Lecture 7

Divide and Conquer – Classification Using Decision Trees and Rules

Why decision trees?

Many classifiers, including Naïve Bayes and k-NN, expect all attribute values to be presented at the same time. Such a scenario, however, is not always possible. For example, a physician seeking to analyze a patient's condition often has no more than few subjective symptoms to begin with. Then, to narrow down possible diagnoses, the physician prescribes lab tests, and, based on the results, orders new tests, and so on. At any given moment, the doctor considers only a limited number of "attributes" that promise to add meaningful information or enhance understanding. It is not feasible to ask for all possible lab tests (thousands and thousands of them) right from the start in order to collect a full set of examples and attributes. In other words, an exhaustive set of examples and attributes might not be immediately available or might be unnecessary or costly. The classifier may do better by choosing the attributes one at a time, according to the demands of the situation.

The decision tree approach offers a solution to such situations.

Classification

Decision tree induction is the learning of decision trees from class-labeled training data. A **decision tree** is a flowchart-like tree structure, where each **internal node** (nonleaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (or *terminal node*) holds a class label. The topmost node in a tree is the **root** node.

How are decision trees used for classification? Given a tuple of attribute values, ***X***, for which the associated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

Why are decision tree classifiers so popular? The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle multidimensional data. Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans. The learning and classification steps of decision tree induction are simple and fast. In general, decision tree classifiers have good accuracy. Decision tree induction algorithms have been used for classification in many application areas such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology. Decision trees are the basis of several commercial rule induction systems.

An Illustrative Example

Decision trees classify data instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the data instance (the data instance is represented as a tuple of attribute values), and each branch descending from that node corresponds to one of the possible values of this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute. The process is then repeated for the subtree rooted at the new node.

Let's consider the following illustrative example. The training set consists of *objects* that are described in terms of a collection of *attributes.* Each attribute measures some important feature of an object. For example, if the objects are Saturday mornings and the classification task involves the weather, attributes might be:

- outlook, with values {sunny, overcast, rain}
- temperature, with values {cool, mild, hot}
- humidity, with values {high, normal}
- wind, with values {true, false }

The training set is shown on the next slide. *P* and *N* stand for Positive and Negative, respectively.

An Illustrative Example *(cont.)*

Day	*Outlook*	*Temperature*	*Humidity*	*Wind*	*PlayTennis*
Dl	Sunny	Hot	High	False	N
D2	Sunny	Hot	High	True	N
D3	Overcast	Hot	High	False	P
D4	Rain	Mild	High	False	P
D5	Rain	Cool	Normal	False	P
D6	Rain	Cool	Normal	True	N
D7	Overcast	Cool	Normal	True	P
D8	Sunny	Mild	High	False	N
D9	Sunny	Cool	Normal	False	P
Dl0	Rain	Mild	Normal	False	P
Dl1	Sunny	Mild	Normal	True	P
Dl2	Overcast	Mild	High	True	P
Dl3	Overcast	Hot	Normal	False	P
Dl4	Rain	Mild	High	True	N

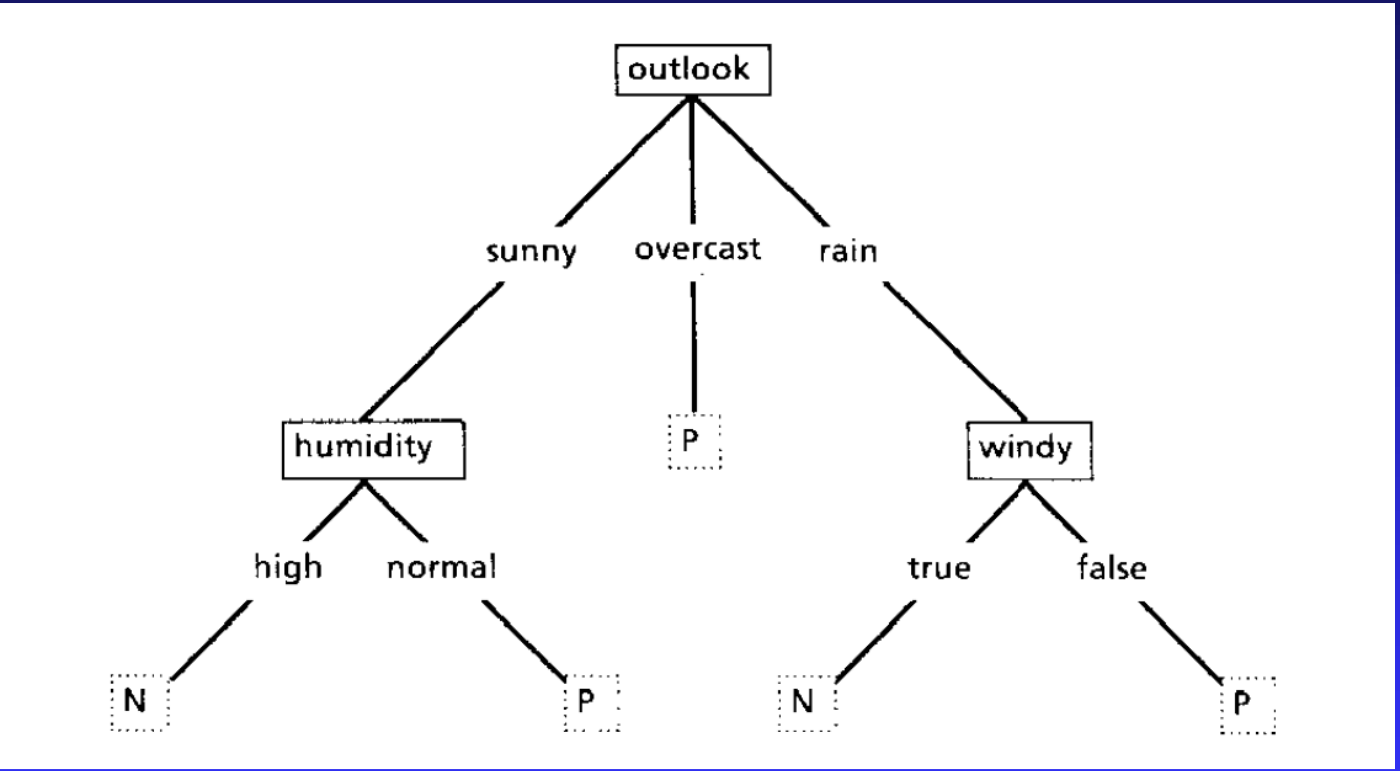
An Illustrative Example *(cont.)*

A particular Saturday morning might be described as: {Outlook = Overcast, Temperature = Cool, Humidity = Normal, Wind = False}.

The task is to learn the target attribute *PlayTennis*, which can have values *P* (for positive) or *N* (for negative) for different Saturday mornings, based on the other attributes of the morning in question. In other words, there are two classes, *Play Tennis on Saturdays* and *Does Not Play Tennis on Saturdays* represented as attribute values *P* and *N* for attribute *PlayTennis*.

The decision tree shown on the next slide classifies Saturday mornings according to whether they are suitable for playing tennis. For example, the instance *(Outlook = Sunny, Temperature = Hot, Humidity = High, Wind = True)* would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that *PlayTennis = no*). The leaves of a decision tree are class names, other nodes represent attribute-based tests with a branch for each possible outcome. In order to classify an object, we start at the root of the tree, evaluate the test, and take the branch appropriate to the outcome. The process continues until a leaf is encountered, at which time the object is asserted to belong to the class named by the leaf.

