Lecture 5: April 25

Lecturer: Yishay Mansour *Scribe: Adi Karol and Ami Peled*

5.1 Lecture Overview

In this lecture, we will first develop an algorithm for studying OR using a small number of examples. We will then look at algorithms for learning k-DNF's, decision lists and k-decision lists. We also deal with Decision Trees, and present an algorithm for learning them.

5.2 Learning OR with few attributes

In this section, our goal is to find an algorithm for learning an OR function using a small number of examples (logarithmic in the number of variables). We will look again at the *ELIM* algorithm from lecture 3.

5.2.1 Learning OR - ELIM algorithm

We try to learn a target function which is an OR of literals (e.g. $x_1 \vee x_4 \vee \overline{x_7}$). Our goal is to learn the function in time polynomial in k and in log n (for constant ε and δ). The *ELIM* algorithm is as follows:
Algorithm ELIM:

- Keep a list of candidate literals (initialize to all literals)
- For every example with classification 0:
  - Remove all literals that are 1
- Return h = OR of the remaining literals

Correctness: Since our initial h includes all literals, it includes also the target literals. The algorithm only removes literals which are 1 from examples classified as 0. These literals cannot be a part of the target function (otherwise the example should have been classified as 1). Hence, our resulting hypothesis h still includes all literals of the target function, and classifies all inputs which are 1 correctly. In addition, since we remove all literals which are 1 from examples classified 0, h classifies all these examples as 0 as well. In conclusion, h classifies all positive inputs (and specifically, all positive examples) and all negative examples correctly, as required, i.e. h is consistent with the sample.

Sample Size: Using PAC analysis we get $m > \frac{1}{\varepsilon}ln(\frac{3^n}{\delta}) = O(\frac{1}{\varepsilon}n + \frac{1}{\varepsilon}ln(\frac{1}{\delta}))$.

One can show that ELIM requires linear sample size. For example when the target function is $C_t = X_n$. The sample set might look like this:

Sample	$\mathbf{C_t}$	Variables in list
000...00	0	$x_1, x_2, x_3, ..., x_{n-1}, x_n$
100...00	0	$---, x_2, x_3, ..., x_{n-1}, x_n$
010...00	0	$---, ---, x_3, ..., x_{n-1}, x_n$
001...00	0	$---, ---, ---, ..., x_{n-1}, x_n$
.	.	.
.	.	.
.	.	.
000...10	0	$---, ---, ---, ..., ---, x_n$

In this case, we have to consider n-1 samples before we get the function (since after each example we can "get rid" of only one additional variable). Remember that our goal was to learn the function in time *logarithmic* in n. To achieve that goal, we will need to look at another problem first - Set Cover.

5.2.2 SET COVER

Input: sets $S_1, S_2, ..., S_t$ where $\forall i.S_i \in U$
Output: $S_{j_1}, S_{j_2}, ..., S_{j_k}$ such that $\bigcup_{l=1}^{k} S_{j_l} = U$ and k is minimal.

Although *Set Cover* is known to be NP-Hard, we can approximate it (to within a logarithmic factor) using a simple greedy algorithm.

Algorithm Greedy for Set Cover:

1. let C be a set of S_i sets.
2. let U_j be a subset of items from U which are still *not* in C.
3. Initialize $C \leftarrow \phi$, $U_0 = U$, $j = 0$.
4. While $U_j \neq \phi$ do
   - let $S_i = argmax_{(S_i)} \mid S_i \bigcap U_j \mid$ (S_i is the set with maximum elements from U_j).
   - Add S_i to C.

- let $U_{j+1} \leftarrow U_j / S_i$ (remove S_i's elements from U_j).
- $j \leftarrow j + 1$

Correctness: The algorithm clearly returns a cover of U (since it terminates only when all of U is covered, i.e. when $U_j = \phi$).

Analysis: We shall mark the optimal cover by C^*, and it's size $|C^*| = k$. C^* is a cover of U, therefore it is also a cover of each U_j. Since there are k sets in C^*, there must be (at least) one set $S \in C^*$ that covers (at least) $\frac{|U_j|}{k}$ elements of U_j. In each step, we pick a set that covers a maximal number of such elements, hence we cover at least $\frac{|U_j|}{k}$ new elements:

$$|U_{j+1}| \leq |U_j|(1 - \tfrac{1}{k})$$

The algorithm terminates when $|U_j| < 1$. Solving the recursion yields:

$$j \leq k \ln |U + 1|$$

Since in each step we add one set to C, this is also the number of sets in our final cover.

