AI Programming Project #3:
Boosting with Diverse Classifiers

Purpose:

- Learn to combine several AI systems into one.
- Understand Ensemble Learning by implementing ADABOOST.

1 Assignment

1. Implement the ADABOOST algorithm on page 751 of the AI textbook ("Artificial Intelligence: A Modern Approach").
2. Implement two classifier types to be used by your booster:
   (a) A naive bayesian classifier (NBC)
   (b) A decision-tree classifier (DTC)
3. Test your boosted classifiers on 5 different data sets.
4. Analyze the results in detail.

2 Ensemble Learning

Imagine that you are given the entire question set for a board game, such as Trivial Pursuit, where each question has Q=4 answer choices. Furthermore, imagine that you have K = 10 good friends whom, you are certain, can perform better than randomly on these questions: given the entire set, all would get more than $\frac{1}{4}$ correct. Now, when you sit down to play the game, instead of answering the questions yourself, you will call all K friends, get their answers, and use the majority answer as your response.

As it turns out, this is an exceptionally good strategy, and one that borders on optimal for large K. In Machine Learning, this is the essence of ensemble learning: given a large enough set of classifiers, all of which perform better than randomly on the same training set, then majority-based classification approaches perfection on that same training set as K increases.

Ensemble Learners can combine many different classifiers, and many different **types** of classifiers. For example, the classifier pool may consist of several Bayesian reasoners, feed-forward neural networks, self-organizing maps, and decision trees; or it may be comprised of only a few dozen support vector machines.

2.1 Boosting

In ensemble learning, there are two complementary approaches to training the individual classifiers: bagging and boosting. In bagging, each classifier trains on a different subset of the original data set, while boosting uses the same training set for all. However, boosting algorithms attach a weight to each data instance, roughly indicating the importance of it being classified correctly by the current classifier. These weights change from classifier to classifier, so each is built under the influence of a different distribution of priorities, which can have a large effect upon the behavior of the constructed classifier.

Figure 1 depicts the basic boosting process, showing the interaction between hypotheses/classifiers and the training set. During hypothesis generation, the training data and their weights determine the structure of the resulting classifier, while the accuracy of the classifier on each instance determines its ensuing weight change.

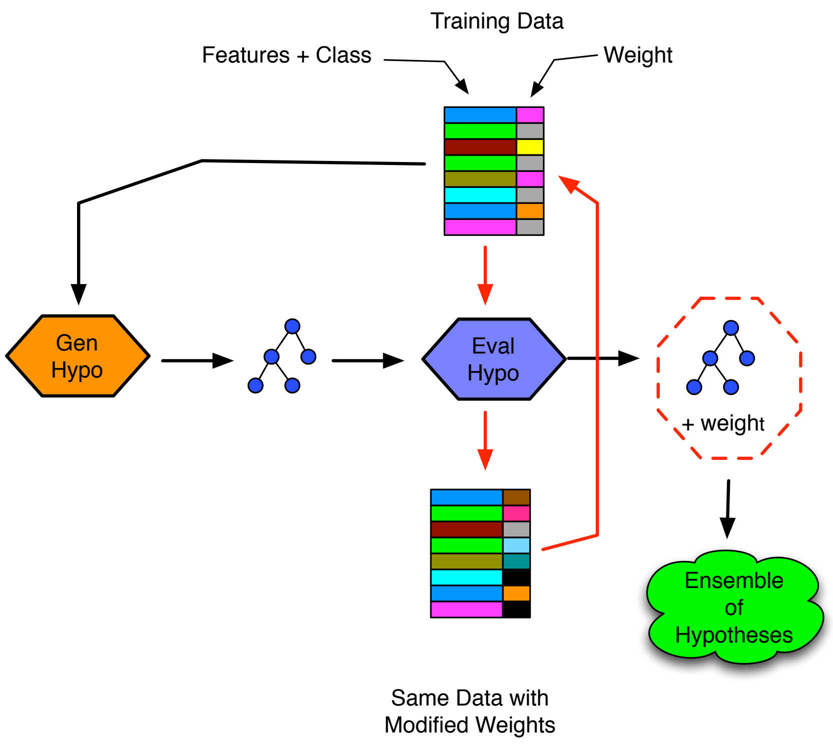

Figure 1: Overview of Boosting. Hypotheses/classifiers (depicted as trees) are generated using the training data and their current weights. The classifier is then evaluated on each training instance, with the results affecting both a) the weight attached to the classifier, and b) the modifications performed on each instance's weight. The classifier and associated weight are then added to the ensemble.

Given an original data set, the booster divides it into training and testing subsets. It then uses the training set to build each of the K hypotheses/classifiers. During testing, each test case is classified by all K hypotheses of the ensemble, with a weighted vote among their classifications being the final answer returned by the booster.

Two types of weights are the cornerstones of boosting:

- Each classifier is weighted according to its accuracy on the **training** data. Thus, after the learning

system uses the training data to build a classifier (e.g. a decision tree, a naive Bayes classifier, etc), it runs the classifier on the entire training set and sums the weights of the instances that were incorrectly classified. That sum is used to compute the weight of the classifier as:
$\log \frac{1-error}{error}$

- Each training instance is weighted according to the **previous classifier's** ability to classify it correctly, with weights increasing (decreasing) for incorrectly (correctly) classified instances.

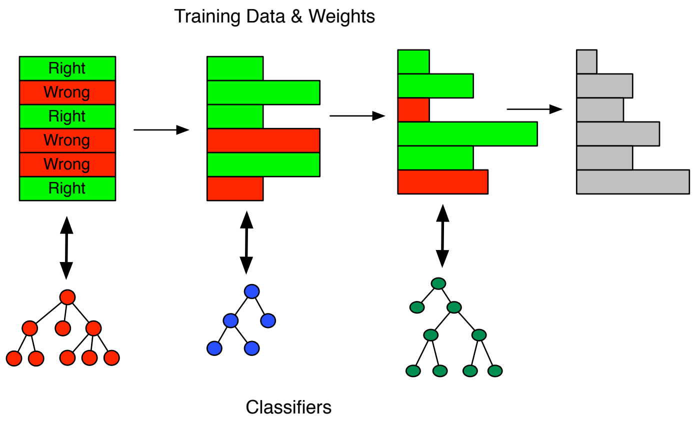

Figure 2: Modification of weights (represented by the width of each training instance) during boosting. Correctly classified instances (green) experience weight decreases in the next round of classifier generation, while missed instances (red) have their weights increased. The rightmost data set has yet to be run through a new classifier, but its instances' widths reflect performance by the previous classifier.