An Illustrative Example *(cont.)*



Problems appropriate for decision trees

- Instances are represented by attribute-value pairs: Instances are described by a fixed set of attributes (e.g., *Temperature*) and their values (e.g., *Hot*). The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., *Hot, Mild, Cold*).
- The target function has discrete output values: The decision tree in the previous example assigns a Boolean classification (e.g., *yes* or *no*) to each data instance. Decision tree methods easily extend to learning functions with more than two possible output values.
- The training data may contain errors: Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.
- The training data may contain missing attribute values*:* Decision tree methods can be used even when some training examples have unknown values. For example, it is not a problem to use the decision tree from the previous slide on the following data instance in which the humidity, the temperature and the wind of the day are unknown: *(Outlook = Overcast, Temperature = Hot, Humidity = ?, Wind = True)*

Divide and conquer

Decision trees are built using a heuristic called **recursive partitioning**. This approach is also commonly known as **divide and conquer** because it splits the data into subsets, which are then split repeatedly into even smaller subsets, and so on and so forth until the process stops when the algorithm determines the data within the subsets are sufficiently homogenous, or another stopping criterion has been met.

The root of the tree always corresponds to the entire training set. In the previous example, the root, *Outlook*, corresponds to the whole data set of 14 data tuples. After the first attribute test, the original data set is partitioned onto tree subsets corresponding to the children nodes of *Outlook*. The node *Sunny* corresponds to the subset of data instances for which *Outlook* = *Sunny*. These are D1, D2, D8, D9 and D11. The node *Overcast* corresponds to the subset of data instances for which *Outlook* = *Overcast*. These are D3, D7, D12 and D13. Similarly, the node *Rainy* corresponds to the subset containing D4, D5, D6, D10, D14. After further partitioning, the subset corresponding to *Sunny* is further divided into a subset for which *Humidity* = *High* (the subset of D1, D2 and D8) and a subset for which *Humidity* =*Normal* (the subset of D9 and D11). The subset corresponding to *Rain* is further divided into a subset for which *Wind* = *True* (the subset of D6and D14) and a subset for which *Wind* = *False* (the subset of D4, D5 and D10).

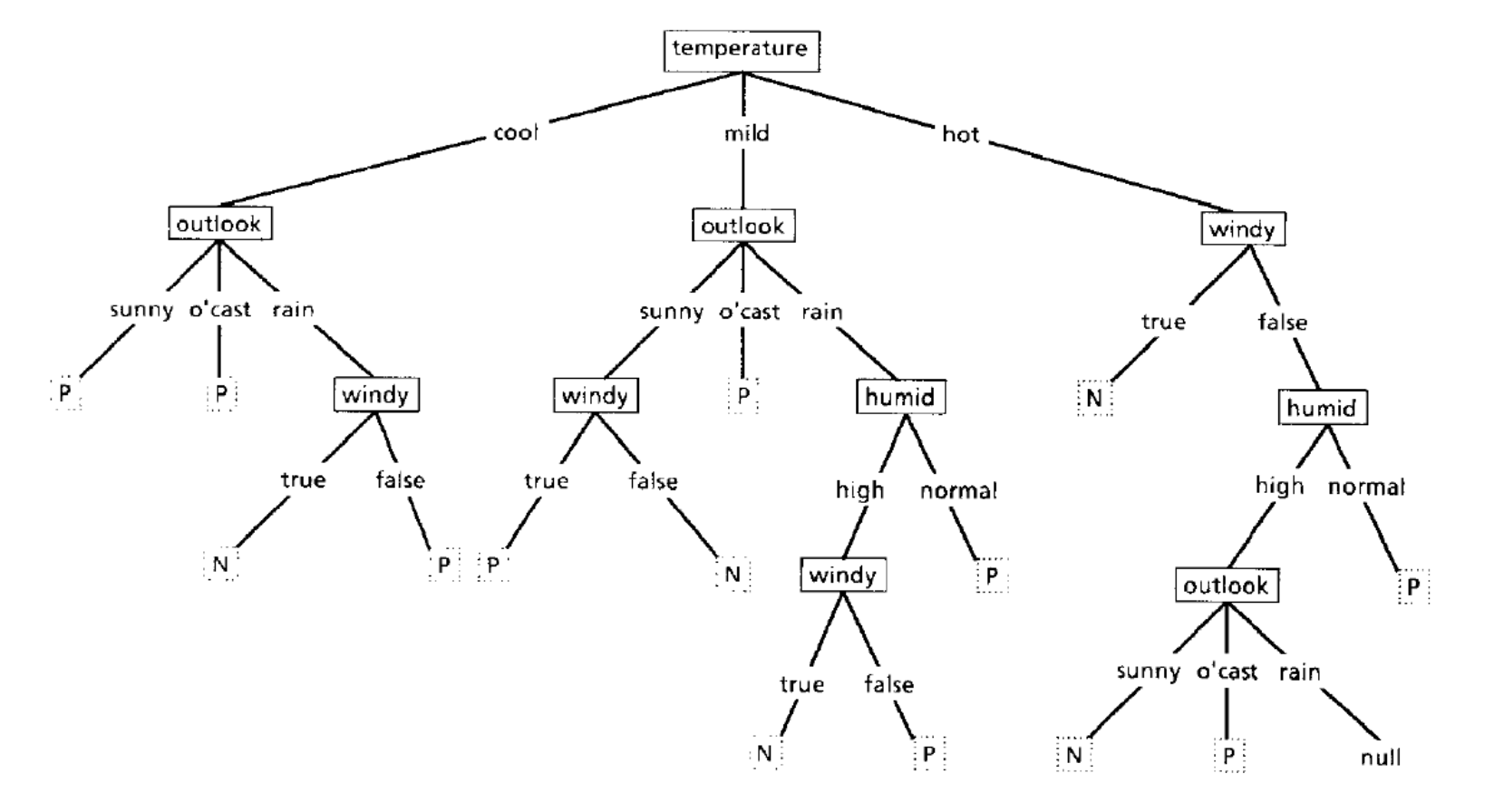
Why are small trees preferred?

Depending on the order of attribute tests, we can produce large or small decision trees for the same training data. For example, there is another, much larger, decision tree shown on the next slide, which uses the attribute tests in different order. The null leave node in the lower right corner means that the training set does not provide enough information to classifiy the node as positive or negative.

Why are small tests preferred?

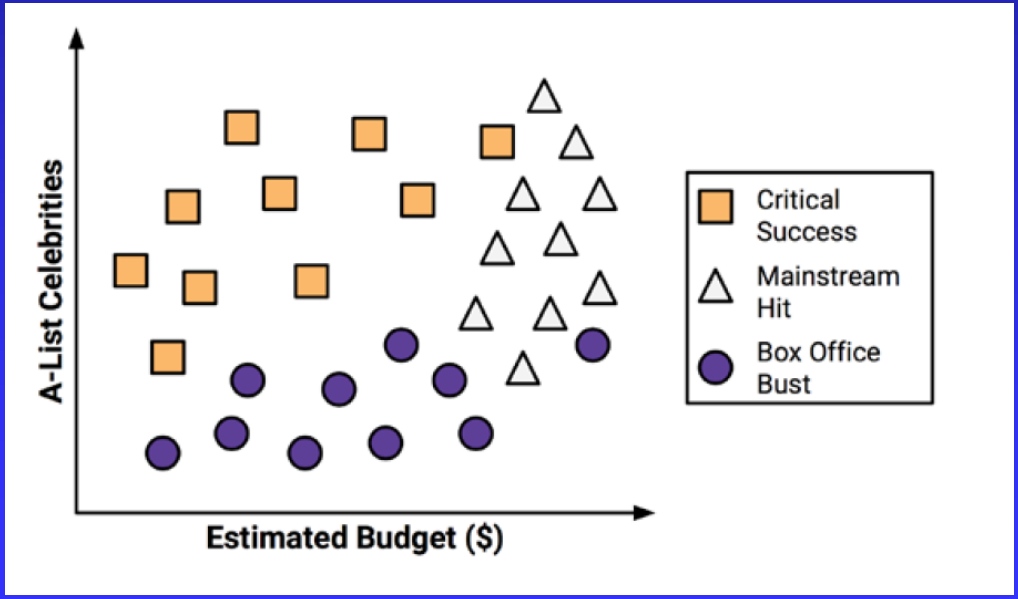
- It is easier for a human expert to analyze, explain, and perhaps even correct, a relatively small decision tree.
- Dispose of irrelevant and redundant information: smaller trees tend to get rid of irrelevant attributes. This is especially important for scenarios when attribute values are expensive or time-consuming to obtain.
- Larger trees are prone to overfitting the training examples. This is because the divide-and-conquer method might split the training set into very small subsets (with the number of these splits being equal to the number of attribute tests in the tree) so that the classes may get separated by an attribute that only by chance or noise has a different value.