5.2.3 Learning OR - an OCCAM algorithm

We are now ready to build our OCCAM algorithm for studying OR. The algorithm runs *ELIM* on the training samples. The result of the algorithm is a hypothesis that is actually a set of literals OR'ed together. We saw that this subset correctly classifies all negative examples from T (We shall mark it T^-). Moreover, any subset of this set will correctly classify T^- (since removing literals from an OR term cannot cause the term to change its value from 0 to 1). As a result, when we try to reduce the size of this set further (to a logarithmic size in n), we need to look at the positive examples from T (mark T^+). The set returned by *ELIM* (mark LIT) classifies T^+ correctly. We wish to find the smallest subset of literals from LIT that still classify T^+ correctly, meaning our resulting set includes (at least) one literal which is 1 from every example of T^+. To find such a subset of LIT we shall use the *Set Cover* algorithm. We build a set S_z for each literal z in LIT. The set S_z will contain all T^+'s examples that are satisfied by z. Then we find a *Set Cover* for T^+ that should be as small as possible. Finally, we construct an hypothesis from literals corresponding to all sets in the cover.

Algorithm OR (second version):

1. INPUT: a sample T of size m (m examples).
2. Run *ELIM* on T.

3. let LIT be the resulting set of literals.

4. for each literal $z \in LIT$ build a set $S_z = \{x \in T^+ | z \text{ satisfies } x\}$.

5. Run *Greedy Set Cover* with sets S_z and universal set T^+.

6. Transform each set S_{z_i} in the cover to literal z_i.

7. Output h = OR of all such z_i literals .

Sample Size: We assume the correct target function is an OR of k literals. The analysis of the *Greedy Algorithm* implies that the resulting number of literals in h is at most $k \ln m$. Hence size$(h) \leq k \ln m \log 2n$. We can now solve by way of OCCAM's razor:

C_t = OR of k literals
H = OR of $k \ln m$ literals
$|H| = C(2n, k \ln m) \leq 2 \cdot (2n)^{k \ln m}$
$m = \frac{1}{\varepsilon} \ln \frac{|H|}{\delta} \leq \frac{1}{\varepsilon} \ln \frac{2 \cdot (2n)^{k \ln m}}{\delta} = \frac{1}{\varepsilon}[\ln 2(2n)^{k \ln m} + \ln \frac{1}{\delta}]$

Solving for m we get that our sample size: $m = O(k \log n \log(k \log n))$, which is (almost) logarithmic in n.

5.3 Learning k-DNF

Definition a k-DNF formula is a disjunction of clause, where each clause is a conjunction of (at most) k literals

Reminder: A literal is a variable or its negation.

Example:
term: $T = x_1 \wedge \overline{x_3} \wedge x_5 \wedge x_6$
DNF: $T_1 \wedge T_2 \wedge T_4$
3-DNF: $(x_1 \wedge \overline{x_2} \wedge x_5) \vee (x_2 \wedge x_7 \wedge \overline{x_2}) \vee (x_5 \wedge x_1)$

Our algorithm for learning a k-DNF will be based on the OR algorithm. We will transform the problem into an OR problem. In our transformed problem, each new literal ("new Input") x' will represent a possible clause. Furthermore, each example will be transformed by assigning each of the new literals the value of the clause it represents. Our k-DNF is now an OR of these new literals. We run *ELIM* over the new problem to obtain the requested hypothesis.

