Ensembles of Biased Classifiers

Rinat Khoussainov RINAT@UCD.IE
Andreas Heß ANDREAS.HESS@UCD.IE
Nicholas Kushmerick NICK@UCD.IE
School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland

Abstract

We propose a novel ensemble learning algorithm called Triskel, which has two interesting features. First, Triskel learns an ensemble of classifiers, each biased to have high precision on instances from a single class (as opposed to, for example, boosting, where the ensemble members are biased to maximise accuracy over a subset of instances from all classes). Second, the ensemble members' voting weights are assigned so that certain pairs of biased classifiers outweigh the rest of the ensemble, if their predictions agree. Our experiments demonstrate that Triskel often outperforms boosting, in terms of both accuracy and training time. We also present an ROC analysis, which shows that Triskel's iterative structure corresponds to a sequence of nested ROC spaces. The analysis predicts that Triskel works best when there are concavities in the ROC curves; this prediction agrees with our empirical results.

1. Introduction

Ensemble techniques have been demonstrated to be an effective way to reduce the error of a base learner across a wide variety of tasks. The basic idea is to vote together the predictions of a set of classifiers that have been trained slightly differently for the same task. There is a strong body of theory explaining why ensemble techniques work.

Nevertheless, it is straightforward to construct learning tasks that confound existing ensemble techniques. For example, consider a synthetic "layered cake" binary learning task shown in Figure 1. The Support Vector Machine algorithm (Vapnik, 1982) with a linear kernel learns a decision surface with a large error. Boosting SVM does not help: at each iteration, the classifier is unable to stop making mistakes on the middle two regions; these regions then get even more weight on the next iteration, and eventually boosting gives up because it can not find a classifier with error less than 0.5. However, Figure 1(b) shows that ensemble methods are in principle well suited to this task: when combined with a simple unweighted vote, the set of three linear decision surfaces yields an ensemble with zero error.

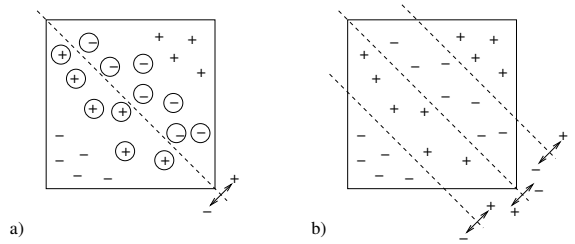

Figure 1. The "layered cake" task: (a) decision surface learned by a single SVM with linear kernel (circled instances are classified incorrectly); (b) an ensemble of three linear SVMs that has zero training error when combined with a simple majority vote.

Motivated by this example, we propose a novel ensemble learning algorithm called Triskel, which has two interesting features. First, Triskel learns an ensemble of classifiers that are biased to have high precision for one particular class. For example, in Figure 1(b), one of the "outer" classifiers is biased to (i.e. has high precision, albeit mediocre recall, for) the positive class, and the other classifier is biased to the negative class. In contrast, most existing ensemble techniques use ensemble members that are biased to focus on various regions of the instance space. For example, each round of boosting focuses on instances that were misclassified in previous rounds; and bagging involves hiding some of the training data from each ensemble member.

The second interesting feature is the manner in which Triskel assigns weights to the ensemble members. Triskel uses weighted voting like most ensemble methods, but the weights are assigned so that certain pairs of biased classifiers outweigh the rest of the ensemble, if their predictions agree. For example, in Figure 1(b), the two "outer" classifiers dominate the vote if they agree. If they disagree, the "inner" classifier casts the deciding vote. Our algorithm is

Appearing in *Proceedings of the 22^{nd} International Conference on Machine Learning*, Bonn, Germany, 2005.

named after a Celtic spiral design with three branches. In its simplest incarnation, Triskel uses an ensemble of three classifiers: one classifier biased to the positive class, one classifier biased to the negative class, and one unbiased classifier to make predictions when the others disagree.

We make the following contributions. First, we motivate and describe Triskel, our novel approach to ensemble learning, and describe various ways to construct the biased classifiers on which Triskel relies. Second, we evaluate Triskel on a variety of real-world tasks and demonstrate that our method often outperforms boosting, in terms of both accuracy and training time. Finally, we present a qualitative analysis of the algorithm using ROC graphs and discuss how Triskel represents a middle ground between covering algorithms and ensemble techniques such as boosting.

2. Related work

Dietterich (2000) discusses three fundamental ways in which an ensemble can achieve better performance: statistical, computational and representational. The statistical analysis starts with the observation that any learning algorithm tries to find a hypothesis that has a good accuracy on the training data. When the amount of the training data is small, there may be many different hypotheses that all give the same accuracy on the training data. Constructing an ensemble out of all these accurate classifiers can allow the algorithm to reduce the risk of choosing the wrong hypothesis. The computational argument is that many learning algorithms perform some sort of local search in the hypotheses space that may get stuck in a local optima. Examples include gradient-based search in neural networks and greedy search in decision trees. An ensemble constructed by running the local search from multiple different starting points may result in a better approximation to the true hypothesis. Finally, the representational analysis follows from the fact that a learning algorithm may not be capable of representing the true function either because it is outside of its hypothesis space or because it does not have sufficient training data to explore all of its hypothesis space to find it (e.g. the classifier would stop searching once it finds a hypothesis that fits the training data). By combining several different hypotheses (e.g. using a weighted sum) it may be possible to expand the space of representable functions.