Why are small trees preferred?



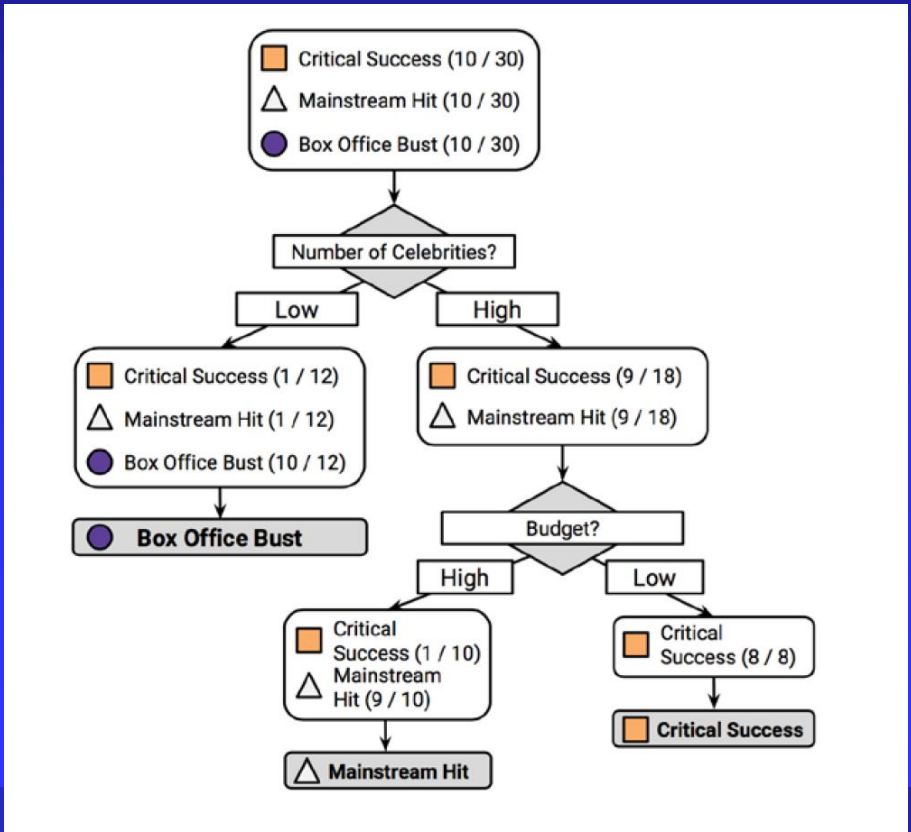
Another example: a decision tree for movies

Consider another example provided in your textbook. The training set consists of 30 movie titles, which have been labeled as *Critical Success, Mainstream Hit,* or *Box Office Bust.* The task is to build a decision tree which can predict the success of a movie based on two attributes: *List of Celebrities* and *Budget*. The movies from the training set are depicted on the following two dimensional plot:



Another example: a decision tree for movies (cont.)

Since there are two attributes, we can build only two decision trees. In the first tree, we can test *the List of Celebrities* attribute first, whereas in the second tree, we can test the *Budget* attribute first and the *List of Celebrities* attribute second. We decide to apply the List of Celebrities test first based on the observation that larger celebrities lists directly correlate with the movies' success, i.e., *List of Celebrities* is a better predictor of success than *Budget*:



Decision tree algorithms

During the late 1970s and early 1980s, J. Ross Quinlan, a researcher in machine learning, developed a decision tree algorithm known as **ID3** (Iterative Dichotomiser). This work expanded on earlier work on *concept learning systems*, described by E. B. Hunt, J. Marin, and P. T. Stone. Quinlan later presented **C4.5** (a successor of ID3) and its improvement **C5.0**, which became a benchmark to which newer supervised learning algorithms are often compared. In 1984, a group of statisticians (L. Breiman, J. Friedman, R. Olshen, and C. Stone) published the book *Classification and Regression Trees* (**CART**), which described the generation of binary decision trees. ID3 and CART were invented independently of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples. These two cornerstone algorithms spawned a flurry of work on decision tree induction.

ID3, C4.5, C5.0, and CART adopt a greedy (i.e., nonbacktracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built.

The order of attribute tests

The process of building a decision tree consists of applying the attribute tests in specific order. Basically, we find which attributes have not been tested so far at a given tree node and choose one of them. Then, we perform an attribute test for the chosen attribute, which splits the data set into further subsets that correspond to different children nodes of the given node. Therefore, the order of the attribute tests determines the "quality" and the shape of the decision tree. An **attribute selection measure** is a heuristic for choosing which attribute to test at a given node of the tree. Since we want to split data sets into smaller partitions, which are pure as we can make them (a partition is pure if all the tuples that fall into a given partition belong to the same class), our main goal is to choose a splitting (i.e., an attribute test) that provides maximum purity. The attribute selection measure provides a ranking for each attribute available at a given node of the tree. The attribute having the best score for the measure is chosen as the *splitting attribute* for the given node. The branches of the node are grown for each value of the selected attribute, and the data instances (tuples) are partitioned accordingly. The next following slides describe a popular attribute selection measure: *information gain.*

Information content of a message

Suppose we know that the training examples are labeled as pos or neg, the relative frequencies of these two classes being p_{pos} and p_{neg}, respectively. Let us select a random training example. How much information is conveyed by the message, "this example's class is pos"? The answer depends on p_{pos}. In the extreme case where all example are known to be positive, $p_{pos} = 1$, the message does not tell us anything new. You know that the example is positive even without being told so. The situation changes if both classes are known to be equally represented so that $p_{pos} = 0.5$. Here, the guess is no better than a flipped coin, so the message *does* offer some information. And if a great majority of examples are known to be negative, say, $p_{pos} = 0.01$, then you are all but certain that the chosen example is going to be negative; the message conveying the information that this is not the case you unexpected. And the lower the value of p_{pos}, the more information the message offers. In general, **the information content of a message is the amount of uncertainty it resolves**. The greater the surprise, the greater the information content is.

The following function have been used to qualify the information contents of such a message:

$I_{pos} = - \log_2 p_{pos}$

The negative sign compensates for the fact that the logarithm of $p_{pos} \in [0; 1]$ is always negative.

Information content of a message *(cont.)*

In general, if a message occurs with probability p, then the information content of the message is:

$I(p) = - \log_2 p$

Why this function? If we want to define a mathematical function that can measure the information content of a message, we would want our information measure I(p) to have several properties:

1. Information is a non-negative quantity: $I(p) > 0$.

2. If an event has probability 1, we get no information from the occurrence of the event:
$I(1) = 0$.

3. If two independent events occur (whose joint probability is the product of their individual probabilities), then the information we get from observing the events is the sum of the two pieces of information:
$I(p_1 + p_2) = I(p_1) + I(p_2)$:

4. We will want our information measure to be a continuous (and, in fact, monotonic) function of the probability (slight changes in probability should result in slight changes in information).

Information content of a message *(cont.)*

It can easily be proved that there is only one function satisfying the properties above.

$I(p) = \log_b(1/p) = - \log_b(p)$;

for some positive constant b. The base b determines the units we are using to measure information content.

The function $I(p) = - \log_2(p)$ uses base 2 and measures information content in bits.