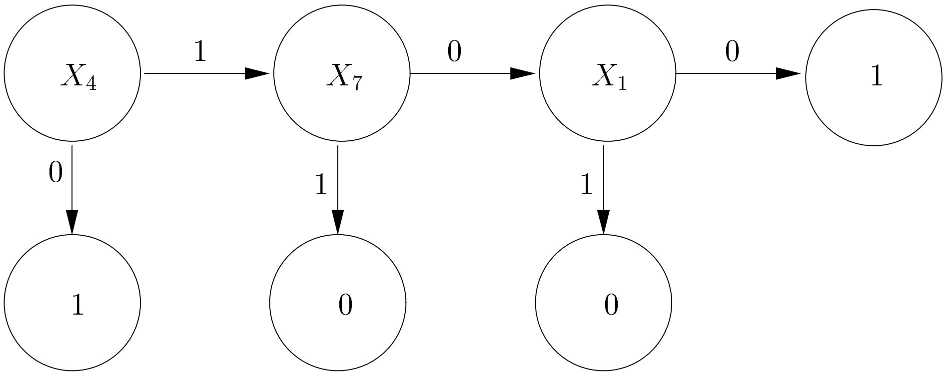

Figure 5.1: A Decision List Example

Sample size: The number of new literals ("new inputs") is at most $(2n)^k$. We can easily compute these "new inputs" in $O(k(2n)^k)$ time. Putting it into the formula we obtained for *ELIM* we get a sample size of $O(\frac{(2n)^k}{\varepsilon} + \frac{1}{\varepsilon}\ln(\frac{1}{\delta})$. The running time is the same.

5.4 Learning Decision Lists

Definition A decision list is a connected graph with the following structure:

1. The graph consists of internal nodes, each marked with a literal, connected in a directed list structure.
2. Each node in the list has an additional outgoing edge (pointing down in figure 5.1). Each of these edges points to a leaf (marked 0 or 1).
3. The last node in the list points to 2 leafs.

Each internal node in the list holds one literal. To find the value of a decision list in a given variable assignment we start from the leftmost node. We recursively follow the directed edges according to the value of the node's literal, until we reach a leaf. The classification of the leaf is the value of the DL on the specific assignment.

Our algorithm for learning decision lists is very similar to the *ELIM* algorithm. We start with a sample S of m examples. In each step of the algorithm, we build for each literal z a set T_z which holds all examples x of S in which z is 1 (we say that z satisfies example x). We then find a specific group T_{z_i} in which all of the examples have the same classification (such a group exists since we assume the examples are classified by a DL). We choose z_i as the next literal in the list (or its negation if the literal is already in negative form). Since

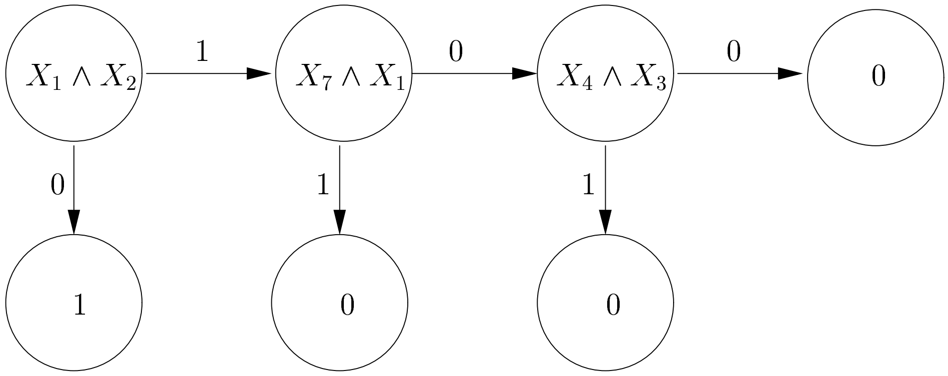

Figure 5.2: A 2-Decision List Example

all of the examples with the variable z_i (or $\overline{z_i}$) have the same classification, we are sure to be able to classify the leaf pointed at by the "down going" arrow correctly. We remove all of T_{z_i} examples from S and recurse. When S is empty, we are done.

Algorithm DL:

1. INPUT: a sample S of size m (m examples).
2. While S is not empty do:
   (a) For each literal z, build $T_z = \{x \in S | z \text{ satisfies } x\}$.
   (b) Find z_i such that all $x \in T_{z_i}$ have the same classification.
   (c) Add z_i to the decision list.
   (d) $S \leftarrow S - T_{z_i}$

Analysis: The output DL is consistent with the examples, since in each step we classify those examples which have the same classification. To obtain a bound on the number of examples needed, we notice that the length of a DL is less than n+1. An internal node can be any of 2n literals, and a leaf one of {1,0}. The total bound for the number of decision lists is $(2 \cdot 2n)^{n+1}$, giving us a bound of $m = O(n \log \frac{n}{\varepsilon} + \frac{1}{\varepsilon} \ln \frac{1}{\delta})$.

5.4.1 Learning k-Decision Lists

A k-DL is a DL in which each internal node is marked not with a single literal but rather with a conjunction of k literals. An example of a 2-DL is in figure 5.2. A k-DL is a generalization of k-DNF and k-CNF.

