Working Paper Series
ISSN 1170-487X

Applying Propositional Learning Algorithms to Multi-Instance Data

Eibe Frank and Xin Xu

Working Paper: 06/03
June 2003

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Applying propositional learning algorithms to multi-instance data

Eibe Frank and Xin Xu

Department of Computer Science
University of Waikato
Hamilton, New Zealand
{eibe, xx5}@cs.waikato.ac.nz

June 17, 2003

Abstract

Multi-instance learning is commonly tackled using special-purpose algorithms. Development of these algorithms has started because early experiments with standard propositional learners have failed to produce satisfactory results on multi-instance data—more specifically, the Musk data. In this paper we present evidence that this is not necessarily the case. We introduce a simple wrapper for applying standard propositional learners to multi-instance problems and present empirical results for the Musk data that are competitive with genuine multi-instance algorithms. The key features of our new wrapper technique are: (1) it discards the standard multi-instance assumption that there is some inherent difference between positive and negative bags, and (2) it introduces weights to treat instances from different bags differently. We show that these two modifications are essential for producing good results on the Musk benchmark datasets.

1 Introduction

Despite a lack of fielded applications, multi-instance learning continues to attract a good deal of interest in the machine learning community. A possible reason is the potential for applications in the life sciences, as exemplified by the drug activity problem that motivated the first in-depth investigation of this type of learning problem [2]. Drug activity prediction can be viewed as a supervised classification problem where each molecule is an example and the class of the example reflects whether the molecule is active or not. Each example has only one class label but consists of a bag of instances. An instance represents a possible conformation of the molecule. Propositional learners cannot be applied directly to this type of learning task because they require a class label for every individual instance. A natural way of circumventing this problem is to give the instances their bags' label. At classification time a bag is classified as positive if at least one of its instances is classified as positive (i.e. a molecule is classified as active if at least one of its conformations is classified as active), and

negative otherwise [2]. However, results based on this wrapper method were disappointing and spawned research into multi-instance learning algorithms that are specifically designed to deal with bags of instances.

In this paper we present results suggesting that it may be premature to abandon standard propositional learning algorithms when it comes to tackling multi-instance problems. We present a simple wrapper that, in conjunction with appropriate propositional learning algorithms, achieves high accuracy on the Musk benchmark datasets. Like the wrapper described above our method assigns every instance the class label of the bag that it pertains to. However, it differs in two crucial aspects: (1) at training time, instances are assigned a weight inversely proportional to the size of the bag that they belong to, and (2) at prediction time, the class probability for a bag is estimated by averaging the class probabilities assigned to the individual instances in the bag. This method does not require any modification of the underlying propositional learner as long as it generates class probability estimates and can deal with instance weights. We present empirical results for a collection of well-known propositional learners that are competitive with published results for multi-instance learners. Note that this procedure does not exploit the original multi-instance assumption, which states that a bag is positive if and only if at least one of its instances is positive.

The paper is structured as follows. In Section 2 we describe our method in more detail and motivate the design decisions we made. Section 3 interprets the procedure in terms of the assumptions that it makes. Section 4 illustrates the intuition behind it based on an artificial example problem. In Section 5 we present empirical results for the Musk data based on applying our procedure in conjunction with a variety of well-known propositional learning algorithms. In Section 6 we discuss related work on multi-instance learning and in Section 7 we summarize our results.

2 A simple wrapper for multi-instance learning

In standard propositional classification problems each training example consists of a single instance (a fixed-length vector of attribute values) and a corresponding class label. The set of all possible instances is called the instance space and the learning algorithm generates a mapping from this space to the set of possible class labels. Usually, the aim is to find a mapping minimizing the number of misclassifications on future data that has not been used for training.