Entropy: average information content

Suppose that the experiment is repeated many times and both messages will occur, "the example is positive," and "the example is negative," the first with probability p_{pos}, the second with probability p_{neg}. The average information content of all these messages is then obtained by the following formula where the information content of either message is weighted by its probability (the *S* in the argument refers to the data set of both messages):

$E(S) = - p_{pos} \log_2 p_{pos} - p_{neg} \log_2 p_{neg}$

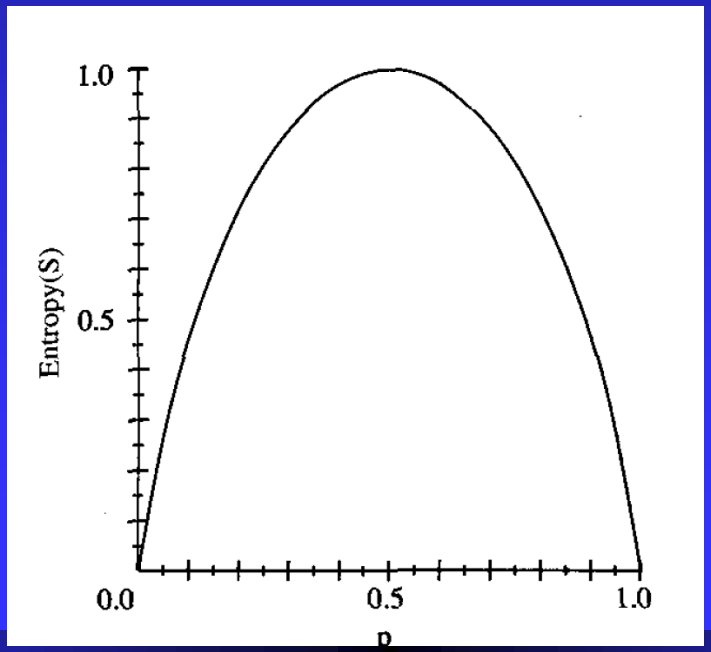
E(*S*) is called *entropy* of the message set *S*. Its value reaches its maximum, *E*(S) = 1, when $p_{pos} = p_{neg} = 0.5$, and it drops to its minimum, *E*(S) = 0, when either $p_{pos} = 1$ or $p_{neg} = 1$.

To illustrate, suppose S is the collection of the 14 Saturday mornings examples. S represents a Boolean concept, PlayTennis), which includes 9 positive and 5 negative examples. Then the entropy of S is:

$$E(S) = -\frac{9}{14} * log_2 \frac{9}{14} - \frac{5}{14} * log_2 \frac{5}{14} = 0.94$$

Entropy: average information content *(cont.)*

Note that the entropy is 0 if all members of S belong to the same class. For example, if all members are positive (p_{pos} = I), then E(S) = -1 * $\log_2 1$ - 0 * $\log_2$ 0 = 0 - 0 = 0 (we use the fact that $\log_2 1 = 0$ and that $0 * \log_2 0 = 0$, using the L'Hopital's rule). Note that the entropy is 1 when the set S contains an equal number of positive and negative examples. If the data set contains unequal numbers of positive and negative examples, the entropy is between 0 and 1. The following figure shows the form of the entropy function relative to a Boolean classification, as p, varies between 0 and 1.



Entropy: average information content *(cont.)*

So far we have discussed entropy in the special case where the classification is Boolean. More generally, if the target attribute can take on ***c*** different values, i.e., we have ***c*** different classes, then the entropy of a set of examples *S* is:

$E(S) = \sum_{i=1}^{c} -(pi * log_2 p_i)$

where p_i is the proportion of data instances belonging to class ***i.***

Since we want to have data sets containing only a singe class at the leaves of our decision trees, we will introduce the following concept: the degree to which a set of examples contains only a single class is known as **purity**, and any set composed of only a single class is called **pure**.

The definition of entropy suggests that it can be used as a measure of purity. Sets with high entropy are very diverse and include examples of various classes, i.e., there is high uncertainty about the class membership of a randomly selected item from the class. In contrast, sets with low entropy are homogeneous and include a small number of classes. A zero entropy means that the set includes only one class, i.e., there is no uncertainty about the class membership of a randomly selected item from the set.

Information gain from testing an attribute

Given entropy as a measure of the impurity of a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called ***information gain,*** is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, *Gain(S, A)* of *an* attribute ***A***, is defined as:

Gain(S, A) = E(S) – E(S, A)

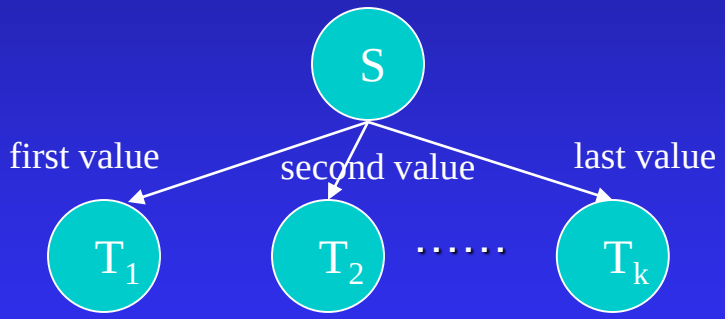
Where *E(S)* is the entropy before applying the attribute test and *E(S, A)* is the entropy after applying the attribute test. The higher the information gain, the better an attribute is at creating homogeneous subsets after a split on this attribute. If the information gain is zero, there is no reduction in entropy for splitting on this attribute, i.e., the newly produced subsets remain relatively diverse in terms of the classes they include. On the other hand, the maximum information gain is equal to the entropy prior to the split. In this case, the resulting subsets will be pure, i.e., each of them will include a single class.

Information gain from testing an attribute *(cont.)*

We know how to compute E(S), but how to compute E(S, A) in the formula:

Gain(S, A) = E(S) – E(S, A)?

Remember that an attribute *A* divides the example set, *S*, into subsets T_i each characterized by a different value of the attribute *A*. Quite naturally, each subset T_i will be marked by its own probability P_i (estimated by relative frequency) of the subset T_i.



Let $|T_i|$ be the number of examples in T_i, and let |S| be the number of examples in the whole set *S*. The probability that a randomly drawn training example will be in T_i is estimated as follows:

$$P_i = \frac{|Ti|}{|S|}$$

Information gain from testing an attribute *(cont.)*

Then, the entropy after the attribute test E(S, A) is the weighted average of the entropies of the subsets:

$E(S, A) = \sum_{i=1}^{k} P_i * E(Ti)$

If we substitute $\sum_{i=1}^{k} P_i * E(Ti)$ for E(S, A) in the formula of information gain, we obtain:

$Gain(S, A) = E(S) - \sum_{i=1}^{k} P_i * E(Ti)$

Information gain cannot be negative, i.e., information can only be gained, never lost, by splitting the example set using an attribute *A*.

Let's illustrate the concept of information gain using an example. Suppose *S* is the set of the 14 Saturday mornings training examples. Let's compute the information gain of applying attribute *Wind,* which can have the values *True* or *False.* In general, S includes 9 positive and 5 negative examples of the target concept *PlayTennis*. 6 of the positive and 2 of the negative examples have *Wind = False,* and the remainder have *Wind = True.* The information gain due to sorting the original ***14*** examples by the attribute ***Wind*** may then be calculated as:

Information gain from testing an attribute *(cont.)*

Values of Wind = True, False

S = [9+, 5-]

$S_{Wind=False}$ = [6+, 2-]

$S_{Wind=True}$ = [3+, 3-]

