to get the support vector machine classifiers to work in time. We think much more work can be done in this area to help businesses reduce the costs associated with responding to false-positives; however, due to limitations of time and resources, this work could not be accomplished in time for the submission of this paper.

**References**

1. Title Photo: <http://www.fotosearch.com/photos-images/network-security_3.html>

2. Data Mining: Practical Machine Learning Tools and Techniques by Ian H. Witten & Eibe Frank page 161, section 5.7