The basic weight-modification process for data instances is visualized in Figure 2. Notice that each classifier is constructed and evaluated relative to a different set of weights. Essentially, instances that were poorly classified by a previous classifier will have higher weights when sent to the next round of classifier generation, so that (hopefully) the next classifier will be built so that can handle the difficult instance.

It is important to note that even though each classifier may be generated by the same algorithm, such as a decision-tree learner or a Bayesian-network builder, and the same training instances, each call to the learner/builder will involve a different set of weights for the training instances. In many cases, the different instance weights will lead to the production of different decision trees or conditional probability tables, respectively.

During testing, the ensemble of hypotheses is employed as shown in Figure 3. Each votes on the new data instance, with the votes weighted by each classifier's credibility. The sum total of all weighted votes determines the ensemble's output classification.

2.2 Implementing ADABOOST

One of the most popular variants of boosting is the ADABOOST algorithm. There are three main aspects to ADABOOST:

1. Generating hypotheses (a.k.a. classifiers) from the weighted training data. The procedures for this are specific to the classifier type, following the normal construction method with slight twists to account for the case weights. These twists are explained below for Bayesian and decision-tree classifiers.

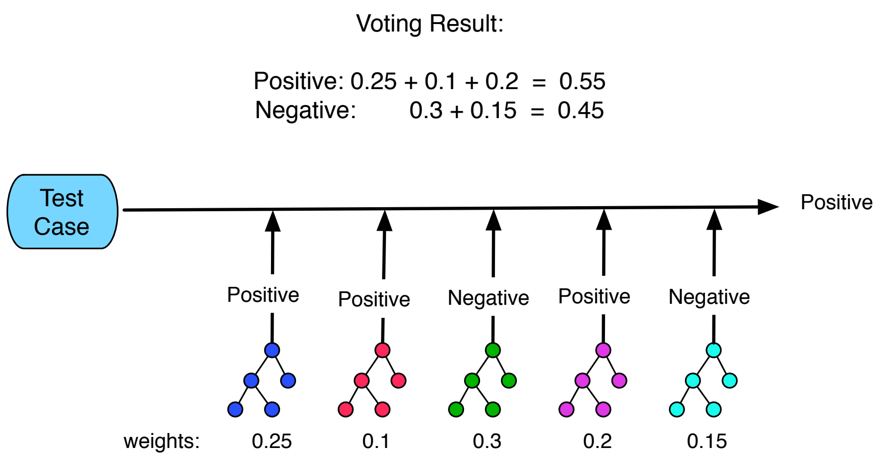

Figure 3: Testing new data instances using an ensemble of hypotheses, each of which gives its recommended classification, which is scaled by the classifier's weight/credibility.

2. Determining the weights of classifiers and modifying those of training instances. This is the core of boosting and explained below in detail.
3. Employing the ensemble of hypotheses to perform weighted-majority voting during the testing phase. This is a very simple procedure involving querying each classifier with test cases and tabulating the results. For example, if the test cases involve k different classes, then the class with the highest weighted sum of classifier votes is the winner. In Figure 3, k = 2, and the *positive* class wins.

Weight calculations in ADABOOST are performed when a hypothesis is evaluated. The weight of the hypothesis (H) itself is a function of its total error on the complete training set, while the weights of individual training instances shift up and down depending upon both a) H's response to the instance, and b) the total error across all responses given by H. Initially, prior to the construction of the first classifier, each instance has the same weight: $\frac{1}{N}$, where N is the size of the data set.

There are several different versions of ADABOOST; the following comes from page 751 of *Artificial Intelligence: A Modern Approach* (3rd edition).

The two pseudocode procedures below handle the training phase, in which K hypotheses are generated, each based on a different setting of the training-case weights. The *Generate-Hypothesis* module is specific to the type of classifier being produced, e.g. neural network, decision tree, etc.

```
**ADABOOST-TRAINING**(K,C)
Given N training cases C = $\{(x_1, y_1)...(x_N, y_N)\}$, this generates K hypotheses.
$\forall i : w_i = \frac{1}{N}$ (Initialize W = $\{w_i\}$, the case weights)
 For k = 1 to K:
        $H_k \leftarrow$ Generate-Hypothesis(W)
        W $\leftarrow$ Update-Weights($H_k$,W)
 Return $\{H_1, H_2, ...H_K\}$
```

```
**UPDATE-WEIGHTS**(H,W)
$error \leftarrow 0$
$\forall i$ : If $H(x_i) \neq y_i$ then $error \leftarrow error + w_i$
$\forall i$ : If $H(x_i) = y_i$ then $w_i \leftarrow w_i(\frac{error}{1-error})$
$w_T = \sum_i w_i$
$\forall i : w_i \leftarrow \frac{w_i}{w_T}$ (Normalize all weights)
weight(H) $\leftarrow \log \frac{1-error}{error}$
```

In other words, for each hypothesis, H, the error is the sum total of the weights of all instances that H classifies incorrectly. That error is then used to update the weights of all instances, and to calculate H's weight.

The instances that were correctly classified by H receive weight updates directly, by the code above, while incorrectly-classified instances have their weights changed as a result of a) direct changes to the correct instances, and b) the weight-normalization process. The relationship between error and weight change is complicated by the non-linear update equation:

$$w_i \leftarrow w_i(\frac{error}{1-error}) \tag{1}$$

The essential qualitative behavior is this (where w_i is the weight of a correctly-classified instance):

- If error $= 0.5$, then w_i does not change.
- If error < 0.5, then w_i decreases. Hence, after normalization, weights of unsolved examples will increase, thus increasing their importance.
- If error > 0.5, then w_i increases. Hence, after normalization, weights of unsolved examples will decrease. Here, there is so much error that the solved examples need to be emphasized. Alternatively, some boosting algorithms simply throw away classifiers that give such a high error value, and thus, these weak hypotheses cannot influence the training-case weights. Feel free to implement either option.

Once these general details of weight assignment and adjustment are in place, the specific usage of instance weights in classifier formation must be addressed. The next two sections describe two different classifier types and how each should be tailored to work with a boosting algorithm.