Perhaps the best-known ensemble methods are bagging (Breiman, 1996) and boosting (Shapire, 1990; Freund, 1995), in particular AdaBoost (Freund & Shapire, 1997). Bagging generates different training sets by drawing randomly with replacement from the original data set. The classifiers' decisions are combined using the majority vote. AdaBoost performs several learning iterations on the same training set. However, in each iteration it adjusts the weights of the training instances to emphasise the examples that were misclassified by the last learned classifier. The decisions of the classifiers in the ensemble are combined using weighted voting, where the weights depend on the error of the classifier on the training set. Another example of error-based instance weighting is (Breiman, 1999).

Since bagging constructs its training sets (and, hence, its ensemble members) independently from the others, it mainly addresses the statistical and, to a lesser extent, computational problems. AdaBoost constructs each new hypothesis to eliminate remaining errors and, thus, is directly trying to address the representational problem. It has been shown that by focusing on incorrectly classified instances AdaBoost minimises a particular error function of the ensemble on the training data called the negative exponential of the margin (Freund & Shapire, 1997; Dietterich, 2000).

3. The Triskel Algorithm

One of the problems with AdaBoost is that in each subsequent iteration, the base learner is presented with more and more difficult problems. The redistribution of instance weights is based on the errors of the last learned hypothesis on the training data. Over multiple iterations, this can result in weight distributions that are too complex for the base learner to handle. For example, suppose we would like to boost an SVM with a linear kernel on the data set shown in Figure 1(a). The diagram shows the decision surface that an SVM would learn on this data in the first iteration. We can see that the distribution of errors is such that a linear decision surface will do a poor job on such task. Specifically, the weight distribution will switch in this case between inner and outer instances after each boosting iteration without improvements to the resulting ensemble accuracy.

Nonetheless, the example in Figure 1(a) can be handled perfectly by an ensemble of three linear separators shown in Figure 1(b) combined using a majority vote. One classifier separates part of positive instances from the rest of positives and negatives, one classifier separates part of negative instances, and the remaining classifier handles the instances where the first two classifiers disagree.

An analogy with covering (e.g. (Fürnkranz, 1999)) can be drawn. A covering algorithm tries to identify rules with high precision that cover a large number of (ideally, uniformly labelled) training examples. These instances are then removed from the training set, and the learning continues until all examples are covered. In our algorithm, one classifier covers the data instances that can be confidently classified as positive, one classifier covers the data that can be confidently classified as negatives, and the last classifier is used to handle the remaining "hard" instances. A similar idea of separating "hard" instances, but using a different approach was also explored in (Ferri et al., 2004).

To identify instances that can be confidently classified as positive or negative, we use *biased* classifiers. A classifier that is biased towards predicting positives will usually have a high precision on negative instances and vice versa. We train a biased classifier for each class (see Sec. 3.2). All instances where the biased classifiers agree are considered "easy", all other instances are "hard". The third classifier, the *arbiter*, is then trained only on those "hard" instances. The intuition is that the feature patterns among the "hard" instances may be different from those among the "easy" training examples. By separating away the "easy" instances and training the arbiter only on the "hard" ones, we make the learning problem for the arbiter easier since it only has to deal with a supposedly more regular subset of the data.

Like AdaBoost, we are trying to improve (boost) the classification accuracy of the base classifier on the training set by increasing the representational power using the ensemble. Both AdaBoost and Triskel try to enhance the decision surface of the ensemble by focusing on hard instances. The main difference is, however, in how the hard instances are defined. In AdaBoost, the hard instances are defined as the instances where the base classifier makes mistakes. In Triskel, the hard instances are defined as the instances that cannot be classified "confidently", where we assume that we can classify an instance "confidently" if a pair of biased classifiers agree on its label. The expectation is that we can achieve better results by splitting one hard classification problem into a series of easier ones, instead of constructing increasingly difficult problems as in AdaBoost.

Consider first the Triskel algorithm for binary tasks. Assume that a classifier is a function mapping data instances onto a binary set of classes: $h : X \to \{-1, +1\}$. Similarly to AdaBoost, Triskel is an iterative algorithm. In each iteration, we train a pair of biased classifiers: one classifier biased towards the positive class, and one classifier biased towards the negative class. For a discussion of different ways of biasing classifiers, see Sec. 3.2.

Next, we evaluate the biased classifiers on the training data and obtain two sets of instances: "easy" examples, where the biased classifiers agree; and "hard" examples, where they disagree. To obtain the training set for the next iteration, the weights of the "easy" instances are reduced and the weights of the "hard" instances are increased. The training set obtained after the last iteration is used to train the *arbiter* classifier. Algorithm 1 shows the details.