Multi-instance learning is a generalization of this where each example consists a bag of instances instead of only one, and the number of instances can vary from one example to the next. However, there is still only one class label for each example. The difficulty of multi-instance learning arises from the fact that it is unclear which of the instances are responsible for the bag's class label. The approach we take in this paper is to assume that all instances contribute equally and independently to the bag's label. This is a departure from the standard approach, where the label of a bag is assumed to be determined by the presence or absence of specific "key" instances. Whereas the latter approach requires a sophisticated method for identifying the key instances from the training data, the former allows us to apply standard propositional learning algorithms by breaking a bag up into its individual instances and labeling

each instance with its bag's label. The only caveat is that some examples have more instances than others, yet they should receive equal weight in total. This problem can be rectified by weighting the instances derived from a particular bag so that the total weight for the bag is one. If there are n instances in a bag we give each instance the weight $1/n$ (because we assume that the instances contribute equally to the example's class label we give them the same weight). This weighting scheme ascertains that the learning algorithm will not be biased towards particular examples (i.e. those with more instances). The experimental results in Section 5 show that this is indeed important for obtaining accurate classifications.

The question remains as to what to do at prediction time in order to generate a classification for a new bag of instances. Because we assume that the instances in a bag contribute independently to its class label the first step is to filter each instance through the classification model built at training time to obtain a class probability estimate for each of the possible classes. Then, in the second step, these class probability estimates are combined to form a prediction for the bag as a whole. Because we assume that all instances contribute equally to the bag's class label, we simply average the class probability estimates obtained from the individual instances, giving each probability estimate equal weight. A side effect of this is that the sum of a bag's class probability estimates (taken over all the classes) will be one.

This method of dealing with multiple instances is inspired by the way in which the single-instance decision tree inducer C4.5 [10] deals with missing attribute values. When C4.5 encounters an instance with a missing attribute value at a particular node, the instance is cloned, once for each branch, and each clone gets a weight proportional to the number of instances with known values going down its branch. Thus the original example, consisting of only one instance, is effectively replaced by a bag of instances. The only difference to our approach is that C4.5 creates the bags on the fly, using knowledge inferred from the data, whereas in our setting the bags are given prior to induction time (and all the instances in a bag receive equal weight). Note also that C4.5 uses the same method at prediction time: the class probability estimates obtained at the leaf nodes are combined using the weights of the cloned instances arriving at those leaves. Again, the only difference is that the bag is created on the fly and that the instances' weights are usually not equal. The underlying motivation is the same: we are not sure which instantiation of the example is the correct one and consequently compute the "expected" class probability.

3 Interpretation

In this section we interpret the above wrapper method in terms of the underlying generative model that it assumes. Let b be a bag of instances. Then we compute the probability of class c given that bag as follows:

$$Pr(c|b) = E_X(Pr(c|x)|b) = \int_X Pr(c|x)Pr(x|b), \tag{1}$$

In other words, we marginalize x out and assume that the probability of class c is conditionally independent of b given x (i.e. knowledge of b does not affect the class probability for a particular x). Given a concrete bag b of size n,

the estimated value of $Pr(x|b)$ for each instance in the bag is $1/n$, and zero otherwise, and $Pr(c|b)$ becomes the average of the class probability estimates for the instances in the bag.

The question remains as to how we can estimate $Pr(c|x)$. If we had a single-instance version of the data available for training, we could estimate this function using a propositional learner (e.g. a decision tree inducer) and then apply Equation 1 for the bags observed at prediction time. However, in a true multi-instance problem the training data is also in bag form. If we had a class label for each individual instance in a bag, we could still use a standard propositional learning algorithm to estimate $Pr(c|x)$ by giving each instance the weight $1/n$. Assuming an instance-level loss function $Loss(\beta, C)$ for a model parameterized by β (e.g. the negative log-likelihood), the learner would then minimize the following expected loss:

$$E_B\left[E_{X|B}[E_C(Loss(\beta, C)|X)|B]\right] = \sum_i \frac{1}{N} \sum_j \frac{1}{n_i} E_C(Loss(\beta, C)|x_{ij}), \quad (2)$$

where N is the number of bags and n_i the number of instances in bag i.