3 The Naive Bayesian Classifier

Naive Bayesian Classifiers compute the most likely class for a set of attributes using:

- Bayes rule to compute probabilities of classes given attributes by using the probabilities of attributes given classes, and
- the naive assumption that all attributes are conditionally independent, given the class.

The basic idea is easier to understand by looking at diagnosis, where the relationship between causes (diseases) and effects (symptoms) mirrors the relationship between classes and attributes in learning, since classes are can be viewed as causes of the attributes.

For a particule cause, c_i and particular effects $e_1, \ldots, e_n$, Bayes rule implies:

$$p(c_i|e_1, \ldots, e_n) = \frac{p(e_1, \ldots, e_n|c_i)p(c_i)}{p(e_1, \ldots, e_n)} \tag{2}$$

If C is the complete set of m mutually-exclusive causes, then:

$$p(e_1, \ldots, e_n) = \sum_{i=1}^{m} p(e_1, \ldots, e_n|c_i)p(c_i) \tag{3}$$

Now, let us use the notation $P(C \mid e_1, \ldots, e_n)$ to denote the vector of conditional probabilities of all possible causes, given these specific effects:

$$P(C \mid e_1, \ldots, e_n) = \langle p(c_1 \mid e_1, \ldots, e_n), p(c_2 \mid e_1, \ldots, e_n), \ldots p(c_m \mid e_1, \ldots, e_n) \rangle \tag{4}$$

Then we can compute all of the probabilties in this vector in the following manner:

$$\begin{aligned} & P(C \mid e_1, \ldots, e_n) \\ = & \frac{P(e_1, \ldots, e_n|C)P(C)}{p(e_1, \ldots, e_n)} \\ = & \frac{P(e_1, \ldots, e_n|C)P(C)}{\sum_{i=1}^{m} p(e_1, \ldots, e_n|c_i)p(c_i)} \\ = & \alpha P(e_1, \ldots, e_n|C)P(C) \\ = & \alpha P(C) \prod_i P(e_i|C) \end{aligned}$$

In this derivation, each quantity beginning with P is a vector of probabilities, not a single probability. The second line is a vector divided by a scalar, which produces a new vector with each element divided by the same scalar. The introduction of α is shorthand to indicate that the elements of the vector are normalized, i.e., they are all divided by their sum. As shown in equation 3, that sum is the same as $p(e_1, \ldots, e_n)$. Thus, when working with these vectors over the conditional probablities of all causes, we never need to explicitly represent $p(e_1, \ldots, e_n)$. The final step of the derivation represents the *naive assumption* that all of the effects are conditionally independent of one another.

Diagnosis is just a special case of classification, so we can change the variable names from *Cause* to *Class* and from *Effect* to *Attribute* and derive similar relationships:

$$\begin{aligned} & P(Class \mid Attribute_1, \ldots, Attribute_n) \\ = & \frac{P(Attribute_1, \ldots, Attribute_n|Class)P(Class)}{P(Attribute_1, \ldots, Attribute_n)} \end{aligned}$$

$$
\begin{aligned}
&= \alpha P(Attribute_1, \ldots, Attribute_n | Class) P(Class) \\
&= \alpha P(Class) \prod_i P(Attribute_i | Class) *
\end{aligned}
$$

Here, the attributes are assumed conditionally independent, given the class, and normalization is based on the fact that, for a complete set of mutually-exclusive classes:

$$P(Attribute_1, \ldots, Attribute_n) = \sum_{c_i \in Classes} P(Attribute_1, \ldots, Attribute_n | c_i) P(c_i) \tag{5}$$

Given a set of attributes, the Naive Bayesian Classifier finds the class hypothesis with the maximum a-posteriori probability, h_{MAP}, i.e. the class that maximizes $P(Class) \prod_i P(Attribute_i | Class)$.

3.1 Building a Naive Bayesian Classifier (NBC)

Computing h_{MAP} is a simple operation when you have data concerning $P(Class_i)$ for all classes (i.e. the a-priori probabilities of each class) and $P(Attribute_j | Class_i)$ for all classes and attributes. These values come from prior experience, or in our case, from a pre-defined data set in which each case consists of a set of attributes and its associated class. Cases are known as *atomic events* in probability theory.

For example, the data set might consist of millions of atomic events from the Blackjack tables of Las Vegas, as shown in Figure 1. For ease of explanation, assume that the set consists only of the 8 cases shown. Then, the a-priori probabilities of the two classes, hit and no hit, would each be 0.5, since exactly half of the cases exhibit each of these actions.

The conditional probabilities to compute have the form $p(attribute_i \mid class_j)$, where the two classes are hit and no-hit, and the attributes are these situations: a) over 15, b) below or equal to 15, c) greater than 10 face cards remaining, d) 10 or fewer face cards remaining, e) 0 opponents remaining, f) 1 opponent remaining, g) 2 opponents remaining, h) 3 opponents remaining, etc.

Again, assuming the truncated data set of 8 cases, the value of $P(RemainingPlayers = 2 \mid Hit? = true)$ is estimated to be 0.75, since, of the 4 cases where hit is true, 3 have 2 remaining players as well.

All of the a-priori and conditional probabilities can be estimated by these simple counting methods. Once stored in tables, these values enable the straightforward calculation of h_{MAP} for any combination of attributes.

When running an NBC **without** BOOSTING, the basic approach is the following:

- Read in a data set of atomic events.
- Separate the data set into a training set and a test set, with a ratio of 3:1 or 4:1, i.e. most of the instances can be used for training.
- Use the training data to create tables for a) the a-priori probabilities of the classes, and b) the conditional probabilities of individual attributes given classes. Each training instance should have the same weight. Differential weighting will only be used during boosting.
- For each instance (I) in the **test set**, which consists of attributes(I) and class(I):

Table 1: Fictitious training instances from Blackjack games in Las Vegas. The class is whether or not to take a *hit*, i.e., another card, and the attributes a) whether or not the current hand is over 15, b) the number of opponents remaining in the hand, and c) whether more than 10 face cards are remaining in the deck (which assumes a certain degree of card counting).

Hit?	Over 15?	Opponents Remaining	> 10 face cards remaining?
Yes	No	2	No
Yes	Yes	4	Yes
No	Yes	1	Yes
No	Yes	1	Yes
No	Yes	2	No
Yes	Yes	2	No
Yes	Yes	2	No
No	Yes	1	No
...	...	...	...