Gain(S, Wind) = *E(S)* – $\sum_{i=1}^{2} P_i * E(Ti)$

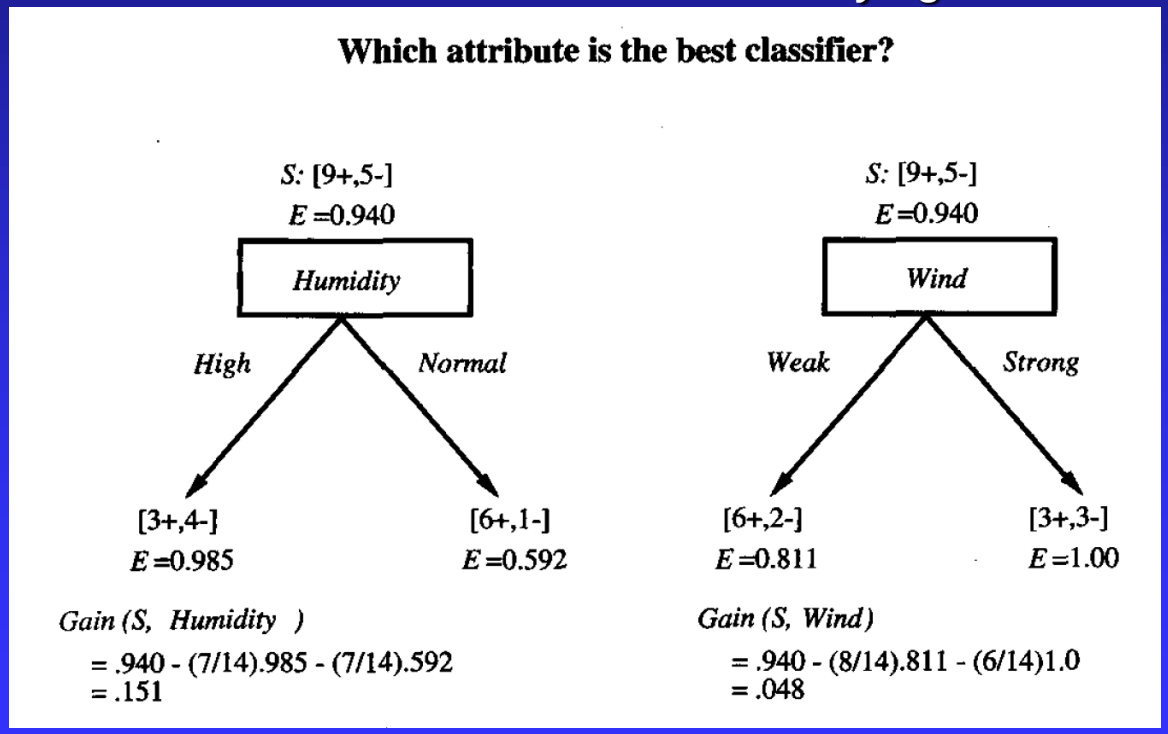
We know that E(S) = 0.94. Therefore,

Gain(S, Wind) = 0.94 – $\sum_{i=1}^{2} P_i * E(Ti)$

The proportion P_1 of the subset T_1 for which *Wind=False* is $\frac{8}{14}$ since we have 8 tuples for which *Wind=False*, and 14 tuples in total. The proportion P_2 of the subset T_2 for which *Wind=True* is $\frac{6}{14}$. Therefore, *Gain(S, Wind) = 0.94* – $P_1 * E(T_1) - P_2 * E(T_2)$ = $0.94 - \frac{8}{14} * E(T_1) - \frac{6}{14} * E(T_2)$ = *0.94* – $\frac{8}{14} * 0.811 - \frac{6}{14} * 1$ = 0.048

ID3

ID3, C4.5 and C5.0 all use **information gain** as its attribute selection measure. That is, at any given node of the tree, the information gain for all attributes that have not been tested so far is computed. The attribute with the highest information gain is selected as the new test attribute. The use of information gain to evaluate the relevance of attributes is summarized in the figure below, where the information gain of two different attributes, ***Humidity*** and ***Wind,*** is computed in order to determine which is the better attribute for classifying the training examples.



Humidity provides greater information gain than *Wind.*

ID3: example

To illustrate the operation of ID3, consider the learning task represented by the Saturday mornings training example. Here the target attribute ***PlayTennis,*** which can have values ***P*** (for positive) or ***N*** (for negative) for different Saturday mornings, is to be predicted based on other attributes of the morning in question. Consider the first step through the algorithm, in which the root node of the decision tree is created. Which attribute should be tested first in the tree? ID3 determines the information gain for each candidate attribute (i.e., Outlook, Temperature, Humidity, and Wind), then selects the one with highest information gain. The information gain values for all four attributes are:

Gain(S, Outlook) = 0.246

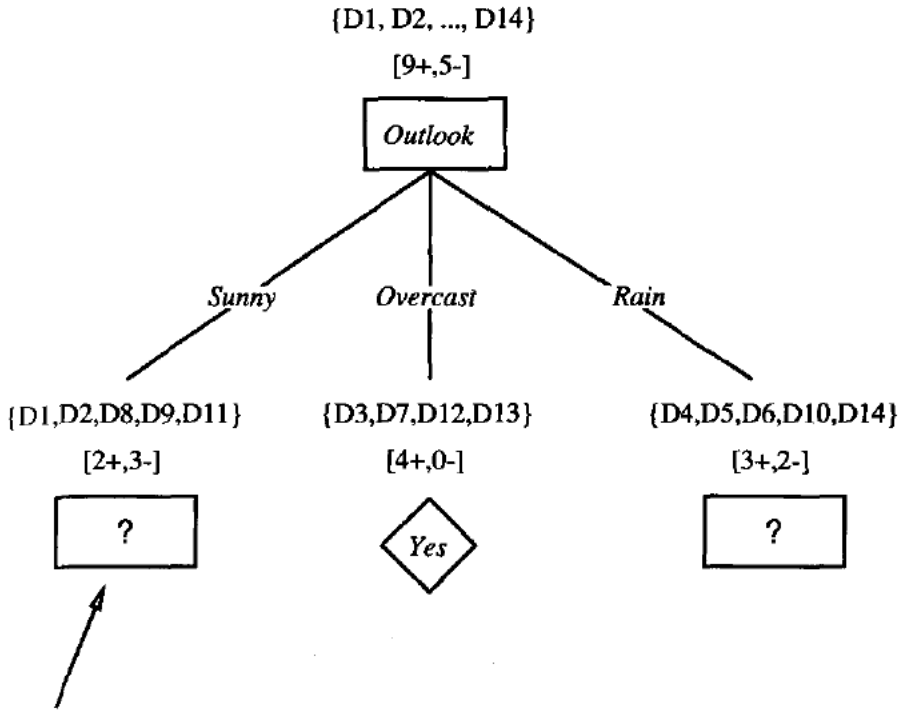
Gain(S, Humidity) = 0.151

Gain(S, Wind) = 0.048

Gain(S, Temperature) = 0.029

According to the information gain measure, the *Outlook* attribute provides the best prediction of the target attribute, *PlayTennis,* over the training examples. Therefore, *Outlook* is selected as the decision attribute for the root node, and branches are created below the root for each of its possible values (i.e., *Sunny, Overcast,* and *Rain).* The resulting partial decision tree is shown on the next slide along with the training examples sorted to each new descendant node.

ID3: example *(cont.)*



Which attribute should be tested here?

S_{sunny} = {D1,D2,D8,D9,D11}

Gain (S_{sunny}, *Humidity*) = .970 - (3/5) 0.0 - (2/5) 0.0 = .970

Gain (S_{sunny}, *Temperature*) = .970 - (2/5) 0.0 - (2/5) 1.0 - (1/5) 0.0 = .570

Gain (S_{sunny}, *Wind*) = .970 - (2/5) 1.0 - (3/5) .918 = .019

ID3: example *(cont.)*

Note that every example for which *Outlook* = *Overcast* is also a positive example of *PlayTennis.* Therefore, this node of the tree becomes a leaf node with the classification *PlayTennis* = *Yes.* In contrast, the descendants corresponding to *Outlook* = *Sunny* and *Outlook* = *Rain* still have nonzero entropy, and the decision tree will be further elaborated below these nodes.