In order to learn a k-DL, we will use a similar transformation as we did for the *ELIM OR* algorithm for k-DNF's. We create a new literal x' ("new input") for each conjunction

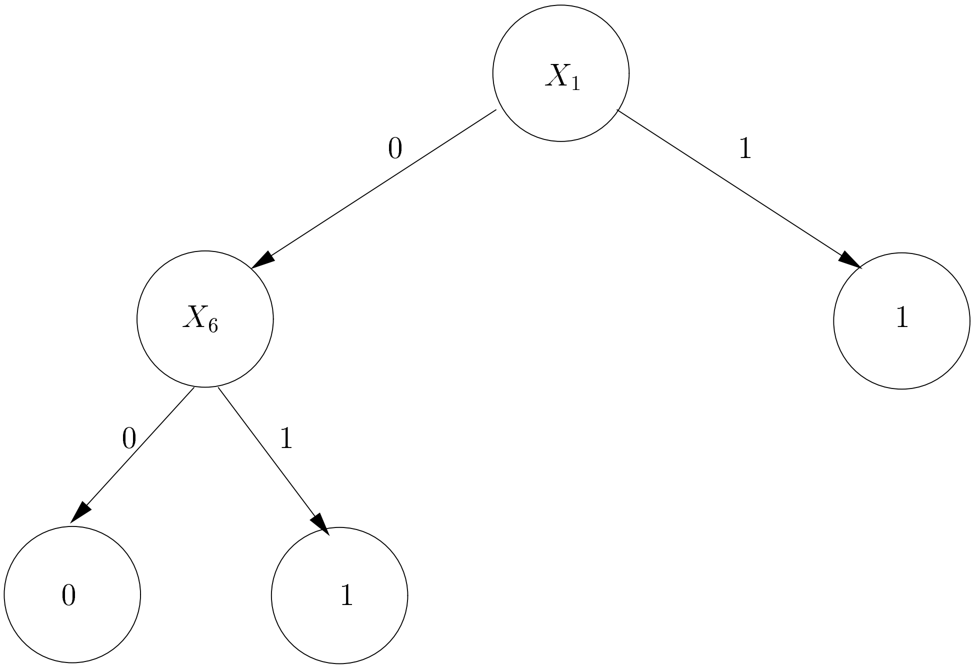

Figure 5.3: A Decision Tree Example

of k literals. As before, we can compute the value of each new literal in a given assignment. There are at most $(2n)^k$ such "new literals". The input can be easily transformed in time $k(2n)^k$. We then run our DL learning algorithm on the new input. Our resulting DL over the new literals is a k-DL over the original ones. Our sample size will be $m = O((2n)^k \log \frac{(2n)^k}{\varepsilon} + \frac{1}{\varepsilon} \ln \frac{1}{\delta})$.

5.4.2 Learning Decision Trees Using k-DL

A decision tree is a generalization of a decision list. Instead of using the structure of a list, we can use any rooted tree. Each internal node holds a literal, and has 2 outgoing edges (marked 0 and 1). Each leaf is assigned a value of 0 or 1 according to its classification. To find the value of a tree on a given assignment, we start at the root and follow the edges marked with the actual assignment of the internal node's literals until we arrive at a leaf, which holds our classification (value). Figure 5.3 shows an example of such a tree.

Theorem 5.1 *Let T be a decision tree with r nodes. Then there exists a k-DL, with* $k = \log(r+1)$ *that computes T.*

Proof: In T there must be a leaf of depth at most $\log(r+1)$, since if such a leaf does not exist, the size of T is at least r+1, which contradicts to our assumption. This leaf can be represented as a conjunction of the $\log(r+1)$ literals on the path from the root to it. This conjunction can be placed as a node in the $\log(r+1)$-DL. Removing the leaf, we are left with a DT of size r-1. We continue to add leaves to the DL until the tree is empty. □

Having proved the theorem, we can learn a DT using the algorithm we described for a k-DL. The running time will be $O(n^{\log r})$ where n is the number of attributes, and r is the tree size.

5.5 Decision Trees

In section 5.4.2 we saw the basic structure of a decision tree. We will now generalize this structure.

Definition A *Decision Tree* is a connected tree, in which each internal node holds a predicate, and a leaf holds classification. To find the classification of an input, we start at the root, and recursively descend down the tree, applying the node's predicate to the input and choosing the correct edge accordingly, until we reach a leaf. The classification of the leaf is the classification of the input.