  - Use attributes(I) to compute the h_{MAP}.
  - Compare h_{MAP} to class(I).
  - If h_{MAP} = class(I), record a hit, else record a miss.

- Return the test accuracy, which is simply: $\frac{hits}{hits+misses}$

3.2 Weighted Training Instances in Naive Bayesian Classification

In a Boosted Naive Bayesian Classifier, the NBC uses the training set to build conditional probability tables, which then provide the basis for computing the maximum a-posteriori hypothesis, h_{MAP}, as described above. In normal non-boosted NBC, the weights of all training instances are assumed equal.

For example, consider an even smaller set of Blackjack cases:

Table 2: Unweighted training-instance set

Hit?	Over 15?	Opponents Remaining	> 10 face cards remaining?
Yes	No	2	No
Yes	Yes	4	Yes
No	Yes	1	Yes
No	No	0	No
Yes	Yes	2	No

Using this table to generate conditional probabilities, consider $P(Over15? = true \mid Hit? = true)$. First we gather all instances in which Hit? = true. Among those 3 instances, two have Over 15? = true, so $P(Over15? = true \mid Hit? = true) = 0.666$ is one entry in the conditional table for Over 15?, given Hit?.

Now consider the same training set, but with uneven weights:

To compute $P(Over15? = true \mid Hit? = true)$ in this case, gather all instances with Hit? = true, as before.

Table 3: Weighted training-instance set

Hit?	Over 15?	Opponents Remaining	> 10 face cards remaining?	Weight
Yes	No	2	No	0.4
Yes	Yes	4	Yes	0.1
No	Yes	1	Yes	0.15
No	No	0	No	0.25
Yes	Yes	2	No	0.1

But now, the estimated probability becomes:

$$P(Over15? = true \mid Hit? = true) = \frac{0.1 + 0.1}{0.4 + 0.1 + 0.1} = 0.333 \tag{6}$$

Notice that now we calculate fractions based on the weights of the instances, not simply the counts.

This general idea is used to generate all conditional tables for the NBC. Since each NBC will be based on a differently-weighted training set, each will have different probabilities in its conditional probability tables. Thus, different NBCs can give different answers to the same classification problem. And, of course, if you need to generate K NBCs, then you will need K sets of conditional probability tables.

4 Decision-Tree Classifiers

A decision tree (DT) represents a function for computing the class of a feature vector, where the function, if written as a computer program, would consist of nested IF-THEN-ELSE statements. The execution of that program embodies a walk from root to leaf of the DT. At the root of each subtree of the DT, a question is asked of the feature vector, such as ”Is your value for feature F5 a 1, 2 or 3?” Depending upon the answer, the walk proceeds down a particular branch, which invokes another question, channelling the search down another branch, etc. Eventually, the DT reaches a leaf node, which houses the classification of all feature vectors that guide search to that particular destination.

Ideally, a DT can find the proper classification using a minimum of questions: the tree is shallow. In other words, no matter what feature vector is received as input, the DT search moves quickly to a leaf. So, for example, if you could build a DT for the classic game of ”20 questions”, then an efficient DT would be able to come up with the target answer using, on average, few (or at least less than 21) questions.

Decision Trees are very useful as classifiers because they can look at the data with a fine-toothed comb. If there are no inconsistencies in the data, then a perfect DT can be built for the data set, although it may involve a lot of questions (i.e. subtrees) to funnel each vector to the proper leaf node. Few other classifier types go to this level of detail in the sense of testing, for example, the value of feature 3 only if feature 2 had a value of 7 and feature 1 has a value of 5.

The general inductive problem is simple: given a data set, build a decision tree that properly classifies all feature vectors in the data, and hopefully can properly classify many others not in the training set. For DT’s, training constitutes building the tree in a manner that efficiently partitions the data.

Consider the data set depicted in Figure 4. Though shown pictorially, these items could be represented as 3-element vectors (shape, size, color) with an associated classification of positive or negative. The goal is to build a DT for this data set that can classify (as positive or negative) an item with these 3 features in as few questions, on average, as possible.

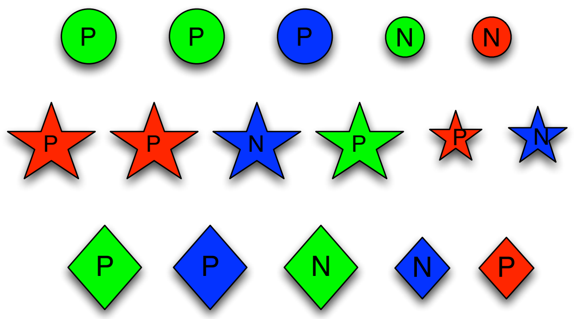

Figure 4: A data set consisting of 16 items whose features are: size, color and shape, and whose class is either positive (P) or negative (N). Features: **Color** (Red, Blue, Green), **Size** (Big, Small), **Shape** (Circle, Star, Diamond). Counts: Red: 5, Blue: 5, Green: 6 Big: 10, Small: 6 Circle: 5, Star: 6, Diamond: 5, Positive: 10, Negative: 6.

The task would be easy if, for example, all red objects were positive and all blue and green objects were negative. Then, simply by asking for the color attribute of an instance, the class could be determined immediately, without further questioning. Thus, the DT for such a data set would consist of a root node housing the *Color?* question. Three branches would come out from the root, one for each of the possible color values. The branch labeled *red* would lead to a positive leaf, while the branches for green and blue would each lead to a different negative leaf.

However, in this case, the data set is not so trivial to separate into positives and negatives. It takes more than one question. Intuitively, we would like to ask for features that do a good job of partitioning the set of training instances such that *most* of the instances that have a particular value for that feature are positive, while *most* that have a different value are negative. The closer *most* comes to *all*, the more appropriate the question (at that particular point in the tree).

Beginning with the entire data set, let us experiment with the three different questions and see which gives the most clear-cut partition. Figures 5, 6 and 7 illustrate the partitions for each of the 3 options. Note that none of the three give a perfect partition. In fact, for all 3 questions, every subtree still contains a mixture of positive and negative instances. Thus, no question gives clearly-superior results over the others, at least based on a simple qualitative analysis.