However, in typical multi-instance problems there is only one label for the whole bag. Consequently our wrapper method performs a further simplification of the problem: it assigns the bag label to each instance in the bag and uses this as the training data for the propositional learner. This is a heuristic solution and does not enable the propositional learner to recover the true function $Pr(c|x)$. It will necessarily produce a biased estimate of the class probabilities. However, as we will see in the next section, there are special cases where the correct *decision boundary* can be recovered (i.e. the classification performance is not affected), and it is well-known that unbiased class probability estimates are not necessary to obtain accurate classifications. Intuitively, this heuristic will work well if the true class probabilities $Pr(c|x)$ are "similar" for all the instances in a bag (and the above generative model is correct). As our experimental results in Section 5 show, the method generates very accurate classifiers for the Musk benchmark datasets.

4 An artificial example domain

To illustrate the behavior of our wrapper technique we consider an artificial domain with two attributes. More specifically, we created bags of instances by defining rectangular regions and sampling instances from within each region. First, we generated coordinates for the centroids of the rectangles according to a uniform distribution with a range of $[-5, 5]$ for each of the two dimensions. The size of a rectangle in each dimension was chosen from 2 to 6 with equal probability. Each rectangle was used to create a bag of instances. To this end we sampled n instances from within a rectangle according to a uniform distribution. The value of n was chosen from 1 to 20 with equal probability.

It remains the question as to how we generate the class label for a bag. Our generative model assumes that the class probability of a bag is the average class probability of the instances within it, and this is what we used to generate the class labels for the bags. The instance-level class probability was defined by the

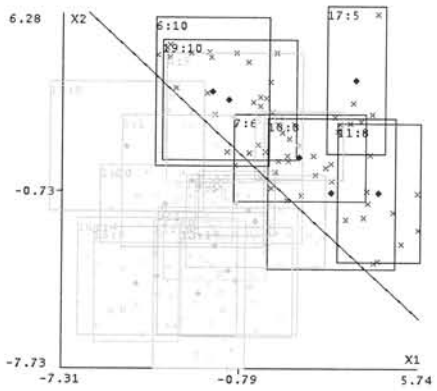

Figure 1: An artificial dataset with 20 bags

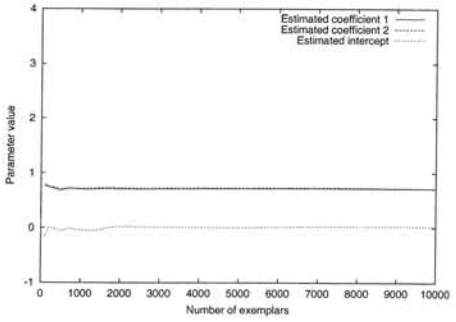

Figure 2: Estimated parameters

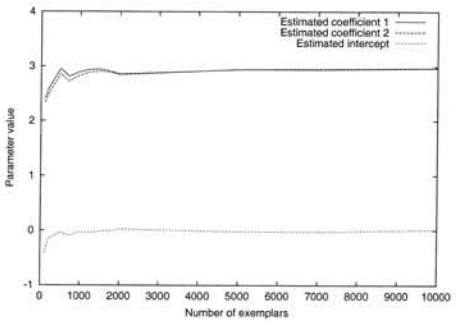

Figure 3: "Unmasked" case

following linear logistic model:

$$Pr(y = 1|x_1, x_2) = \frac{1}{1 + e^{-3x_1 - 3x_2}}$$

Figure 1 shows a dataset with 20 bags that was generated according to this model. The black line in the middle is the instance-level decision boundary (i.e. where $Pr(y = 1|x_1, x_2) = 0.5$) and the sub-space on the right side has instances with higher probability to be positive. A rectangle indicates the region used to sample points for the corresponding bag (and a diamond indicates its centroid). The top-left corner of each rectangle shows the bag index, followed by the number of instances in the bag. Bags in gray belong to class "negative" and bags in black to class "positive". Note that bags can be on the "wrong" side of the instance-level decision boundary because each bag was labeled by flipping a coin based on the average class probability of the instances in it.