An example of a generalized decision tree is in figure 5.4.

Given a basic class of hypotheses H, and a sample of examples, we would like to build a decision tree whose internal nodes are predicates in H and whose leaves are classification values, such that:

- the tree classifies all (or most) the examples correctly
- the tree is small

By Occam Razor, these two qualities imply PAC-learnability of decision trees.
H will typically be a set of simple predicates. For example, in Figure 5.4, $H = \{x_i > \alpha | \alpha \in R\}$ and the classifications are +1 and 0.

There are several reasons for the popularity of decision trees. There are efficient algorithms for both construction of decision trees and classification. Their performance is comparable to other methods used in Machine Learning. Moreover, there are software packages that support decision trees, e.g. C4.5, C5 and CART.

5.5.1 Construction of Decision Trees

The general idea of building a decision tree is to grow it from the root to the leaves by repeatedly replacing an existing leaf with an internal node. This top-down approach naturally

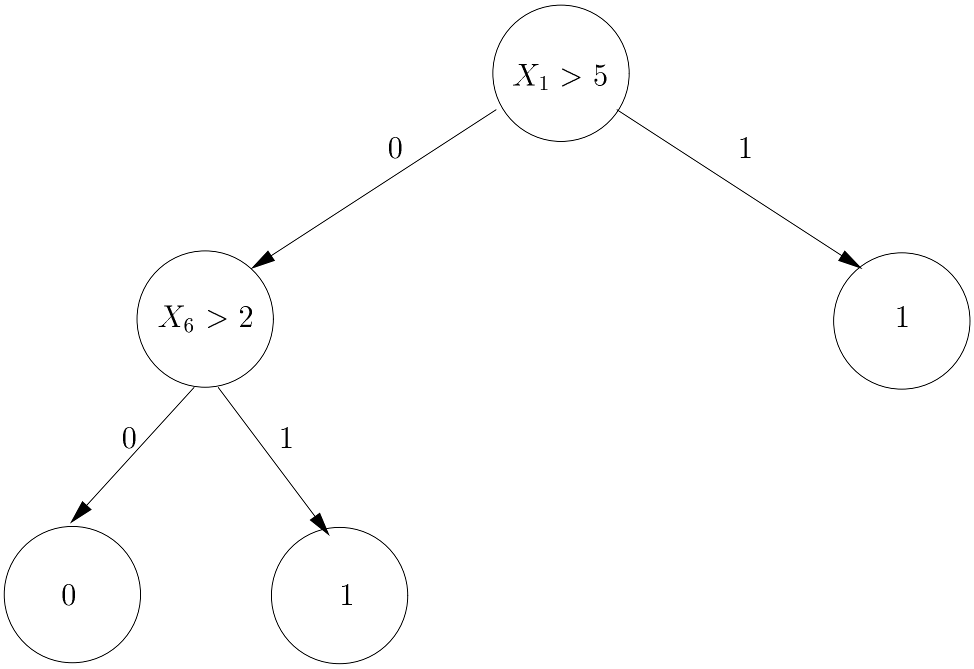

Figure 5.4: A Generalized Decision Tree Example

fits into a recursive procedure:
We choose a"good" predicate $h \in H$ to place at the root and split the examples using h. Then we build the left subtree (for $\{x|h(x) = 0\}$) and right subtree (for $\{x|h(x) = 1\}$) by recursive calls with the appropriate subsets of examples (We assume that the classifications are 0 and 1). The recursion terminates when all the examples have the same classification. The running time of the procedure for a tree of size s is $T(s)$, where
$T(s) = O(s) + T(s^+) + T(s^-)$
$s^+ = |\{x|h(x) = 1\}|$,
$s^- = |\{x|h(x) = 0\}|$
which turns out to be $T(s) = O(s \log s)$ (but may be as bad as $O(s^2)$ if the splits are no so good).