Fortunately, information theory provides quantitative criteria for comparing distributions: the concept of *entropy*, which expresses the uncertainty in a probability distribution. Formally, the entropy of a system (S) is:

$$H(S) = -\sum_{s_i \in S} p(s_i) log_2 p(s_i) \tag{7}$$

where S is modeled as the set $\{s_i\}$ of mutually-exclusive states, $p(s_i)$ is the probability of state s_i, and $\sum_i p(s_i) = 1$. H(S) then characterizes the uncertainty of the system, with a maximum value (of $log_2 M$, where M is the size of $\{s_i\}$) occurring when all states are equiprobable, and a minimum value of zero when one state is known to be true and all others false. Intuitively, if we are certain about the state of a system, then one state has a probability of 1, while all others are zero; and if we are maximally uncertain, then each

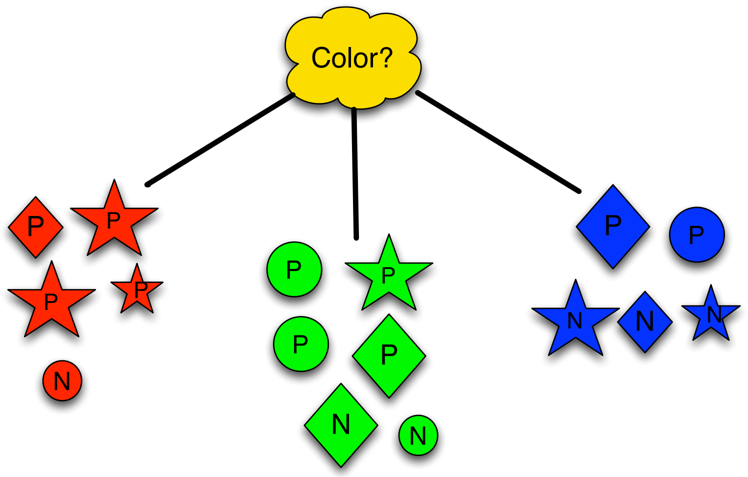

Figure 5: Partitioning of the data set if *color* is the first queried feature.

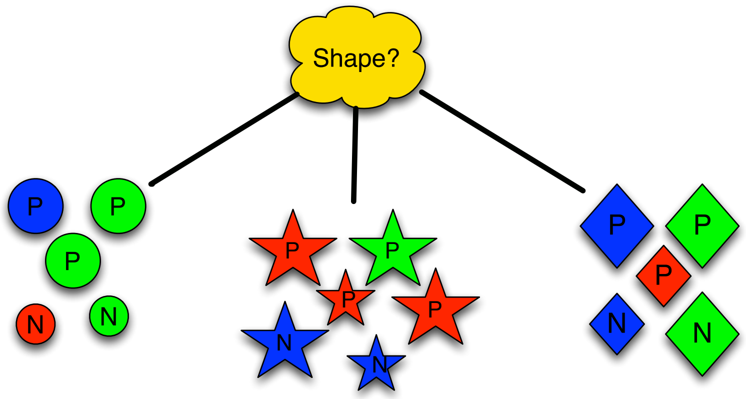

Figure 6: Partitioning of the data set with *shape* as the first queried feature.

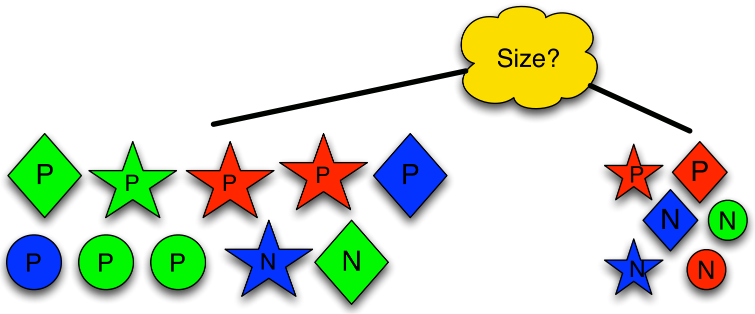

Figure 7: Partitioning of the data set with *size* as the first queried feature.

state has an equal probability. So the quantitative notion of entropy makes good qualitative sense as well.

Next, the utility of a query can be measured in terms of the average entropy that it produces down each answer branch. A query that produces low entropy (i.e. low uncertainty) along each branch should be preferred to one that has several high-entropy (i.e. high uncertainty) branches. So if a question partitions the data set (D) into m subsets, $d_1, d_2, ...d_m$ (one for each of the m possible values for the feature), then the average entropy of the partition is:

$$\sum_i \frac{\| d_i \|}{\| D \|} H(d_i) \tag{8}$$

where $\| \, . \, \|$ denotes the cardinality of a set (i.e., d_i or D), and H(d_i) is the entropy of the ith subset.

We can then use this equation to quantitatively assess the 3 questions. Initially, the entropy of the distribution of items is:

$$-\frac{10}{16}\log_2\frac{10}{16} + -\frac{6}{16}\log_2\frac{6}{16} = 0.954 \tag{9}$$

since 10 of the 16 items are positive, and the other 6 negative. The feature that is queried should reduce this entropy as much as possible.

If *color* is queried first, then the following calculations show the weighted entropies down each of the three branches in Figure 5, which sum to 0.873.

- Red: $\frac{5}{16}(-0.8\log_2 0.8 + -0.2\log_2 0.2) = .226$
- Green: $\frac{6}{16}(-0.666\log_2 0.666 + -0.333\log_2 0.333) = .344$
- Blue: $\frac{5}{16}(-0.4\log_2 0.4 + -0.6\log_2 0.6) = .303$

To see how this works, examine the *red* branch and its calculation. Note that 80% of the instances along that branch are positive, while 20% are negative, thus accounting for the 0.8 and 0.2 in the entropy calculation. The *red* branch entropy is multiplied by $\frac{5}{16}$, because 5 of the 16 data instances are red.

Next, consider the question of shape and find the following weighted branch entropies for Figure 6, which sum to 0.950.