Because the instance-level probabilities are given by a linear logistic model, we expect logistic regression (based on maximum likelihood) to be a good learning technique for this problem. Consequently we applied this technique in conjunction with our wrapper method by giving each instance its bag's class label and weighting the instances so that the total weight of all the bags was the same. Figure 2 plots the estimated coefficients of the linear model as the number of

training bags increases. The plot shows that the coefficients do not converge to their "true" value (which is three for both coefficients and zero for the intercept). However, they do converge to scaled versions of these parameters. This means that the wrapper method finds the correct instance-level decision boundary although the estimated value of $Pr(y = 1|x_1, x_2)$ is systematically biased (basically, the function has been "flattened"). Bag-level classification performance is not affected because flattening $Pr(y = 1|x_1, x_2)$ has no influence on whether a bag is classified as positive or not. In fact, we found that for this problem the wrapper achieves the same bag-level accuracy as the true model on a collection of 10,000 independent test bags if trained on approximately 150 bags or more.

The bias in the probability estimates is introduced because the true instance-level class labels have been "masked": every instance receives its bag's label. However, because we have full control over the data generation process, we can give each instance a class label according to $Pr(y = 1|x_1, x_2)$ and run the wrapper method on this "unmasked" data. Figure 3 shows the resulting estimates for the coefficients of the linear model. Note that the same weighting scheme was used and the same bags, only the instances' class labels were generated differently. In this case, the method converges to the correct estimates.

Of course, in practice we only have one class label for each bag, and this will result in biased class probability estimates. In fact, the wrapper will generally not find the correct decision boundary either (even if we choose the correct propositional learner as we did in this case). The above problem was carefully constructed to be totally symmetric. For example, changing the slope of the linear model means that the wrapper method will no longer identify it correctly. However, it will find one that is very similar to it, resulting in a relatively small loss in classification accuracy. This property appears to be sufficient to achieve good performance on the Musk data as we will see in the next section.

5 Experimental results

In the following we present empirical results for the Musk drug activity data [2] using our wrapper technique in conjunction with a number of well-known techniques for propositional learning. Table 1 summarizes some properties of the two Musk problems. Musk 1 is the smaller of the two datasets, mainly because it contains a smaller number of instances per bag. We expect that Musk 2 presents a more challenging problem for the wrapper method because the bags are larger and therefore more likely to exhibit instances whose "true" class label is not consistent with the bag's class label. However, the default accuracy is greater for Musk 2: predicting the majority class results in 61.76% correct classifications. For the Musk 1 data the default accuracy is only 51.09%.

Table 2 shows accuracy estimates obtained using a variety of learning algorithms implemented in the Weka workbench [11]. All estimates were obtained using stratified 10-fold cross-validation repeated 10 times. The cross-validation runs were performed at the bag level. For each run the order of the bags was randomized and the data split into 10 subsets so that each subset had approximately the same number of bags and the same class distribution. The same

	Musk 1	Musk 2
Number of bags	92	102
Number of attributes	166	166
Number of instances	476	6598
Number of positive bags	47	39
Number of negative bags	45	63
Average bag size	5.17	64.69
Median bag size	4	12
Minimum bag size	2	1
Maximum bag size	40	1044

Table 1: Properties of the Musk 1 and Musk 2 datasets.

cross-validation runs were used for each learning scheme. The standard deviation of the 10 cross-validation estimates is also shown in Table 2. The results are sorted according to the average accuracy on the two datasets. If not explicitly stated otherwise, all learning algorithms were applied with their default parameter settings in order to avoid introducing bias by parameter tuning.

Some of the learning schemes are sensitive to the *absolute* value of the instance weights. For example, C4.5 stops splitting if the total weight of the instances at a node is smaller than four.[1] In our wrapper method this would happen very early in the tree construction process if all bags received a total weight of one each. Consequently we multiplied the weight of each instance by a constant factor before applying the learning scheme so that the total weight of all the instances in the data was the same as the number of instances, i.e. the weight w_{ij} for instance i of bag j was rescaled into a new weight w'_{ij} as follows:

$$w'_{ij} = \frac{m}{N} \times w_{ij}, \tag{3}$$

where m is the total number of instances in the dataset (across all bags), and N the total number of bags. Note that maximum likelihood methods (e.g. ordinary logistic regression and fully grown decision trees) are not affected by this weight rescaling step. However, techniques that perform some form of regularization involving a trade-off between the complexity of the model and the performance on the training data (e.g. support vector machines and Bayesian techniques) are.