5.5.2 Selecting a Predicate

It remains to show how to select a predicate h for the root. We use the notations in Figure 5.5: v is the root, f is the target function and h is the predicate at the root, v_1 and v_2 are the left and right subtrees, respectively, q is the probability that f is 1 on an example in the root, p is the probability that f is 1 on an example in v_1, and r is the probability that f is 1 on an example in v_2. If we calculate the probabilities from the actual set of examples, we

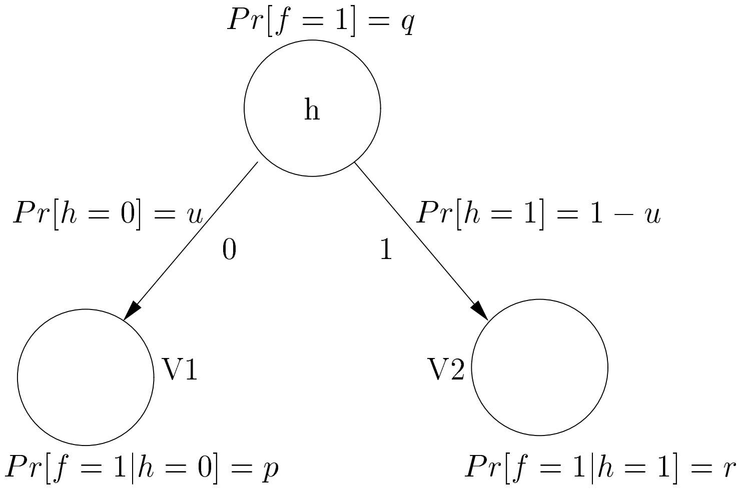

Figure 5.5: Notations for DT algorithm

get $q = up + (1 - u)r$ since the number of positive examples remains the same in each level of the tree (we just divide them between the 2 leaves).

In order to compare predicates we define a potential function, which on input q, u, p, r produces a value (hereafter *Potential*). We would like the potential to be an upper bound on the error. This way, decreasing the potential toward zero results in minimizing the error. For simplicity we consider only potential functions that are node-dependent. Such functions assign a value for each node (predicate). Given a split of node v, we define:
$val(v) = u\ val(v_1) + (1 - u)\ val(v_2)$.
In general, the potential of a tree is the sum of the potential of its leaves weighted by their probability.

Potential As Classification Error

First let us consider taking the classification error as the potential function. Namely:
$val(v) = \min\{q, 1 - q\}$
If we classify v using majority rule, this is exactly the probability of a classification error. In this case, averaging the potential over the tree gives the classification error for it. It is clear that the average potential can only drop as the algorithm proceeds because of the explicit minimization of $val(v)$. We terminate the algorithm when the average potential is zero, which is equivalent to the case of perfect classification.

Is this a good potential function? lets look at the example in figure 5.7. The potential of the tree before the split (v is the only node, and hence a leaf) is:

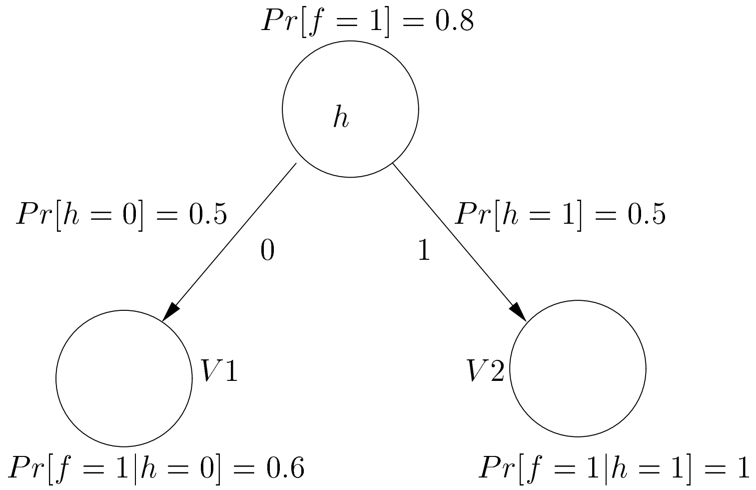

Figure 5.6: Classification Error As Potential Example

$val(v) = \min\{0.8, 1-0.8\} = 0.2$.
After the split, the potential is:
$val(v) = 0.5val(v_1) + 0.5val(v_2) = 0.5\min\{0.6, 1-0.6\} + 0.5\min\{1, 1-1\} = 0.2$.
Thus, the potential did not change.
Looking at the split more closely, we can see that after the split, all the examples in leaf v_2 are classified correctly. These examples constitute half of the examples. Basically speaking, this is a very good split, but there is no decrease in the potential. Hence, the potential function we chose does not correctly describe the value we gain from a split of a node, and cannot serve (well) as a tool for deciding which predicate we should use.