- Circle: $\frac{5}{16}(-0.6\log_2 0.6 + -0.4\log_2 0.4) = .303$
- Star: $\frac{6}{16}(-0.666\log_2 0.666 + -0.333\log_2 0.333) = .344$
- Diamond: $\frac{5}{16}(-0.6\log_2 0.6 + -0.4\log_2 0.4) = .303$

This barely reduces the entropy at all, so it has little chance of being the best question. Conversely, if size is queried first, the following branch entropies for Figure 7 occur:

- Large: $\frac{10}{16}(-0.8 \log_2 0.8 + -0.2 \log_2 0.2) = .451$
- Small: $\frac{6}{16}(-0.333 \log_2 0.333 + -0.666 \log_2 0.666) = .344$

This yields $0.795 = 0.451 + 0.344$, which is the lowest of the average entropies among the 3 questions. Thus, *size* will remove the most uncertainty from the original data set and becomes the optimal first query to perform.

Decision-Tree creation now continues recursively. The optimal question of *size* produced two branch subsets. Neither of these has an entropy of zero, so further questioning is necessary to fully separate the positive and negative instances. There is no point querying *size* again, so the choice is among *shape* and *color*. Also, the questions can differ along the two branches: shape may give lower entropy than color along the size=large branch, but color could win along the size=small branch. So each branch requires a new set of tests to find the best entropy-reducing question.

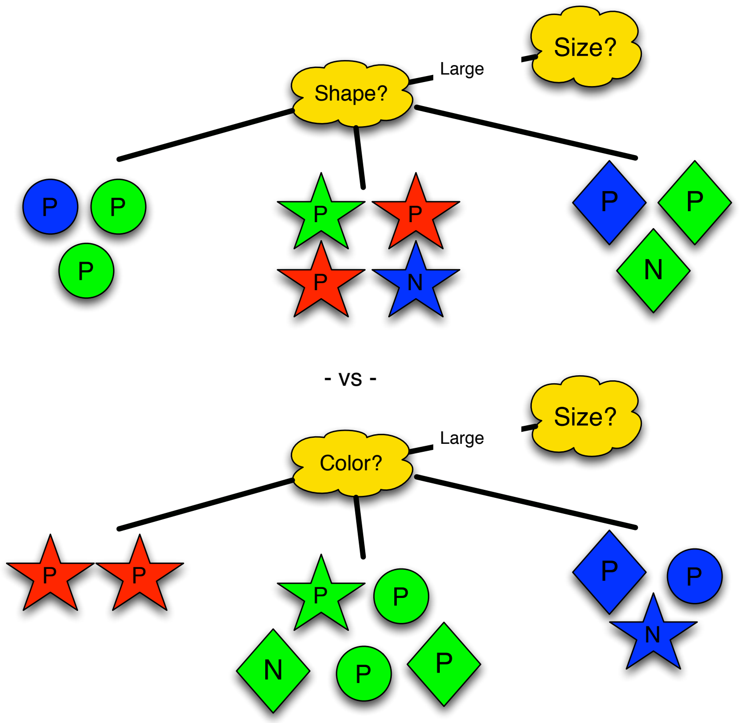

Figure 8: Comparison of distributions formed by the *shape* and *color* questions along the size=large branch

Figure 8 compares the partitions for both remaining questions along the size=large branch. Once again, eyeballing the figures does not immediately reveal the best entropy-reducing question, but a quick calculation reveals an average branch entropy of 0.600 (0 + 0.325 + 0.275) for the *shape* question:

- Circle: $\frac{3}{10}(-1.0 \log_2 1.0 + -0.0 \log_2 0.0) = 0$
- Star: $\frac{4}{10}(-0.75 \log_2 0.75 + -0.25 \log_2 0.25) = .325$
- Diamond: $\frac{3}{10}(-0.666 \log_2 0.666 + -0.333 \log_2 0.333) = .275$

While the *color* query produces an average branch entropy of 0.636 (0 + 0.361 + 0.275):

- Red: $\frac{2}{10}(-1.0\log_2 1.0 + -0.0\log_2 0.0) = 0$
- Green: $\frac{5}{10}(-0.8\log_2 0.8 + -0.2\log_2 0.2) = .361$
- Blue: $\frac{3}{10}(-0.666\log_2 0.666 + -0.333\log_2 0.333) = .275$

So *shape* is the best feature to query along the size=large branch. But along the size=small branch, *color* turns out to be the best query, as you can easily verify by doing the branch-entropy calculations for Figure 9.

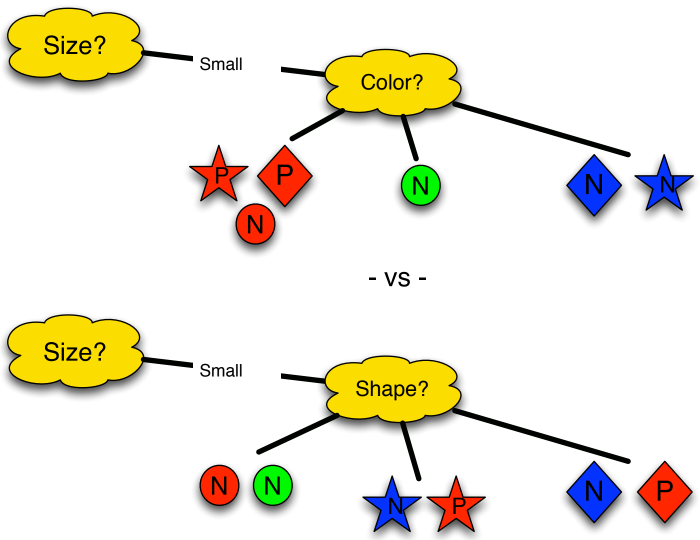

Figure 9: Comparison of distributions formed by the *shape* and *color* questions along the size=small branch.

After two levels of querying, some paths lead to nodes with entropies of zero, i.e., the training instances for the node (n) are either all positive or all negative. Node n then becomes a leaf node of the DT, with this unanimous classification becoming the class of the node, c(n). In future runs of the DT, the classification of any new feature vector (V) that ends up in n will be c(n). For example, in the DT produced so far, all large, red objects in the training set lead to a leaf node with a unanimous classification (positive), so the class for this node is *positive*, and any future vectors ending up there will be classified as positive.

If a node has non-zero entropy, then further questions are needed, with the ensuing partitions. However, if all questions are exhausted down a path, and the leaf node still has a non-zero entropy, then c(n) becomes the majority class among its training instances. If there are an equal number of each class (or of the two most popular classes in cases of 3 or more classes), then c(n_T) is ambiguous: the DT cannot give a reliable classification of any new instance that ends up in n (unless given more training data).