To combine the decisions of the learned classifiers, we use a conventional weighted voting scheme, with the weights set in such a way that some ensemble members' votes can dominate the others. Specifically, we use a sequence of exponentially decreasing weights such that if two biased classifiers from a given iteration agree on the label of a new instance, then their combined vote outweighs the votes of the classifiers from all subsequent rounds. Essentially, in each iteration we classify and separate the "easy" instances and then use the ensemble members from subsequent iterations to handle the remaining "hard" instances in a recursive way.

Algorithm 1 Triskel

/* *To train on* $\{\ldots,(x_i,y_i),\ldots\}$ $(y_i = \pm 1)$ */
Choose the method of weight adjustment:
$W_{easy} = 0; W_{hard} = 1$, or /* *"separation"* */
$W_{easy} = 1/2; W_{hard} = 2$ /* *"soft covering"* */
$D_0(i) = 1/N$ for each instance i
for $t = 1, 2, \ldots, K$ **do**
  $h_t^+ =$ Learn with weights D_{t-1}, biased to class +1
  $h_t^- =$ Learn with weights D_{t-1}, biased to class -1
  $\alpha_t = 2^{K-t}$
  for each instance i **do**
    $\Delta_{t,i} = \begin{cases} W_{easy}, & \text{if } h_t^+(x_i) = h_t^-(x_i) = y_i \\ W_{hard}, & \text{otherwise} \end{cases}$
    $D_t(i) = D_{t-1}(i) \cdot \Delta_{t,i}$ and normalise
  end for
end for
$h_{K+1} =$ Learn with weights D_K, unbiased
$\alpha_{K+1} = 1$
/* *To classify instance* x */
return $y = \text{sign}\left[\sum_{t=1}^{K+1} \alpha_t h_t^*(x)\right]$, where $h_t^*(x) = h_t^+(x) + h_t^-(x)$ for $t \leq K$, and $h_{K+1}^*(x) = h_{K+1}(x)$.

There are two principle ways in which the instance weights can be adjusted during training. One way is to set the weights of the "easy" instances to zero, leaving the weights of the "hard" instances unchanged. In this case, the classifiers in each subsequent iteration are trained on a shrinking subset of the training data. This method is similar to covering, since after each iteration, the covered part of the training instances is completely removed from consideration. The problem with this method is that it may quickly "run out of instances". That is, the number of instances left in consideration may quickly become too small to train a sensible classifier. The second way to adjust the instance weights is more similar to boosting, when the weights of "easy" instances are reduced, while the weights of "hard" instances are increased. In our experiments, we increase or reduce the weights by the factor of 2 (see Algorithm 1).

3.1. Handling Multi-class Problems

Two well-known methods to convert a multi-class problem into a series of binary problems are *one-vs-all* and *one-vs-one*. In the one-vs-all method, the initial multi-class problem is converted to a set of C binary classification problems, where $C > 2$ is the number of classes. In each of these binary problems, the goal is to learn to separate a selected class from the instances belonging to all other

classes. The one-vs-one method is very similar to one-vs-all, except that a binary classification problem is created for each pair of classes, and a separate binary classifier is trained for each of these tasks. Classification of a new instance is done by classifying it with all binary classifiers and then voting their predictions.

Both of these methods have disadvantages. The one-vs-all method is known to perform quite poorly when used in conjunction with SMO. In contrast to that, the one-vs-one method performs well, but is computationally more expensive, because a larger number of binary classifiers is needed (for $C > 3$). However, in conjunction with Triskel it is possible to use a compromise between one-vs-all and one-vs-one methods. We call this extension Triskel-M. For each class, a binary problem is created in order to separate this class ('positive instances') from all others ('negative instances'). These classifiers are biased towards high precision on the positive class and used similar as in binary Triskel: if exactly one of the biased classifiers predicts positive, this prediction is returned. If more than one or none of the biased classifiers predict positive, the prediction of the arbiter is returned. The arbiter should be trained in one-vs-one mode to achieve a better accuracy.

3.2. Generating Biased Classifiers

Some machine learning algorithms have an inherent way of setting a bias. Bayesian classifiers, for example, output a probability distribution. The class with the highest posterior probability as calculated by the classifier is predicted. It is easy to bias a Bayesian classifier by either modifying the prior probabilities or to impose biased thresholds on the posterior probabilities. Support Vector Machines also use a confidence value threshold.

There are, however, more generic ways to bias classifiers. Resampling techniques have been used to address the problem of imbalance in the training set. But resampling can of course also be used to create an imbalance, which is what we need for Triskel. Akbani et al. (2004) found that for imbalanced datasets undersampling the majority class to eliminate the bias leads to good performance, although some of the training examples are discarded.

We implemented a biased classifier with undersampling with a tunable bias parameter that ranges from 0 (no bias) to 1 (discard all instances of one class). In preliminary experiments, we also experimented with oversampling to create biased classifiers, but we found that creating the bias through undersampling does not hurt the overall performance of Triskel, even if as little as 10% of the training instances of one class are kept (bias value 0.9). For some datasets, the performance was even slightly better than the approach with oversampling. Additionally, because we drop 90% of the instances for one class, training becomes faster. Therefore we decided to use undersampling with bias 0.9 for our final experiments.