Requirements From Potential Functions

Our last example motivates us to define the requirements from a "good" potential function. We will intuitively require the potential P to comply with 2 rules:

1. $P = 0 \quad \Rightarrow \quad$ Perfect Classification

2. P is (strictly) convex

The first requirement is trivial. The second is placed in order to solve the problem we saw in the example above. When a node v is split into 2 nodes v_1 and v_2, the new value of the tree is the (weighted) average over the values of v_1 and v_2. If the potential is covex, every split causes a decrease in the potential, as can be seen in figure 5.7. if we go back to the example above, $\min\{q, 1-q\}$ is not strictly convex, causing the "anomaly" we saw.

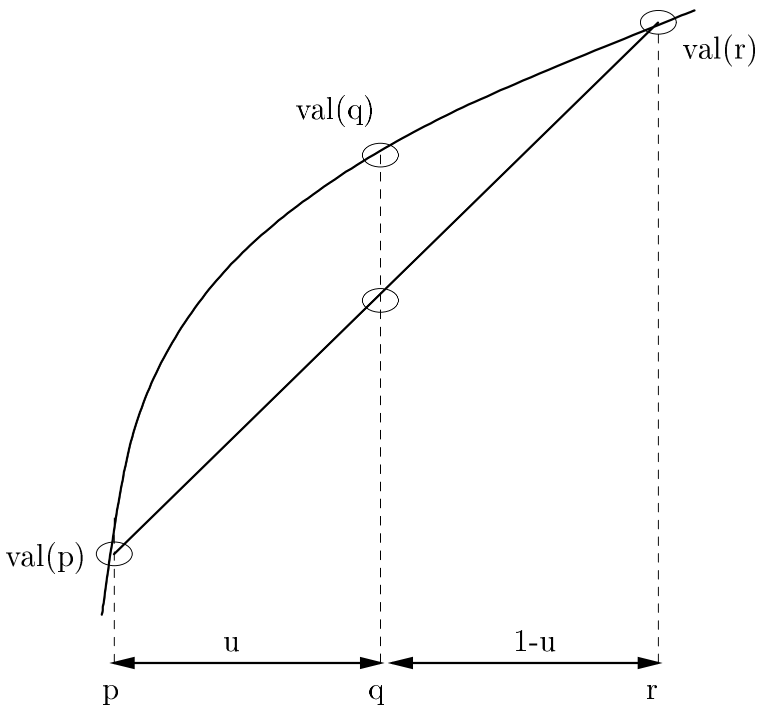

Figure 5.7: A convex potential energy ensures that a split will decrease the potential

Potential Functions: Candidates

In the following list are potential functions that are frequently used:

Ginni(q)	$val(q) = 2q(1-q)$	Used by CART software package
Entropy(q)	$val(q) = -q\log(q) - (1-q)\log(1-q)$	Used by C4.5 software package
New function	$val(q) = \sqrt{2q(1-q)}$	

These potential functions satisfy our previous requirements, as well as some additional ones:

- Symmetry: $val(q) = val(1-q)$
- $val(0) = val(1) = 0$
- $val(0.5) = 1$

5.5.3 The Decision Tree Construction Algorithm

Now we are able to present the algorithm for DT construction.

Algorithm: Decision Tree DT(S)

1. Input: A sample S of m examples.
2. If all examples in S have the same classification - b, return a leaf classified as b.
3. For each predicate h, compute $val(h,S) = u_h val(p_h) + (1-u_h)val(r_h)$ (This is the potential after a split using predicate h).
4. Let $h' = argmin_h\{val(h,S)\}$ (h' is the predicate that results in the smallest potential after the split).
5. Split S using h', resulting in S_0 and S_1.
6. Recursively call DT(S_0) and DT(S_1).

Analysis of the algorithm

In our analysis of the algorithm, we will use the notation:
$val(T) = \Sigma_{v\in L_T} Pr[v]val(q_v)$ $\quad$ L_T = Set containing all leaves of T
where $Pr[v]$ is the *actual* probability that example reaches leaf v, and q_v is the *actual* probability of such an example to be positive (we use actual probabilities for simplicity reasons). Our potential gives a bound on the classification error ($Error(t) \leq val(T)$), hence we can study how fast $val(T)$ drops as the algorithm proceeds. We will preform our analysis on a