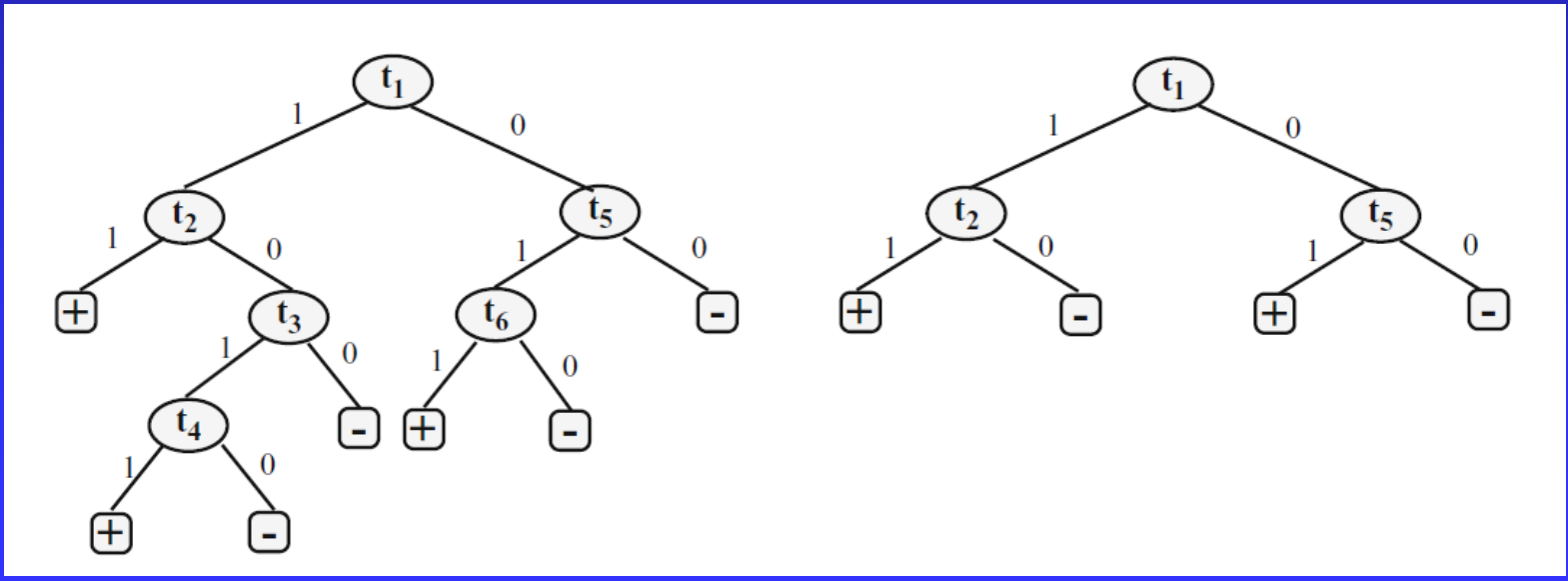
The process of selecting a new attribute and partitioning the training examples is now repeated for each nonterminal descendant node, this time using only the training examples associated with that node. Attributes that have been incorporated higher in the tree are excluded, so that any given attribute can appear at most once along any path through the tree. This process continues for each new leaf node until either of two conditions is met: (1) every attribute has already been included along this path through the tree, or (2) the training examples associated with this leaf node all have the same target attribute value (i.e., they belong to the same class and the entropy is zero).

Tree pruning

- We mentioned before that small decision trees are preferred over large decision trees for several reasons, the most important one being overfitting. There are two common approaches to tree pruning: *pre-pruning* and *post-pruning*. In the **pre-pruning** approach, a tree is “pruned” by halting its construction early (e.g., by deciding not to further split or partition the subset of training tuples at a given node). Upon halting, the node becomes a leaf. The leaf may hold the most frequent class among the subset tuples or the probability distribution of those tuples.
- The second and more common approach is **post-pruning**, which removes subtrees from a “fully grown” tree. A subtree at a given node is pruned by removing its branches and replacing it with a leaf. The leaf is labeled with the most frequent class among the subtree being replaced. Post-pruning is usually preferred over pre-pruning because it is quite difficult to determine the optimal depth of a decision tree without growing it first.

Tree pruning *(cont.)*

The figure below illustrates tree pruning. On the left is the original decision tree whose six attribute tests are named $t_1,\ldots,t_6$. On the right is a pruned version. Note that the subtree rooted in test t_3 in the original tree is in the pruned tree replaced with a leaf labeled with the negative class; and the subtree rooted in test t_6 is replaced with a leaf labeled with the positive class. The point is that pruning consists of replacing one or more subtrees with leafs, each labeled with the class most common among the training examples that reach the removed subtree in the original classifier.



Identifying risky bank loans with C5.0: exploring data

Decision trees are widely used in the banking industry due to their high accuracy and ability to formulate a statistical model in plain language. The idea behind our credit model is to identify factors that are predictive of higher risk of default. The dataset we will be using contains information on loans obtained from a credit agency in Germany. It was donated Hans Hofmann of the University of Hamburg to the UCI Machine Learning Data Repository (http://archive.ics.uci.edu/ml). You can also download the dataset from Blackboard and store it in your working directory.

```
> credit <- read.csv("credit.csv")
```

We will first explore the structure of the data frame *credit*. The str() command shows 1,000 examples with 17 attributes, which are a combination of factor and integer data types.

Identifying risky bank loans with C5.0: exploring data *(cont.)*

```
> str(credit)
'data.frame':   1000 obs. of  17 variables:
 $ checking_balance    : Factor w/ 4 levels "< 0 DM","> 200 DM",..: 1 3 4 1 1 4 4 3 4 3 ...
 $ months_loan_duration: int  6 48 12 42 24 36 24 36 12 30 ...
 $ credit_history      : Factor w/ 5 levels "critical","good",..: 1 2 1 2 4 2 2 2 2 1 ...
 $ purpose             : Factor w/ 6 levels "business","car",..: 5 5 4 5 2 4 5 2 5 2 ...
 $ amount              : int  1169 5951 2096 7882 4870 9055 2835 6948 3059 5234 ...
 $ savings_balance     : Factor w/ 5 levels "< 100 DM","> 1000 DM",..: 5 1 1 1 1 5 4 1 2 1 ...
 $ employment_duration : Factor w/ 5 levels "< 1 year","> 7 years",..: 2 3 4 4 3 3 2 3 4 5 ...
 $ percent_of_income   : int  4 2 2 2 3 2 3 2 2 4 ...
 $ years_at_residence  : int  4 2 3 4 4 4 4 2 4 2 ...
 $ age                 : int  67 22 49 45 53 35 53 35 61 28 ...
 $ other_credit        : Factor w/ 3 levels "bank","none",..: 2 2 2 2 2 2 2 2 2 2 ...
 $ housing             : Factor w/ 3 levels "other","own",..: 2 2 2 1 1 1 2 3 2 2 ...
 $ existing_loans_count: int  2 1 1 1 2 1 1 1 1 2 ...
 $ job                 : Factor w/ 4 levels "management","skilled",..: 2 2 4 2 2 4 2 1 4 1 ...
 $ dependents          : int  1 1 2 2 2 2 1 1 1 1 ...
 $ phone               : Factor w/ 2 levels "no","yes": 2 1 1 1 1 2 1 2 1 1 ...
 $ default             : Factor w/ 2 levels "no","yes": 1 2 1 1 2 1 1 1 1 2 ...
```

Identifying risky bank loans with C5.0: exploring data *(cont.)*

The *default* vector indicates whether the loan applicant was unable to meet the agreed payment terms and went into default. A total of 30 percent of the loans in this dataset went into default:

```
> table(credit$default)
 no yes
700 300
```

You can see applicants' savings and checking balances:

```
> table(credit$checking_balance)
    < 0 DM   > 200 DM 1 - 200 DM    unknown
       274          63         269         394
> table(credit$savings_balance)
     < 100 DM     > 1000 DM  100 - 500 DM 500 - 1000 DM        unknown
          603             48            103             63            183
```

The checking and savings balances are factors with 4 and 5 levels, respectively. The currency unit is Deutsche Mark (DM), the official German currency before the adoption of Euro in 2002. You can see that most people have checking balances below 200 DM and saving balances below 100 DM.