4. Experimental Results

We evaluated Triskel on several multi-class datasets from the well-known UCI repository. Because of its very good accuracy, we chose AdaBoost as the benchmark ensemble algorithm for our experiments. We used SMO (Platt, 1999) with a linear kernel as a base classifier. However, additional experiments conducted with nonlinear SMO kernels and Naive Bayes classifiers suggest that our claims are valid regardless of the base classifier.

When comparing ensemble methods, accuracy is not the only important factor. The reduced error of ensemble algorithms comes at the price of a greater computational effort. Therefore, time and memory consumption has to be compared as well. Both are usually related to the ensemble size. We used three different AdaBoost ensembles with 3, 10 and 50 rounds. We compared these against standard Triskel with 1 round and discarding easy instances for the arbiter ($W_{easy} = 0$) (Triskel-1) and against Triskel with weighting ($W_{easy} = 1/2; W_{hard} = 2$) with 2 and 4 rounds (denoted as Triskel-W2 and Triskel-W4). Note that for a binary Triskel, the actual ensemble size is twice the number of rounds plus one. SMO can handle only binary problems. We chose a one-vs-one scheme to convert the multi-class problems into a series of binary problems. For comparison, we also used Triskel-M (see section 3.1) with 1 round (denoted as Triskel-M1). For AdaBoost, boosting the binary classifiers individually yielded a better performance than using AdaBoost-M1 (Freund & Shapire, 1997).

We used the Weka framework (Witten & Frank, 1999) to conduct our experiments. We evaluated all algorithms using 10-fold cross-validation, with 10 randomised repetitions for statistical significance testing, using a corrected resampled t-test as implemented in the Weka experimenter. Detailed results can be found in Tables 1 and 2.

AdaBoost-50 shows the best performance compared to SMO and the other configurations of AdaBoost. Triskel-W4 outperforms AdaBoost-50 with 3 significant wins out of the 15 datasets used while being considerably faster due to the smaller ensemble. Triskel-W2 achieves a performance that is comparable to AdaBoost-10 (2 wins, 2 losses), but is significantly faster most of the time because of the smaller ensemble size. As expected, Triskel-M1 is both the least accurate but also the fastest ensemble method. Although the biased classifiers are only trained in a one-vs-all mode, the ensemble can still significantly outperform the base SMO in one-vs-one mode on the anneal.ORIG, hypothyroid and segment datasets. Because of its one-vs-all nature of the biased classifiers, this setup of

Triskel can be faster than one-vs-one plain SMO, especially on large datasets (here on the audiology, hypothyroid and soybean datasets), while not hurting accuracy.

Figure 2 illustrates the relation between training time and accuracy for the algorithms on six typical datasets. The data points on the Triskel line correspond to (from fastest to slowest) Triskel-M1, -1, -W2 and -W4, while the data points for AdaBoost show the setup for 3, 10 and 50 rounds. Note that in most cases the line for Triskel is above the AdaBoost line, indicating that Triskel offers a better trade-off between accuracy and speed. Triskel achieves greater accuracy in the same time, and the same accuracy can be reached faster. Furthermore, note that the highest accuracy for Triskel is usually above the highest accuracy for AdaBoost, indicating that, given enough time, Triskel can typically outperform any setting of AdaBoost.

5. ROC Analysis

In this Section, we present a qualitative description of Triskel using ROC analysis. ROC analysis has recently become a popular technique for studying classification algorithms (Flach & Wu, 2005).

Assume a binary classification problem. The ROC space has the false positive rate FPr on its X-axis and the true positive rate TPr on its Y-axis. A classifier (hypothesis) for a given dataset can be mapped onto a point in the ROC space. For example, the perfect classifier is mapped to $(0, 1)$, and the random hypothesis is mapped to $(0.5, 0.5)$. If classifier $\bar{h}$ outputs the opposite of what h predicts, then it is equivalent to mirroring h relative to the $(0.5, 0.5)$ point in the ROC space, since $TPr(\bar{h}) = 1 - TPr(h)$ and $FPr(\bar{h}) = 1 - FPr(h)$. By definition, for a given accuracy A, $TPr(h) = A + N/P[A + FPr(h) - 1]$, where P is the number of positives and N is the number of negatives in the dataset. Therefore, a given accuracy value corresponds to a set of different combinations of TPr and FPr that form a line in the ROC space. If a hypothesis is mapped below the 0.5 accuracy line, its performance on the dataset is worse than random. The accuracy of such hypothesis can be improved by simply reversing its decisions (see Figure 3(a)).

Consider now how biasing a classifier affects its position in the ROC space. Let h be the hypothesis learned by the unbiased base learner on the given dataset. Let h^+ be a biased classifier for class $+1$, i.e. h^+ is less likely to predict $+1$ and, thus, is expected to have a higher precision for $+1$. Similarly, h^- is a biased classifier for class -1.