The best performance was obtained from a support vector machine with a Gaussian kernel ("RBF Support Vector Machine"), trained using the sequential minimal optimization algorithm [8]. This algorithm is controlled by two parameters: the complexity parameter C (bounding the coefficient of each support vector), and the width of the Gaussian kernel. Weka's defaults for these parameters are 1 and 0.01 respectively, and those values were left unchanged to generate the result.

Support vector machines do not produce class probability estimates but our wrapper method requires these. To obtain them we fit a simple linear logistic model to the output of the support vector classifier by maximizing the likelihood of the training data. Platt [9] recommends this procedure in conjunction

[1] If the default settings of its parameters are used.

	Musk 1	Musk 2
RBF Support Vector Machine	89.13±1.15	87.16±2.14
Adaboost.M1 with C4.5	85.65±1.90	83.63±2.77
Bagging with PART	87.72±2.05	81.27±1.56
Adaboost.M1 with PART	85.87±2.61	82.94±1.74
Bagging with C4.5	86.41±2.13	79.61±2.44
Linear Support Vector Machine	84.35±2.36	80.69±1.73
PART	81.09±1.86	81.47±2.47
C4.5	84.13±3.21	78.24±2.39
Linear Logistic Regression	79.78±3.13	81.96±2.13
Nearest Neighbor	82.61±2.56	76.27±1.89
Naive Bayes	76.20±2.08	76.37±1.69

Table 2: Accuracy estimates from 10 runs of stratified 10-fold cross-validation. The standard deviation of the 10 estimates is also shown.

with a cross-validation step to generate the training data for the logistic model. However, we found that the cross-validation step was not necessary to obtain accurate results. Another question is how the support vector machine can take into account the instance weights that our wrapper method generates. This is achieved by multiplying the bound C for a particular instance with its weight.

For reference, Table 2 also shows the accuracy for a linear support vector classifier—based on the dot product kernel—applied in the same way. The resulting drop in accuracy indicates that a non-linear estimator is essential to obtain accurate results on this data. This is also reflected in the comparably bad result for linear logistic regression (also shown in Table 2), where the weights are taken into account by maximizing the weighted likelihood of the training data.

Two other simple classifiers are naive Bayes and one nearest neighbor, and both did not perform as well as the more sophisticated schemes (see Table 2). Note that naive Bayes does not strictly fit into our framework because it does not minimize an instance-level loss function based on posterior probabilities. Also, it assumes that the attributes are independent and normally distributed given the class, and both assumptions are likely to be incorrect in the Musk problems. A problem with one nearest neighbor in this setting is that it cannot take instance weights into account and that it produces very discrete probability estimates (either 0 or 1). However, accuracy improves only slightly if more than one neighbor is used (not shown).

Table 2 also shows results for the decision tree learner C4.5 (based on the implementation in Weka), and for the decision list inducer PART [3]. Both produce class probability estimates and can deal with instance weights. The accuracy of these two methods is comparable. The former is slightly more accurate than the latter on the Musk 1 data while the latter wins on Musk 2. However, in both cases the accuracy is not as high as for the non-linear support vector machine.

A possible explanation for this is that both methods can exhibit high variance, especially in large-dimensional numeric spaces like those we are considering here. A popular way to alleviate this problem is to use ensemble methods,

	Musk 1	Musk 2
Bagging with PART	90.22±2.11	87.16±1.42
Bagging with C4.5	90.98±2.51	85.00±2.74
AdaBoost.M1 with C4.5	89.24±1.66	85.49±2.73
AdaBoost.M1 with PART	89.78±2.30	84.02±1.79
PART	84.78±2.51	87.06±2.16
C4.5	85.43±2.95	85.69±1.86

Table 3: Accuracy estimates and standard deviations for data discretized using equal-frequency discretization (10 runs of stratified 10-fold cross-validation).

notably bagging and boosting. The latter often also reduces bias. We tried both—in the case of bagging with unpruned decision trees and lists because this often helps to further improve performance. Before discussing the results we briefly explain how these techniques can generate probability estimates based on weighted instances.