Identifying risky bank loans with C5.0: exploring data *(cont.)*

Other predictors of default are loan amounts and the duration of the loan:

```
> summary(credit$months_loan_duration)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    4.0    12.0    18.0    20.9    24.0    72.0
> summary(credit$amount)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    250    1366    2320    3271    3972   18420
```

The loan amounts ranged from 250 DM to 18,420 DM across terms of 4 to 72 months with a median duration of 18 months and an amount of 2,320 DM.

The next task is to choose a training and test sets. As usual, we will choose a training set that includes 90% of our original data, and a test set that includes the remaining 10%. As we know, both sets must be chosen randomly. Note that we cannot use the first 900 records as a training set (as we did in the previous lectires) because there is no guarantee that the records are stored in random order. Therefore, we need to create a random sample of size 900. The most convenient way to create the sample is by calling the *sample()* function, which uses a pseudo random number generator.

Identifying risky bank loans with C5.0: exploring data *(cont.)*

```
> sample(1:10, 1)
[1] 4
```

Here, the function returns one random number between 1 and 10. Every pseudo random number generator, PSRN, uses a seed that determines the sequence of the pseudo random number it generates. That is, if we start the PSRN with the same seed, it will produce the same sequence of pseudo random numbers

```
> set.seed(1)
```

Now, we can generate 5 random numbers in the range 1:10:

```
> sample(1:10, 5)
[1] 3 4 5 7 2
```

If we reset the seed once again to 1, we can repeat the same sample:

```
> set.seed(1)
> sample(1:10, 5)
[1] 3 4 5 7 2
```

Otherwise, the sample would be different from the first one. By default, the random sample does not include repetitions of the same number (this can be changed by calling *sample()* with attribute replace = TRUE).

Identifying risky bank loans with C5.0: exploring data *(cont.)*

In order to be able to repeat the same random sample from our dataset, we will set the seed of the pseudo random number generator:

```
> set.seed(123)
```

Then, we will create a vector *train_sample* of 900 random numbers between 1 and 1000:

```
> train_sample <- sample(1000, 900)
```

We will use the random vector *train_sample* as index selector to extract 900 records from our dataset:

```
> credit_train <- credit[train_sample, ]
```

We will use the random vector *train_sample* once again as index selector to create a test set by excluding (taken with minus sign) the previously selected 900 records from the dataset. The resulting test set will include the remaining 100 records:

```
> credit_test <- credit[-train_sample, ]
```

Note that the column index is blank because we select all columns for the specified rows.

If the split of the original dataset into training and test sets is random, the percentage of default loans (30%) should remain the same in both sets:

Identifying risky bank loans with C5.0: exploring data *(cont.)*

```
> prop.table(table(credit_train$default))
      no       yes
0.7033333 0.2966667
> prop.table(table(credit_test$default))
  no  yes
0.67 0.33
```

We will use the C5.0 algorithm in the C50 package to train our decision tree model:

```
> install.packages("C50")
> library(C50)
Warning message:
package 'C50' was built under R version 3.2.3
```

The *C5.0()* function takes several attributes that specify different options for the algorithms. The simplest way to run it is:

C5.0(train, class)

Where train is the training data frame and class is factor vector with the classes for each row in the training data.

Identifying risky bank loans with C5.0: exploring data *(cont.)*

We create a decision tree called *credit_model*:

```
>  credit_model <- C5.0(credit_train[-17], credit_train$default)
```

Note that we have excluded the 17th column from the training set because it includes default labels. We have also supplied the 17th column as a second attribute specifying the target factor for classification.

Now, we can take a brief look at the classification tree:

```
> credit_model

Call:
C5.0.default(x = credit_train[-17], y = credit_train$default)

Classification Tree
Number of samples: 900
Number of predictors: 16

Tree size: 57
Non-standard options: attempt to group attributes
```

The size of the decision tree, 57, indicates that it is 57 decision steps deep.

Identifying risky bank loans with C5.0: exploring data *(cont.)*

We can see the tree's decisions using the *summary()* function:

```
the tree is omitted because of its size
```

The tree can be read line by line. For example:

If the checking account balance is unknown or greater than 200 DM, then classify as “no default."

Otherwise, if the checking account balance is less than zero DM or between one and 200 DM.

And the credit history is perfect or very good, then classify as "default.“

And so on ….

The numbers in parentheses indicate the total number of examples that correspond to the given decision node and the number of the examples that are incorrectly classified at the decision node. For example, 412/50 on the first line indicate that there are 412 examples that meet the decision *checking_balance in {> 200 DM, unknown}.* 50 is the number of example that are incorrectly classified by the decision.

Identifying risky bank loans with C5.0: exploring data *(cont.)*

You can find the error rate and the confusion matrix at the bottom of the decision tree. In our case, 133 from 900 examples were incorrectly classified for an error rate of 14.8%. The rows of the confusion matrix correspond to actual classes, whereas columns correspond to predicted classes. According to the confusion matrix, 35 “no default” cases were misclassified as “default”. These are false positives. In addition, 98 “default” cases were misclassified as “no default”. These are false negatives.

The next step is to evaluate the performance of the decision tree by applying it to the test set:

```
> credit_pred <- predict(credit_model, credit_test)
```

We will compare the predicted values, credit_pred, with the actual ones, *credit_test$default*, by creating a crosstab:

```
> credit_pred <- predict(credit_model, credit_test)
> library(gmodels)     # gmodels package includes the CrossTable
                       # function
> CrossTable(credit_test$default, credit_pred,
+ prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
+ dnn = c('actual default', 'predicted default'))
```

Identifying risky bank loans with C5.0: exploring data *(cont.)*

```
    Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|
Total Observations in Table:  100
```

	predicted default		
actual default	no	yes	Row Total
no	59	8	67
	0.590	0.080	
yes	19	14	33
	0.190	0.140	
Column Total	78	22	100

Boosting the accuracy of decision trees

The crosstab shows that the decision tree correctly classified 59 “no default” examples and 14 “default” examples. 8 “no default” examples are misclassified as “default”. These are false positives. 19 “default” examples are misclassified as “no default”. These are false negatives. The total error rate is 27% and 58% of the “default” examples are misclassified as “no default”. In general, the performance of the decision tree is not satisfactory.

One way to improve our model is to use ***adaptive boosting***. The idea is to build many decision trees and to take a vote on the best class for each example. In general, boosting is based on the idea that that by combining a number of weak performing learners, you can create a team that is much stronger than any of the learners alone. Adaptive boosting (AdaBoost) is an improvement of boosting and it works in rounds. At each round, subsequent classifiers are added, which are tweaked in favor of those instances misclassified by previous classifiers. Hence the name *adaptive*. Note that adaptive boosting is not limited to decision trees only. It can be applied to a wide range of machine learning algorithms.

There is an adaptive boosting attribute in the *C5.0()* function called *trials*, which specifies the maximum number of decision trees to be built and used for voting.

Boosting the accuracy of decision trees *(cont.)*

We will run C5.0 again. This time, we will use 10 trials, i.e., a maximum of 10 decision trees:

```
> credit_boost10 <- C5.0(credit_train[-17], credit_train$default,
trials = 10)
> credit_boost10
Call:
C5.0.default(x = credit_train[-17], y = credit_train$default, trials
= 10)

Classification Tree
Number of samples: 900
Number of predictors: 16

Number of boosting iterations: 10
Average tree size: 47.5

Non-standard options: attempt to group attributes
```

Boosting the accuracy of decision trees *(cont.)*

We can see that 10 decision trees have been created and an average size of 47.5. The `summary()` function can be used to show all decision trees. Because of their size, we will skip them and will go directly to the confusion matrix:

```
   (a)   (b)    <-classified as
  ----  ----
   629     4    (a): class no
    30   237    (b): class yes
```

The confusion matrix shows 4 false positives and 30 false negatives, 34 errors in total. Keep in mind that these are errors on the training set. Let's test the new algorithm on our test set:

```
> credit_boost_pred10 <- predict(credit_boost10, credit_test)
> CrossTable(credit_test$default, credit_boost_pred10,
+ prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
+ dnn = c('actual default', 'predicted default'))
```

We reduced the total error rate from 27% down to 18% in the boosted model. On the other hand, the model is still not doing well at predicting defaults, predicting only 61% correctly.