We make the following *inclusion assumption*: $(\mathcal{TP}(h^+) \cup \mathcal{FP}(h^+)) \subseteq (\mathcal{TP}(h) \cup \mathcal{FP}(h)) \subseteq (\mathcal{TP}(h^-) \cup \mathcal{FP}(h^-))$, where $\mathcal{TP}()$ is the set of true positives, and $\mathcal{FP}()$ is the set of false positives for the corresponding hypothesis. That is, whenever h^+ predicts $+1$, h also predicts $+1$, and whenever h predicts $+1$, h^- also predicts $+1$. Under the inclusion assumption, we obtain that $TPr(h^+) \leq TPr(h)$ and $FPr(h^+) \leq FPr(h)$. Similarly for class -1, we obtain that $TPr(h^-) \geq TPr(h)$ and $FPr(h^-) \geq FPr(h)$.

It is easy to see that the inclusion assumption always holds, when h, h^+, and $h-$ classifiers are obtained by setting different confidence thresholds for the same probabilistic model (e.g. a Bayesian model). The assumption may not hold when we bias an SVM by re-weighting instances. Apart from moving the decision boundary, biasing can also change the orientation (slope) of the hyperplane (i.e. the hyperplanes of biased classifiers may not be parallel). However, we can expect that the violating overlap would be insignificant for a sufficiently strong bias.

Consider the set of the hard instances, i.e. the instances where the biased classifiers disagree. If we look at the ROC space for these instances, then h^+ is mapped there to the $(0, 0)$ point, and h^- is mapped to $(1, 1)$. Suppose now that we stretch or shrink the X and Y axis of this ROC space to position its $(0, 0)$ and $(1, 1)$ corners into the h^+ and h^- points in the initial ROC space for the whole dataset as shown in Figure 3(b). Essentially, the ROC space for the hard instances (the arbiter's dataset) becomes *embedded* into the ROC space for the whole dataset.

Prop. 1: *If we map the performance of the arbiter classifier on the hard instances to the corresponding point in the embedded ROC (sub)space (taking into account scaling of its X and Y axis), then the performance of a single-round Triskel with this arbiter on the whole dataset will correspond to the position of this point in the initial ROC space.*

Proof. Let y be the Y coordinate of the arbiter classifier in the initial ROC space. If $TPr(h^A)$ is the true positive rate of the arbiter on the hard instances, then $y = TPr(h^+) + TPr(h^A)(TPr(h^-) - TPr(h^+))$. Under the inclusion assumption, the number of positive instances P_A in the arbiter's dataset can be computed as $P_A = |\mathcal{TP}(h^-)| - |\mathcal{TP}(h^+)| = P(TPr(h^-) - TPr(h^+))$. The true positives rate of the single-round Triskel ensemble is by definition equal to $TPr(e) = (|\mathcal{TP}(h^+)| + |\mathcal{TP}(h^A)|)/P = TPr(h^+) + TPr(h^A)P_A/P = y$. The analysis for the X coordinate is analogous. □

For example, if we pick the centre point in the embedded ROC space, its position in the outer ROC space will correspond to the performance of a single-round Triskel with the given biased classifiers h^+ and h^- and a random arbiter.

Prop. 1 can be recursively applied to the embedded ROC space. This way we can represent running Triskel with multiple rounds and reason about the performance of a Triskel ensemble. It is easy to see that for a given pair of biased classifiers obtained at round 1, the best possible performance of Triskel on the training set (with

Dataset	SMO	vs	Triskel-M1		Ada-3	vs	Triskel-1		Ada-10	vs	Triskel-W2		Ada-50	vs	Triskel-W4	
anneal	97.46		97.90		99.02		97.85		99.13		99.07		98.99		99.39	
anneal.ORIG	87.44		95.83	◇	91.11		94.04	◇	90.04		94.21	◇	90.42		94.11	◇
audiology	80.77		80.10		81.43		79.58		81.39		80.28		81.39		80.42	
autos	71.34		70.27		74.95		71.31		76.35		76.73		75.92		76.59	
balance-scale	87.57		87.63		87.73		87.78		89.09		88.05		89.09		91.65	◇
Glass	57.36		56.94		60.66		61.95		62.23		63.40		62.33		63.41	
hypothyroid	93.58		93.87	◇	95.62		95.27		96.15		95.04	∗	96.35		95.91	
lymphography	86.48		85.00		86.28		84.73		85.32		84.87		85.19		84.40	
primary-tumor	47.09		47.44		43.93		45.23		42.96		45.05		42.72		42.93	
segment	92.92		93.61	◇	93.66		94.31		94.11		95.08	◇	94.48		95.95	◇
soybean	93.10		92.56		93.32		92.65		93.28		92.93		93.29		92.71	
vehicle	74.08		74.15		75.30		75.40		77.53		77.39		79.21		79.48	
vowel	70.61		70.60		89.91		83.71	∗	92.27		90.00	∗	94.08		93.80	
waveform	86.48		86.49		86.50		86.43		86.45		86.44		86.44		86.36	
zoo	96.05		95.08		96.05		95.15		96.05		96.05		96.05		96.05	

Table 1. Accuracy comparisons between SMO, AdaBoost and different settings of Triskel on multi-class problems from the UCI repository. ◇ denotes a significant increase in accuracy of Triskel over the other algorithm at the 0.05-level according to a corrected resampled t-test, ∗ denotes a significant decrease in accuracy. Note that compared to AdaBoost-10 and AdaBoost-50 Triskel achieves equal or better performance with a considerably smaller ensemble.