Figure 10 shows the decision tree produced by the recursive application of this basic entropy-based, node-splitting procedure. Notice that there are leaf nodes at levels 2 and 3: some instances can be classified using only 2 questions, but others require 3. However, some leaves have no classification, due to either an empty set of training instances for the node or an equal number of positive and negative instances.

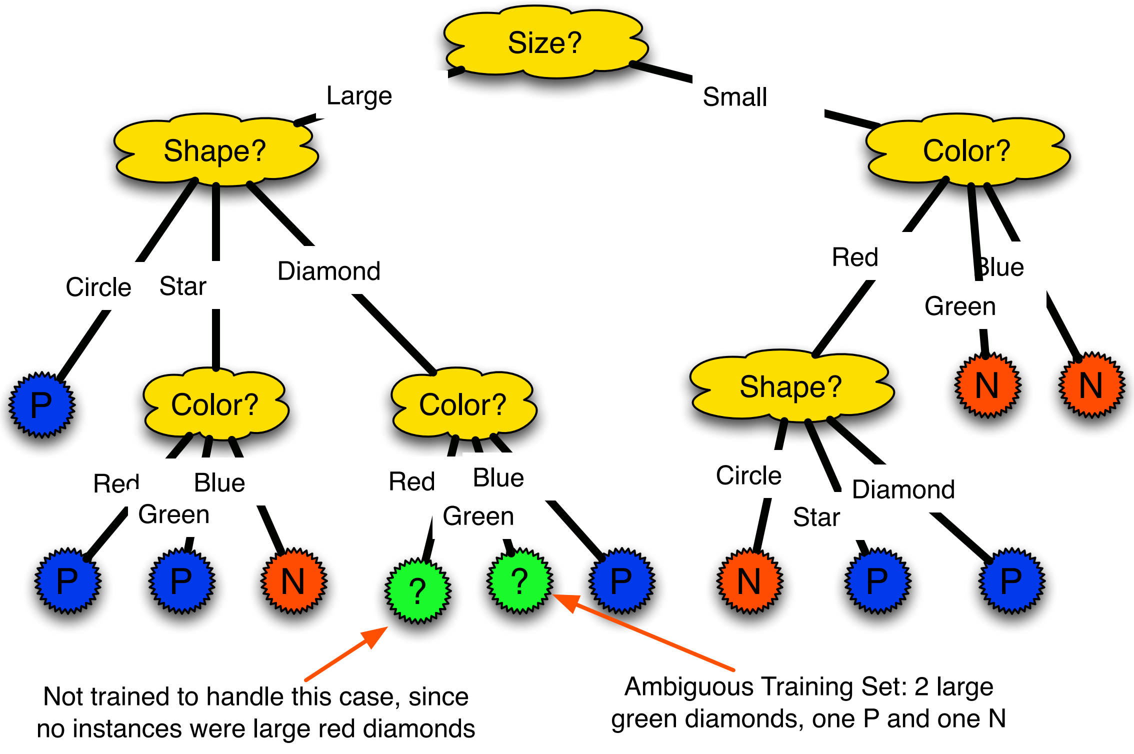
Size?
Large
Small
Shape?
Color?
Circle
Star
Diamond
Red
Blue
Green
P
Color?
Color?
Shape?
N
N
Red
Blue
Green
Red
Blue
Green
Circle
Star
Diamond
P
P
N
?
?
P
N
P
P
Not trained to handle this case, since
no instances were large red diamonds
Ambiguous Training Set: 2 large
green diamonds, one P and one N

Figure 10: The final decision tree for the 16-item collection. Labels on leaf nodes denote the unanimous classification of all training instances partitioned into that node. The label "?" denotes ambiguity.

This entropy-based approach to DT construction was introduced by J. Ross Quinlan in the 1980's, via his ID3 system, which he improved over the years, culminating in the C4.5 system. The basic algorithm is as follows:

```
Procedure Quinlan-DT(Instances,Features):
1. If Features = $\emptyset$ and Instances $\neq \emptyset$ then
     • Create leaf node with class = Unknown
     • Return
2. Test each f $\in$ Features to find f*, the feature whose query produces the best entropy-reducing partition of Instances.
3. f* partitions Instances into subsets $S_1, S_2, ...S_k$ where k = number of possible values of f*.
4. For i = 1 to k:
     • If H($S_i$) = 0 (zero entropy) then Create a leaf node with same class as the instances of $S_i$
     • Otherwise: Call Quinlan-DT($S_i$,Features - f*)
```

To implement this algorithm, you will also need to keep track of the internal and leaf nodes, their parents and children in the DT, their training-instance subsets, etc. However, this basic pseudocode skeleton, along with the entropy calculations, are the heart of Quinlan's popular method.

Once constructed, the DT then serves as a classifier. Any new instance presented to it will be led, by the established tree of questions, to a leaf node, which will either give a classification for the instance or an answer of *Unknown*. If restrictions are placed on DT creation, such as a depth limit for the trees, then more nodes classified as *Unknown* can be expected - as can more nodes that give an answer based on majority, but not unanimous, classes among their training instances.

As you can see from the construction procedure, DT's can be extremely specialized for a set of training instances. Given the freedom to query each feature (i.e., given a depth limit equal to the size of the feature set), they can potentially produce one leaf node for each training instance.

4.1 Decision-Tree Classifiers for Boosting

For DTCs, the weights of the instances will affect the information-gain calculations used to select next-best attributes. Previously, we assumed that all instances were of equal weight when doing entropy calculations. For boosting, we use the instance weights to bias these calculations. So even though an attribute query may produce a branch subset containing k positive and k negative examples, the entropy could still be low if, for example, all the positive examples have low weights but all the negative examples have high weights.

In Figure 11, consider a branch of the decision tree in which the weight of each positive or negative instance is written below it. The total weight of all examples along the *shape* branch is 0.35.