Bagging [1] can produce probability estimates simply by averaging the probabilities obtained from the individual ensemble members. Instance weights can be incorporated by modifying the sampling process used to generate the bootstrap samples that are the basis for the committee members: instead of using equal selection probabilities when sampling instances with replacement, the probabilities can be made proportional to the corresponding instances' weights. If this is done, the distribution of instances in the bootstrap sample will reflect the instance weights.

The boosting method we use, AdaBoost.M1 [5], was not originally designed for producing class probability estimates. However, as Friedman *et al.* [6] show, it can be interpreted as minimizing a loss function very similar to the binomial likelihood commonly used for probability estimation. AdaBoost.M1 classifies an instance as "positive" if the weighted vote of the committee members voting "positive" for this instance is larger than the weighted vote for "negative". The weights are the classifier weights that AdaBoost.M1 generates at training time. Let s_{pos} be the sum of the weights for the classifiers voting "positive", and s_{neg} be the corresponding sum for the negative votes. Then the probability of the instance belonging to the positive class is estimated by:

$$\hat{Pr}(pos|x) = \frac{e^{s_{pos}}}{e^{s_{pos}} + e^{s_{neg}}} \tag{4}$$

Instance weights can be incorporated into AdaBoost.M1 in a very natural way. The standard formulation of the algorithm starts with uniform weights. These can simply be replaced by the weights given by our wrapper method.

Table 2 shows results based on 10 iterations for both bagging and boosting. Bagging increases accuracy on Musk 1, especially for PART. However, there is virtually no change on Musk 2. Boosting, on the other hand, improves the accuracy on both problems. Increasing the number of boosting and bagging iterations leads to further small improvements (not shown) but the accuracy scores fall short of those obtained using the non-linear support vector machine.

In the experiments discussed so far we have not attempted to modify the attribute values of the input data before applying a learning algorithm. However,

the Musk problems exhibit a large number of numeric attributes and global discretization is a technique that often helps decision tree and list inducers if this is the case. We experimented with different discretization methods and found that simple equal-width discretization produced the best results, where each numeric attribute was split into 10 intervals of equal size (10 is the default in Weka), and converted into nine binary attributes by encoding the resulting split points [4].

Table 3 shows the estimated accuracies for C4.5 and PART, and their bagged and boosted versions (again with 10 iterations), on the discretized data. Note that the discretization process was repeated from scratch for every individual training set occuring in the 10 runs of 10-fold cross-validation. Compared to the corresponding results in Table 2, the change in performance is quite striking. Discretization uniformly increases accuracy. For C4.5, PART, and their bagged versions, the increase is more than five percent on the Musk 2 data. On Musk 1, bagging and boosting now achieve accuracies around 90%. In fact, bagged PART performs slightly better than the non-linear support vector machine from Table 2. Note that increasing the number bagging and boosting iterations increases the accuracy only slightly (not shown).

So far we have not presented any evidence that the weighting scheme in our wrapper is important. To demonstrate its effectiveness we modified the procedure to give every instance a weight of one and ran it with C4.5 (without discretization) on the two Musk problems. On Musk 1 the estimated accuracy dropped from 84.13% to 77.28%, and on Musk 2 from 78.24% to 70.78%. This demonstrates that the weighting scheme is indeed important to obtain good results.

Another question is whether the wrapper uses an appropriate method for prediction. In the paper that first introduced the Musk problems [2], propositional learners (more specifically, C4.5 and a multi-layer perceptron) were applied by giving each instance a weight of one at training time, and classifying a bag as positive if at least one of its instances was classified as positive, and negative otherwise. We repeated this experiment using ten runs of ten-fold cross-validation and obtained 71.63% accuracy on Musk 1 and 58.73% on Musk 2. This is comparable to the results published in [2], which were 68.5% and 58.8% respectively. Introducing instance weights at training time improved the accuracy to 77.17% and 65.78% respectively. However, this is worse than the above results obtained by probability averaging at prediction time.