Dataset	SMO	vs	Triskel-M1		Ada-3	vs	Triskel-1		Ada-10	vs	Triskel-W2		Ada-50	vs	Triskel-W4	
anneal	4.45		5.66	◇	7.15		12.92	◇	13.78		25.07	◇	34.50		48.32	◇
anneal.ORIG	4.17		5.46	◇	10.79		11.64	◇	38.52		23.87	∗	95.87		46.41	∗
audiology	85.02		58.79	∗	120.83		538.19	◇	172.62		1069.06	◇	140.52		1065.85	◇
autos	5.05		8.54	◇	11.17		16.97	◇	36.24		30.37	∗	130.76		56.23	∗
balance-scale	0.99		2.36	◇	3.22		3.34	◇	12.27		5.45	∗	15.88		9.72	∗
Glass	4.73		8.05	◇	12.33		15.67	◇	39.10		26.58	∗	106.23		46.96	∗
hypothyroid	16.83		9.63	∗	38.24		12.84	∗	135.19		39.87	∗	289.24		70.09	∗
lymphography	1.95		2.56	◇	2.83		6.59	◇	1.47		10.73	◇	22.25		18.91	∗
primary-tumor	68.11		75.35	◇	145.22		280.4	◇	449.42		542.17	◇	1683.63		1204.90	∗
segment	8.06		11.43	◇	16.85		26.84	◇	56.43		52.77	∗	132.11		116.10	∗
soybean	61.49		37.30	∗	80.45		289.36	◇	174.30		604.43	◇	148.99		779.23	◇
vehicle	1.87		4.80	◇	7.28		6.90	∗	35.14		13.01	∗	90.06		25.63	∗
vowel	21.12		30.98	◇	44.11		62.61	◇	171.23		128.49	∗	676.99		272.55	∗
waveform	5.14		11.19	◇	47.36		8.30	∗	217.95		29.34	∗	361.99		78.60	∗
zoo	7.15		6.48		7.31		21.96	◇	1.85		39.91	◇	7.87		88.39	◇

Table 2. Training time comparisons between SMO, AdaBoost and Triskel on multi-class problems from the UCI repository. ◇ and ∗ denote statistically significant differences as in Table 1. When comparing AdaBoost-10 and -50 to Triskel-W2 and -W4, Triskel usually trains faster due to the smaller ensemble. Note that Triskel-M1 is sometimes faster than SMO.

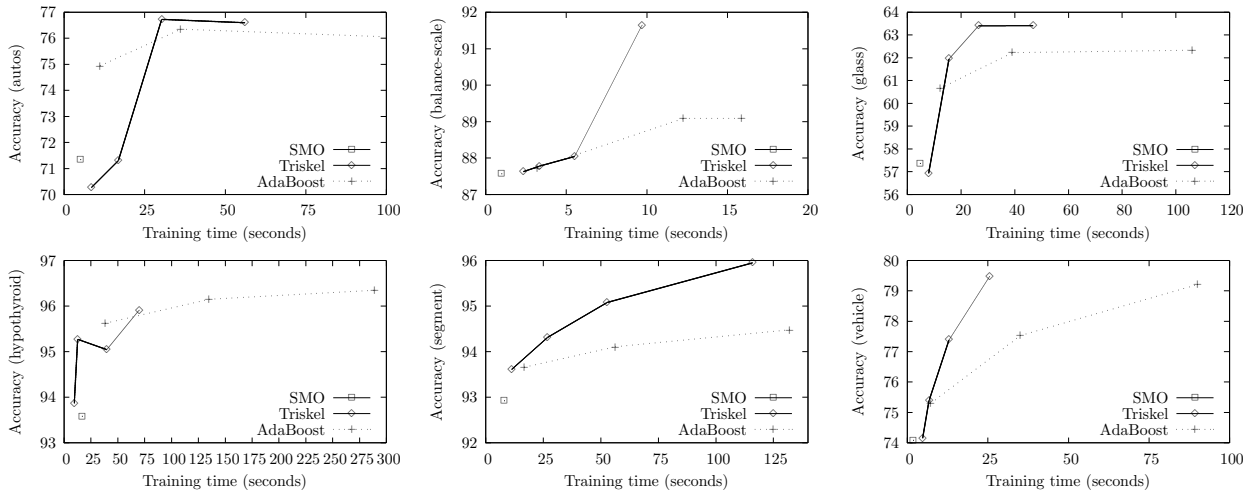

Figure 2. Accuracy and training time for SMO, Triskel-M1, Triskel-1, -W2, -W4, AdaBoost-3, -10 and -50 on the (from left to right and top to bottom) “autos”, “balance-scale”, “glass”, “hypothyroid”, “segment” and “vehicle” datasets.