One alternative is to use the instance weights only as weights of the 3 branches, but not as parts of the individual-branch entropy calculations:

- Circle: $\frac{0.1}{0.35}(-0.666 \log_2 0.666 + -0.333 \log_2 0.333) = .262$

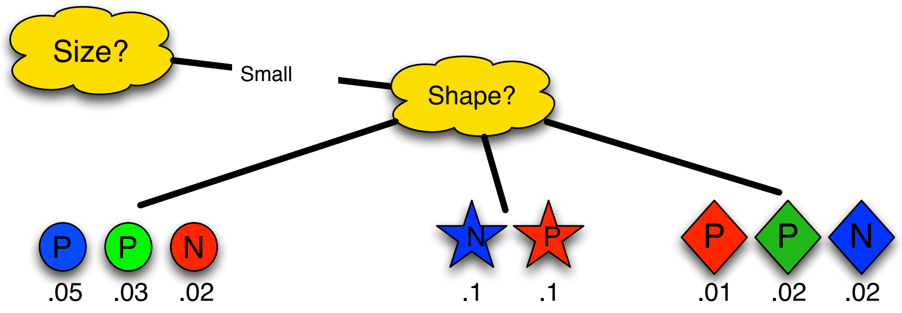

Figure 11: Using instance weights for building boosted decision-tree classifiers

- Star: $\frac{0.2}{0.35}(-0.5\log_2 0.5 + -0.5\log_2 0.5) = .571$
- Diamond: $\frac{0.05}{0.35}(-0.666\log_2 0.666 + -0.333\log_2 0.333) = .131$
- Total: 0.964

For example, the $\frac{0.1}{0.35}$ in the circle equation stems from the fact that the ratio of weights along the *circle* branch to the sum of all weights along the *shape* branch is:

$$\frac{.05 + .03 + .02}{.05 + .03 + .02 + .1 + .1 + .01 + .02 + .02} = \frac{0.1}{0.35} \tag{10}$$

Another option includes the weights in the individual-branch entropy calculations as well.

- Circle: $\frac{0.1}{0.35}(-0.8\log_2 0.8 + -0.2\log_2 0.2) = .206$
- Star: $\frac{0.2}{0.35}(-0.5\log_2 0.5 + -0.5\log_2 0.5) = .571$
- Diamond: $\frac{0.05}{0.35}(-0.6\log_2 0.6 + -0.4\log_2 0.4) = .139$
- Total: 0.916

For example, the 0.8 in the entropy calculation for the circle branch is based on the fact that the ratio of the positive-instance weights to the total weight of the circle branch is:

$$\frac{.05 + .03}{.05 + .03 + .02} = .8 \tag{11}$$

Clearly, then, as the instance weights vary during boosting, different DTCs can be built from the same data set, as long as the instance weights vary between calls to the decision-tree builder.

In this assignment, feel free to use the instance weights in either of the ways described above.

5 Boosting with Both Types of Classifiers

Your ADABOOST algorithm should accept (at least) the following inputs:

1. The number of Naive Bayesian Classifiers (NBCs).
2. The number of Decision-Tree Classifiers (DTCs).
3. The percentage of the data set to be used for training; the rest is used for testing.
4. All key parameters for the individual classifiers, such as the maximum depth for the DTCs.
5. The name of the data file.

It should not be necessary to rebuild your system in order to test different combinations of the above inputs. During demos, if you need to go to the source code to make the changes necessary to do different runs, then you will lose points.

The data sets should be loaded into a data structure that can be used by all the classifier-builder routines.

Your system will need to run on the following:

- The 3 data sets provided by your instructors.
- 2 additional data sets that you choose from the UC Irvine Machine-Learning Repository (archive.ics.uci.edu/ml/) or from some other source. The only constraints on a data set are that it contains at least 100 instances with at least 5 attributes (in addition to the classification), and that the set is based on REAL data, not hand-generated fantasy data.

For each of the FIVE data sets, use a 4:1 training-test ratio and perform ALL of the run combinations described below:

1. A single NBC.
2. A single DTC (with max depth = A, the number of attributes in a case.)
3. 5 NBCs
4. 10 NBCs
5. 20 NBCs
6. 5 DTCs (maximum possible depth)
7. 10 DTCs (maximum depth = 1; i.e. *stubs*)
8. 10 DTCs (maximum depth = 2)
9. 10 DTCs (maximum depth = A)
10. 20 DTCs (maximum depth = A)
11. 5 NBCs and 5 DTCs (maximum depth = 2)
12. 10 NBCs and 10 DTCs (maximum depth = 2)
13. 20 NBCs and 20 DTCs (maximum depth = 2)

Before performing these runs, you will want to do many preliminary runs with both classifier types to debug and find effective parameter settings (for any parameters whose values are NOT specified above).

For each of the many runs, document the following in easy-to-read tables (one table per data set):

- The average and standard deviation of the training errors for each type of classifier. For example, in runs using 20 NBCs and 20 DTCs, you will produce one average and standard deviation for EACH of the two sets of 20.
- The test error for the boosted set of all classifiers. For the two single runs of NBC and DTC (where no boosting is needed), simply show the test error of the classifier itself.

6 A Note on the Irvine Data Sets

In many of the UCI data sets, several of the attributes have quantitative values. If your chosen data set includes such attributes, you may want to preprocess the file and convert all quantitative values into simple qualitative values such as (high, medium and low) or a small set of integers (1, 2, 3 and 4).

For example, if the values of a particular attribute (A) are any real numbers in the range from 3.5 to 9.5, then you can simply convert each value to a qualitative value according to the following simple linear conversion factor:

- $a \in [3.5, 5.5) \rightarrow$ LOW
- $a \in [5.5, 7.5) \rightarrow$ MEDIUM
- $a \in [7.5, 9.5] \rightarrow$ HIGH

You may want to avoid symbolic names altogether and simply use the integers 0, 1 and 2 for low, medium, and high, respectively.

All files in the UCI Repository include a .names file, which contains information about the meanings of attributes, the ranges of their values, and even some historical background.

Also, be aware that some data sets in the repository have missing data. This is also specified on the summary page of the repository. If you use a file with missing data, you will have to take special measures to insure that your system does not crash on these cases. For this assignment, it is acceptable to go through the files and just fill in the missing data with the data-set average for that value over all the (complete) cases.

7 Deliverables

1. **40 points**: Working code for ADABOOST and for the generators of NBCs and DTCs. Your ADABOOST code must be general: there should **not** be any direct mention of the particular classifier (i.e. NBC or DTC) in your boosting code.
2. **35 points**: The 5 Tables, one for each data set.

3. **25 points**: Analysis of the results. Here, you need to a) find interesting trends in your data, and b) explain them as best you can. It is expected that you will write 2-3 pages of detailed and informative analysis.

On this assignment, you can work alone or in a group with AT MOST ONE other person.