6 Related work

Most closely related to the results presented here is the work by Gärtner *et al.* [7]. Their paper also contains an extensive list of special-purpose multi-instance algorithms that can be found in the literature. Gärtner *et al.* use a special set kernel for support vector machines to tackle the Musk problems. The set kernel simply takes all pairwise dot products between instances from two different bags to compute a similarity score for the two bags. They present a proof showing that problems for which the original multi-instance assumption holds can be learned using this set kernel if the training data is fit closely enough. However the learner never actually exploits the assumption and treats both classes in a symmetric fashion (as we do in this paper). Using a Gaussian kernel on top of

the (normalized) set kernel, Gärtner *et al.* obtain 86.4% accuracy on Musk 1 and 88% accuracy on Musk 2.[2] This is very similar to the results we obtained using our wrapper technique and a support vector machine with a Gaussian kernel (89.13% and 87.16% respectively), demonstrating that it is not necessary to resort to a special kernel to tackle the Musk problems successfully with this type of classifier. In addition our experiments show that similar performance can be obtained with ensemble methods in conjunction with discretization.

Gärtner *et al.* also show that the Musk problems can be converted into single-instance datasets by deriving a single instance from each bag based on some form of summary statistics. They present experimental results obtained by taking the minimum and maximum attribute values for each bag to form a single instance (so that the corresponding single-instance dataset has twice as many attributes as the original multi-instance dataset). In conjunction with a standard support vector machine and a polynomial kernel of order five this resulted in an estimated accuracy of 91.6% on Musk 1 and 86.3% on Musk 2. Hence this alternative method of propositionalizing multi-instance problems is competitive with the procedure presented in this paper, at least on the Musk data. However, it is conceivable that the particular summary statistics used may need to be adapted to the specific learning problem at hand, potentially requiring some work from the end user. This is not necessary with our wrapper technique.

7 Conclusions

We have presented a new wrapper approach for tackling multi-instance problems with propositional learning algorithms and showed that it produces competitive results on the Musk benchmark problems (if used in conjunction with an appropriate propositional base learner). The wrapper can be employed with any base learner that can (a) deal with instance weights and (b) produce probability estimates, and most algorithms fall into this category.

We have shown that the wrapper can be viewed as a heuristic approach for minimizing an expected instance-level loss, where the expectation is taken over the bags, if we assume a certain generative model. However, this approach leads to biased probability estimates. In future work we will attempt to create a bag-level loss function that can be minimized directly.

Acknowledgments

Many thanks to Nils Weidmann, Bernhard Pfahringer, and Mark Hall for their comments. This research was supported by Marsden Grant 01-UOW-019.

References

[1] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[2] This was estimated using leave 10 out, which gives approximately the same results as 10-fold cross-validation on this data.

[2] T.G. Dietterich, R.H. Lathrop, and T. Lozano-Pérez. Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1-2):31–71, 1997.

[3] E. Frank and I.H. Witten. Generating accurate rule sets without global optimization. In *Proc Int Conf on Machine Learning*, pages 144–151. Morgan Kaufmann, 1998.

[4] E. Frank and I.H. Witten. Making better use of global discretization. In *Proc Int Conf on Machine Learning*, pages 115–123. Morgan Kaufmann, 1999.

[5] Y. Freund and R.E. Schapire. Experiments with a new boosting algorithm. In *Proc Int Conf on Machine Learning*, pages 148–156. Morgan Kauffman, 1996.

[6] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: A statistical view of boosting (with discussion). *Annals of Statistics*, 28:307–337, 2000.

[7] T. Gärtner, P.A. Flach, A. Kowalczyk, and A.J. Smola. Multi-instance kernels. In *Proc Int Conf on Machine Learning*, pages 179–186. Morgan Kaufmann, 2002.

[8] J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods—Support Vector Learning*. MIT Press, 1998.

[9] J. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In A. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans, editors, *Advances in Large Margin Classifiers*. MIT Press, 1999.

[10] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[11] I.H. Witten and E. Frank. *Data Mining: practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, 1999.