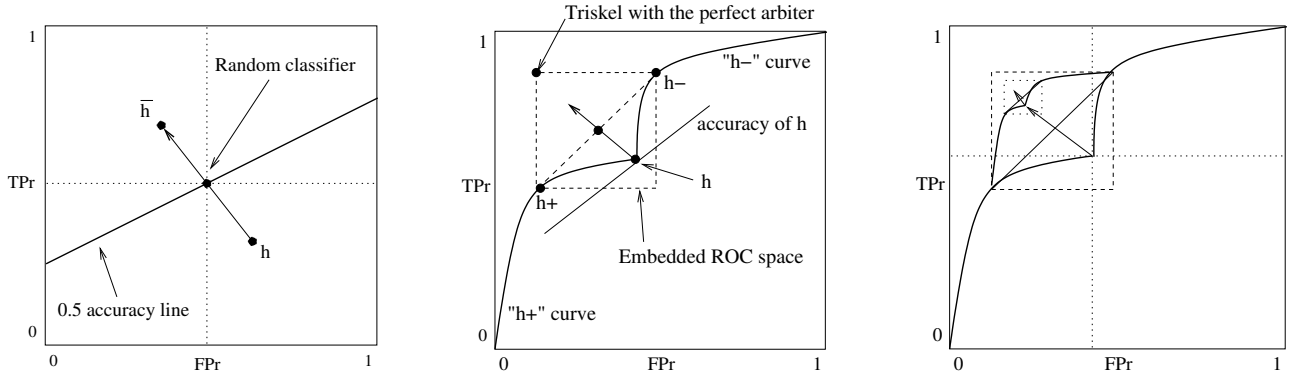

Figure 3. From left to right: (a) Reversing classifiers in the ROC space; (b) Triskel in the ROC space; (c) Triskel over multiple rounds.

any number of rounds $K \geq 1$) corresponds to point $(FPr(h_1^+), TPr(h_1^-))$. This suggests that the closer we get $FPr(h_1^+)$ to 0 and $TPr(h_1^-)$ to 1 the better, because then we have a larger potential for improvement if we use a very clever arbiter (e.g. an ensemble from many rounds of sub-Triskeling). It may seem that the stronger the initial bias, the better. Unfortunately, as we increase the bias, h^+ eventually ends up in $(0, 0)$ and h^- ends up in $(1, 1)$. The arbiter is then left with the whole initial dataset, which makes the biased classifiers on the first round useless.

By changing the bias strength for each of the classifiers h^+ and h^-, we can obtain a family of biased classifiers which would form *bias curves* in the ROC space. Consider this for the h^+ classifier first. If there is no bias, then h^+ is equivalent to h (the unbiased classifier) and it would map to the same point in the ROC space as h. As we increase the bias, h^+ will eventually start always predicting -1. This would correspond to the $(0, 0)$ point. Therefore, the bias curve for h^+ connects the h point with $(0, 0)$. Similarly, the h^- curve connects h with $(1, 1)$ as demonstrated in Figure 3(b).

The shape of these bias curves can be used to predict whether Triskel will improve the performance of the base classifier on the training set. In particular, if there is a pair of points on the bias curves such that the middle point of the line connecting them is above the accuracy line of the base classifier on this training set (classifier h in Figure 3(b)), then a Triskel ensemble with these biased classifiers will have a better accuracy than the base learner. To show this, it is sufficient to point out that the above middle point corresponds to the performance of a Triskel ensemble with the two selected biased classifiers and a random arbiter. Essentially, if the middle point is above the accuracy line of the base classifier, it means that the accuracy of the base classifier h on the hard instances is worse than random and, hence, could be improved even by simply reversing its decisions for those instances (see Figure 3(b)). Figure 3(c) illustrates how the performance of the ensemble may improve when Triskel is performed over multiple rounds.

Since in Triskel we train the arbiter specifically on the hard instances, it is possible that we can improve the ensemble performance on the training set even when the centre point of the embedded ROC space is below the accuracy of the base classifier. For example, if at least one of the biased classifiers h^+ or h^- is above the accuracy line of h (but the middle point is below), then we can improve the accuracy of the Triskel ensemble even if we use an arbiter that always predicts $+1$ (if h^+ is above the accuracy line) or -1 (if h^- is above the accuracy line). Our base learner should be at least clever enough to always predict the majority class.

Reversing the base classifier (trained on the whole dataset) for the hard instances is also relevant to training the arbiter without removing the easy instances, but giving a lower weight to them (i.e. soft covering, $W_{hard} > W_{easy} > 0$). In both cases we take into account all instances, but pay particular attention to the difficult ones either by reversing the base classifier for those instances or by assigning a higher weight to the hard instances when training the arbiter. This may reduce the risk of overfitting on the hard instances, and it does seem from our experiments that soft covering works better than $W_{easy} = 0$.