Boosting the accuracy of decision trees *(cont.)*

```
   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|
Total Observations in Table:  100
               | predicted default
actual default |        no |       yes | Row Total |
---------------|-----------|-----------|-----------|
            no |        62 |         5 |        67 |
               |     0.620 |     0.050 |           |
---------------|-----------|-----------|-----------|
           yes |        13 |        20 |        33 |
               |     0.130 |     0.200 |           |
---------------|-----------|-----------|-----------|
  Column Total |        75 |        25 |       100 |
---------------|-----------|-----------|-----------|
```

Incorporating the cost of mistakes

Depending on the given case, different types of mistakes have different costs. In the credit scenario, the false negatives (i.e., classifying risky loan applicants as non risky) could be more costly than false positives (i., e., classifying non risky loan applicants as risky) because the missed opportunity of selling a loan to a non risky applicant could be far outweighed by the massive loss the bank would incur from a risky applicant who does not pay back his loan.

Fortunately, C5.0 can be fine tuned by assigning different weights to different types of mistakes. Suppose that false negatives cost the bank four times as much as false positives, i.e., loan default costs four times as much as a missed opportunity. This can be represented as a cost matrix:

	actual	
predicted	no	yes
no	0	4
yes	1	0

The zeroes correspond to correct predictions, 1 corresponds to false positive, and 4 corresponds to false negative. In other words, there is no cost assigned when the algorithm classifies a *no* or *yes* correctly, but a false negative has a cost of 4 versus a false positive's cost of 1.

Incorporating the cost of mistakes *(cont.)*

We will build such a matrix and will pass it to the C5.0 function as an optional attribute along with the attributes we have used so far. First, we need to create the matrix. The following function gives a name to each row and column of the 2*2 matrix:

```
> matrix_dimensions <- list(c("no", "yes"), c("no", "yes"))
```

Then, we use rows for predicted values and columns for actual values:

```
> names(matrix_dimensions) <- c("predicted", "actual")
```

Finally, we create a 2*2 matrix with entries 0, 4, 1, 0 using the names defined previously:

```
> error_cost <- matrix(c(0, 1, 4, 0), nrow = 2, + dimnames =
matrix_dimensions)
```

We can print the matrix:

```
> error_cost
         actual
predicted no yes
      no   0   4
      yes  1   0
```

Incorporating the cost of mistakes *(cont.)*

Now, we are ready to run C5.0 with a cost matrix:

```
> credit_cost <- C5.0(credit_train[-17],
credit_train$default,
+ costs = error_cost)
```

The new predictions are saved in *credit_cost_pred:*

```
> credit_cost_pred <- predict(credit_cost, credit_test)
```

As usual, we create a crosstab to compare actual with predicted values:

```
> CrossTable(credit_test$default, credit_cost_pred,
+ prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
+ dnn = c('actual default', 'predicted default'))
```

The number of false negatives is reduced significantly at the cost of increasing false positives.

Incorporating the cost of mistakes *(cont.)*

```
   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|
Total Observations in Table:  100
```

actual default	predicted default: no	predicted default: yes	Row Total
no	37	30	67
	0.370	0.300	
yes	7	26	33
	0.070	0.260	
Column Total	44	56	100

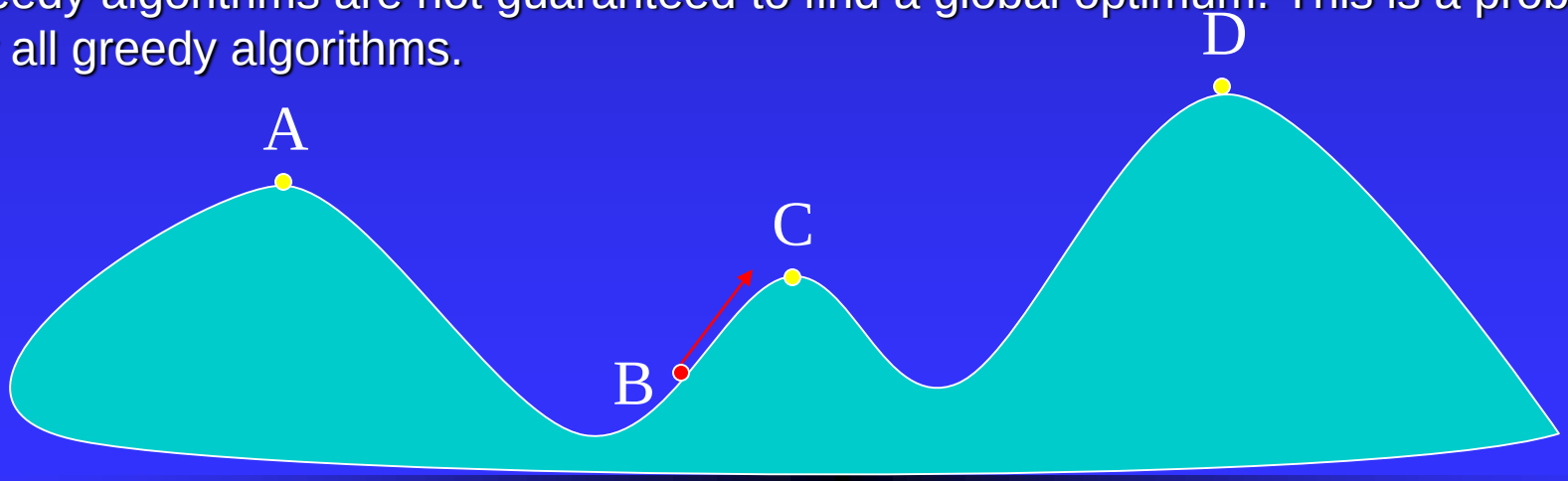
Why decision trees are greedy algorithms?

SKIP the rest of the chapter with the exception of the short (but important) section on page 158 called "What makes trees and rules greedy?"

Since many ML algorithms, including decision trees are greedy, it is important to know what "greedy algorithm" means. In general, an algorithm is greedy if it wants to find a globally optimal solution by taking locally optimal steps. The word "optimal" should be understood in broad sense as effective, short, accurate, etc. In other words, greedy algorithms do not optimize globally, but only locally, hoping to reach a globally optimal solution by taking a series of locally optimal steps. A greedy algorithm always takes the "best" step at any given moment, without realizing that what looks "best" at a given moment might not be "best" later down the road. Hence, the name "greedy": the algorithm does not want to forsake immediate gains for long-term gains. This is a myopic strategy in which the algorithm never looks ahead to see what is coming. Instead, it does it best using its current knowledge and never reconsiders its actions.

Why decision trees are greedy algorithms? *(cont.)*

Here is a typical example of a greedy strategy. An optimization algorithm is supposed to find the global maximum on the curve below by performing hill climbing, starting at point B. The curve has three maxima: A, C, and D. The maxima A and C are local and D is global. A greedy algorithm always goes up and never down. When the algorithm starts at point B, it will climb the hill on its right, following the direction of the slope. When it reaches point C it will stop there forever despite the fact that point D is better (higher). Reaching point B, however, is impossible for the greedy algorithm because it involves going down the hill, i.e., taking locally suboptimal steps. As a result, the greedy algorithm will remain stuck in a local optimum, without being able to reach the global optimum. That is, greedy algorithms are not guaranteed to find a global optimum. This is a problem for all greedy algorithms.



Strengths and weaknesses of decision trees

Strengths:

- Highly automatic learning process, which can handle numeric or nominal features, as well as missing data
- Excludes unimportant features
- Can be used on both small and large datasets
- Results in a model that can be interpreted without a mathematical background (for relatively small trees)

Weaknesses

- Decision tree models are often biased toward splits on features having a large number of levels
- It is easy to overfit or underfit Since it considers one attribute at a time, it cannot model complex relationships between attributes.
- Small changes in the training data can result in large changes to
- decision logic
- Large trees can be difficult to interpret and the decisions they make may seem counterintuitive