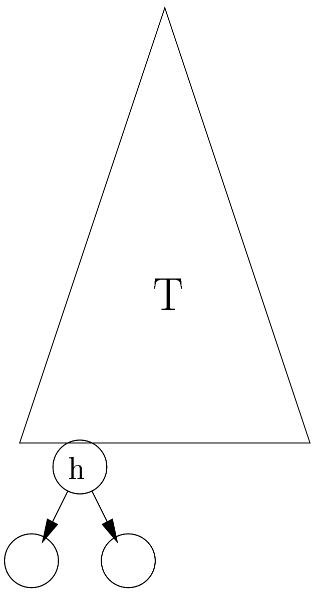

Figure 5.8: T(l,h) where l is a leaf, and h is a predicate

slightly revised algorithm - the *Top-Down Algorithm.* This algorithm works as our original algorithm, and produces the same tree. The difference between the two is the order in which the nodes are added to the tree. Given a leaf l of a tree T and a predicate h, we will use the notation T(l,h) for the resulting tree after expanding leaf l of T using predicate h. T(l,h) is schematically shown in figure 5.8.

Algorithm: Top Down DT

1. Input: H - set of predicates, s - size of tree.
2. let T_0 be a single leaf tree.
3. for t from 1 to s do:
   (a) let $(l,h) = argmax_{(l,h)}\{val(T_t) - val(T_t(l,h))\}$
   (l is the leaf achieving the maximum potential decrease among all leaves and h is the predicate achieving the maximum potential decrease in this leaf).
   (b) $T_{t+1} = T_t(l,h)$

As mentioned, the *Top Down* algorithm "expands" the same leaves using the same predicates as the original algorithm (but does so in a different order - in each step it chooses the best leaf and predicate.
In our discussion, we assume that H satisfies the weak learning hypothesis:

for each distribution D (of examples) there is an $h \in H$ such that $error(h) < \frac{1}{2} - \gamma$ (where $\gamma > 0$).
We show that in each step, there is a large drop in $val(T)$. The results are weaker than those of *AdaBoost* (but the algorithm is not intended to transform weak learning into strong learning).

We will modify our initial distribution of examples to be unbiased. In our analysis, we will use the Ginni index:
$val(q) = 2q(1-q)$

Lemma 5.2 *at every stage t there is a leaf l such that:*

1. $Pr[l] \geq \frac{\varepsilon_t}{2t}$
2. $error(l) = \min\{q_l, 1-q_l\} \geq \frac{\varepsilon_t}{2}$

   where ε_t is the error at stage t

Proof: Leaves that have a probability less than $\epsilon_t/2t$ (meaning they don't fulfil property 1) contribute no more than $\epsilon_t/2$ to the prediction error of T since there are exactly $t+1$ leaves at step t.
Leaves that have error less than $\epsilon_t/2$ (meaning they don't fulfil property 2) also contribute no more than $\epsilon_t/2$ to the prediction error of T.
Therefore, if no leaf satisfies both properties, the total error of the tree T would be less then ϵ_t. □

Let l be a leaf satisfying the conditions of the lemma. By our weak learning assumption, there is an h that achieves a drop of at least $Pr[l]\gamma^2[q_l(1-q_l)]^2$ in the potential after the split of leaf l. Thus, the potential decrease of leaf l is at least:
$Pr[l]\gamma^2[q_l(1-q_l)]^2 \geq \frac{\gamma^2\varepsilon_t^2}{t}$
We notice that for Ginni index, the following observations holds:

- $val(q) = 2q(1-q)$
- $q \geq q(1-q) \geq \frac{val(q)}{2}$

hence the reduction in potential is at least $O(\frac{\gamma^2[val(q_t)]^3}{t})$.
Since our algorithm chooses a leaf-hypothesis combination that achieves the largest reduction in potential, we have at least that reduction in every step.
We need to find the point where the potential drops below the desired ε (the error drops bellow ε too). We are looking for the smallest k for which $val(T_k) < \varepsilon$. Solving this yields the algorithm's time complexity: $O(\frac{1}{\varepsilon^2\gamma^2})$.

discussion

While we were able to show a very good bound on the complexity of *AdaBoost*, our bound in the DT learning algorithm (using Ginni Index) was exponential. In practice, the results are quite similar. The huge difference between theory and practice raises an obvious question. The reason for the difference lies in the γ parameter. While this parameter drops quite rapidly in *AdaBoost* (thus slowing down the process), in DT learning this is not necessarily the case.