Another way to characterise a dataset suitable for Triskel is to say that the bias curves on this dataset form a concavity. The "deeper" the concavity, the bigger improvement is possible. If the concavity reaches the $(1, 0)$ point in the embedded ROC space of the hard instances, it means that the pattern of the hard instances is nearly opposite to the rest of the dataset. This suggests that boosting will not be effective on such dataset, since it will likely lead to cyclic redistributions of weights between easy and hard instances without improving the ensemble performance. To give an empirical illustration, Figure 4 shows bias curves for a randomly generated "layered cake" dataset (see Figure 1) using oversampling as the biasing method.

It is easy to see that if we pick the biased classifiers to form the deepest concavity, then simply reversing the base

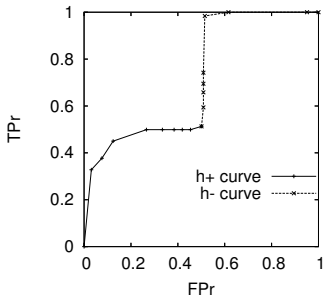

Figure 4. Biasing classifiers for the "layered cake" dataset.

classifier on the hard instances gives us a significant improvement. Also, the performance of the base classifier would be very close to the $(1, 0)$ in the embedded ROC space (i.e. boosting is likely to be ineffective here), and we know that AdaBoost does not work for this dataset. Another interesting point here is that the optimal positions of the biased classifiers on this dataset correspond to a quite small bias (around 0.1). This picture suggests that Triskel with a strong bias would not work well on this dataset. Indeed, running Triskel even with bias 0.4 already gives no improvement compared to the base classifier. This picture also suggests that asymmetric bias can work better (due to random asymmetries in the generated dataset). In particular, for this dataset the optimal bias for h^+ is around 0.11, while for h^- it is around 0.085. Running Triskel with these asymmetric biases gives 0.803 accuracy on the training set, while the maximum accuracy when running Triskel with a symmetric bias is only 0.703 for the bias value of 0.09

6. Conclusion

We have presented a novel ensemble learning algorithm called Triskel that makes use of biased classifiers to separate "easy" and "hard" instances. In its iterative nature, it is similar in style to Boosting methods, while the way Triskel separates easy and hard instances is also related to covering algorithms. Empirical results suggest that, compared to AdaBoost, Triskel offers a better trade-off between accuracy and speed. Furthermore, the experiments show that the maximum accuracy that can be achieved with Triskel is higher than the accuracy of AdaBoost.

We have also presented an ROC analysis of Triskel, which shows that Triskel's iterative structure corresponds to a sequence of nested ROC spaces, where each embedded space is defined by the previous iteration's biased classifiers. This analysis allows us to reason about the behaviour of Triskel at a qualitative level. In particular, it predicts that Triskel works best when there are concavities in the ROC curve; this prediction agrees with our empirical results. We can also see how the upper performance bound of the classifier changes depending on the biased classifiers.

We are currently investigating many aspects of Triskel that we could only sketch briefly in this paper. For example, in this paper we have reported results with undersampling as the only method of generating the biased classifiers. In future work we would like to explore the space of possible methods for generating biased classifiers, such as setting the bias of the classifier directly by means of thresholding, or using oversampling with artificially created instances as used in the SMOTE algorithm (Chawla et al., 2002).

Acknowledgements: This research was supported by Science Foundation Ireland (grant SFI/01/F.1/C015) and the US Office of Naval Research (grant N00014-03-1-0274).

References

Akbani, R., Kwek, S., & Japkowicz, N. (2004). Applying Support Vector Machines to Imabalanced Datasets. *15th European Conference on Machine Learning*. Pisa.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*, 123–140.

Breiman, L. (1999). Pasting Small Votes for Classification in Large Databases and On-Line. *Machine Learning*, *36*, 85–103.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, *16*, 341–378.

Dietterich, T. G. (2000). Ensemble Methods in Machine Learning. *First International Workshop on Multiple Classifier Systems*. New York.

Ferri, C., Flach, P., & Hernandez-Orallo, J. (2004). Delegating Classifiers. *21st International Conference on Machine Learning*. Banf.

Flach, P., & Wu, S. (2005). Repairing concavities in ROC curves. *19th International Joint Conference on AI*. Edinburgh.

Freund, Y. (1995). Boosting a weak learning algorithm by majority. *Information and Computation*, *121*, 256–285.

Freund, Y., & Shapire, R. E. (1997). A Decision-Theoretic Generalization of On-Line Learning. *Journal of Computer and System Sciences*, *55*, 119–139.

Fürnkranz, J. (1999). Separate and Conquer Rule Learning. *Artificial Intelligence Review*, *13*, 3–54.

Platt, J. C. (1999). *Fast Training of Support Vector Machines using Sequential Minimal Optimization*, chapter 12, 185–208. MIT Press.

Shapire, R. E. (1990). The Strength of Weak Learnability. *Machine Learning*, *5*, 197–227.

Vapnik, V. N. (1982). *Estimation of dependencies based on empirical data*. New York / Berlin: Springer Verlag.

Witten, I. H., & Frank, E. (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann.