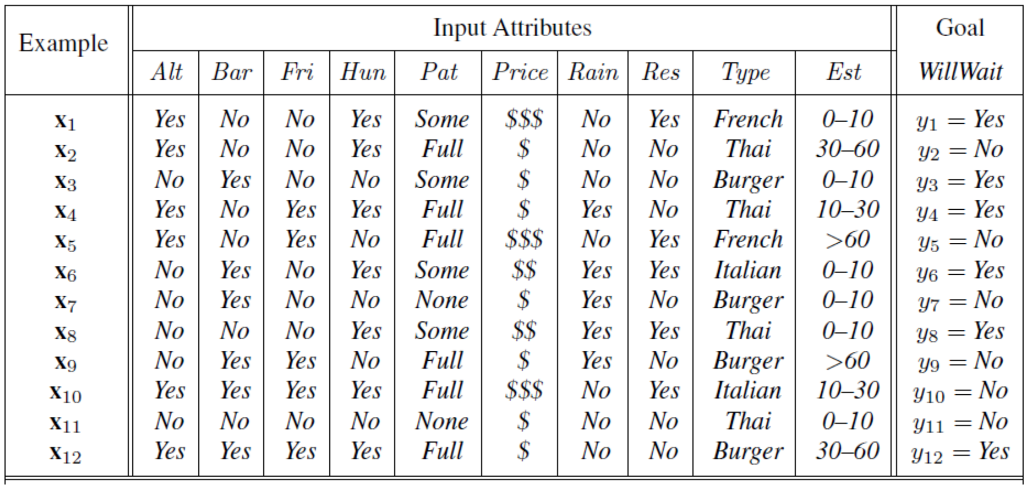
CS 6140: Machine Learning **Spring 2015**
College of Computer and Information Science
Northeastern University
Lecture 1 **January, 12**
Instructor: Bilal Ahmed **Scribe: Bilal Ahmed & Virgil Pavlu**

Decision Trees

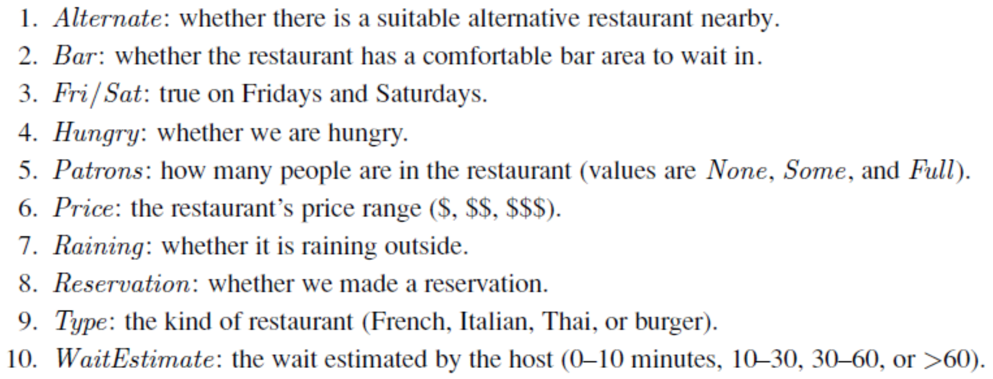
Assume we are given the following data:



Example	Input Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*\$\$\$*	*No*	*Yes*	*French*	*0–10*	$y_1 =$ *Yes*
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*\$*	*No*	*No*	*Thai*	*30–60*	$y_2 =$ *No*
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*\$*	*No*	*No*	*Burger*	*0–10*	$y_3 =$ *Yes*
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*\$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 =$ *Yes*
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*\$\$\$*	*No*	*Yes*	*French*	*>60*	$y_5 =$ *No*
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*\$\$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 =$ *Yes*
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*\$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 =$ *No*
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*\$\$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 =$ *Yes*
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*\$*	*Yes*	*No*	*Burger*	*>60*	$y_9 =$ *No*
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*\$\$\$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} =$ *No*
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*\$*	*No*	*No*	*Thai*	*0–10*	$y_{11} =$ *No*
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*\$*	*No*	*No*	*Burger*	*30–60*	$y_{12} =$ *Yes*

Figure 1: Training data for the *wait-for-table* classification task.

The task at hand is to develop an application that advises whether we should wait for a table at a restaurant or not. To this end we are going to assume that the given data represents the true concept *WillWait* and we will further assume that all future data will be described by the same input attributes *aka* features. A description of these features is provided in Figure-2.



1. *Alternate*: whether there is a suitable alternative restaurant nearby.
2. *Bar*: whether the restaurant has a comfortable bar area to wait in.
3. *Fri/Sat*: true on Fridays and Saturdays.
4. *Hungry*: whether we are hungry.
5. *Patrons*: how many people are in the restaurant (values are *None*, *Some*, and *Full*).
6. *Price*: the restaurant's price range (\$, \$\$, \$\$\$).
7. *Raining*: whether it is raining outside.
8. *Reservation*: whether we made a reservation.
9. *Type*: the kind of restaurant (French, Italian, Thai, or burger).
10. *WaitEstimate*: the wait estimated by the host (0–10 minutes, 10–30, 30–60, or >60).

Figure 2: The available input attributes (features) and their brief descriptions.

We can view this problem as playing a *20 questions* game, where every question we ask:

- should help us in narrowing down the value of the target *WillWait*
- depends on the previous questions that we may have already asked

Here, features represent questions and the answer is the specific feature value for a data instance.
What happens when we ask a question?

- Depending upon the number of possible answers the data is *split* into multiple subsets
- If a subset has the same value for the target concept (e.g., *WillWait*) then we have our answer
- If the subset has different values for the target concept then we need to ask more questions

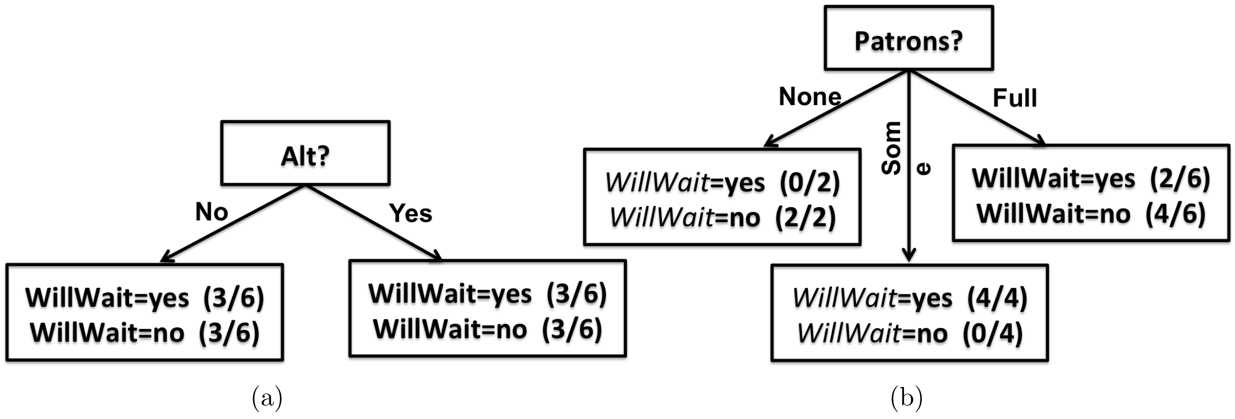


Figure 3: Possible splitting of data based on different features. The root shows the feature we are testing, and its child nodes show how the data is split.(a) Split using the *Alternate?* feature, (b) using the *Patrons?* feature.

Figure-3 shows what happens when we ask two different questions. For instance in the case of Figure-3a, we have selected to ask the question whether an alternate restaurant is available nearby or not, in this case we can see that the question is not informative in regards to answering our question because no matter what the answer is we have a fifty-fifty chance of being correct. This is as good as deciding randomly (flipping a fair coin).
However, if we ask the question about the number of patrons in the restaurant (Figure-3b) then for two out of the three possible answers we can predict the target concept with 100% confidence. If we keep asking questions till we reach nodes in which we can make the predictions with acceptable chances of error or run out of questions we will get a decision tree built using the training data. One possible decision tree for the *WillWait* problem is shown in Figure-4.

1 Decision Trees

A decision tree can be viewed as a function that maps a vector valued input to a single output or "decision" value. The output/decision value is real-valued or continuous when the tree is used for regression and is discrete valued in the case of a classification tree.

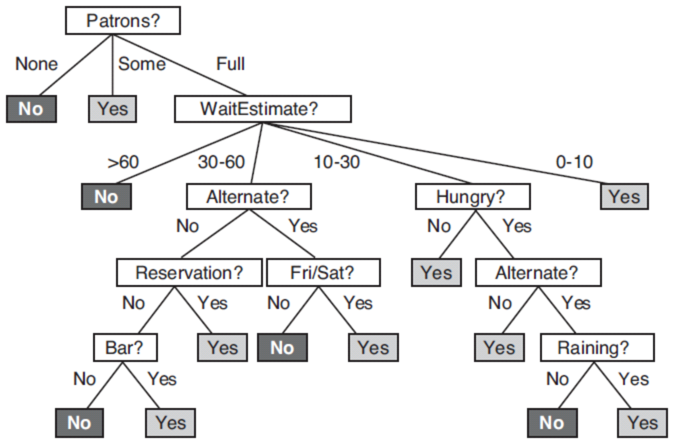


Figure 4: A decision tree for the *WillWait* task

1.0.1 How does it work?

- A decision tree maps the input to the final decision value by performing a sequence of tests on the different feature values.
- For a given data instance, each internal node of the tree tests a single feature value (later we will discuss how this can be generalized to utilize multiple features) to select one of its child nodes.
- This process continues till we reach a leaf node which assigns the final decision value to the instance.

Predicting the label for a data instance using a decision tree: Given an instance:

$$x_1 = (\text{'Yes','No','Yes','Yes','Full','\$\$\$','No','No','French'}, 10-30)$$

and the decision tree in Figure-4, we can predict the label as:

- Start at the root and follow the branch corresponding to the value of the feature that the root tests (*Patrons*). Since the given instance has a value of 'full' for this feature we follow the corresponding branch and reach the next node which tests for the *WaitEstimate* feature.
- Repeat this process till a leaf node is reached. For the given instance this results in a decision of 'No'.

Remark: It should be noted here that by changing the order in which the features are tested we can end up with an entirely different tree which maybe shorter or longer than the one in Figure-4. Similarly, not all features are required to be used in the tree, for example the tree does not test the *Type* or *Price* features.

```
Data: Training data with $m$ features and $n$ instances
Result: Induced Decision Tree
Begin with an empty tree;
while stopping criterion is not satisfied do
    select best feature;
    split the training data on the best feature;
    repeat last two steps for the child nodes resulting from the split;
end
```

Algorithm 1: A greedy algorithm for inducing a decision tree from given training data

2 Learning Decision Trees

From a learning perspective we are interested in learning the "optimal" tree from a given training dataset. There are a number of concerns that we need to address before we develop an algorithm to search for the optimal tree.

2.1 Number of possible decision trees:

The first and foremost problem is to find the number of possible trees that can be constructed to fit our training data. If we consider only binary features and a binary classification task, then for a training dataset having m features there are at most 2^m possible instances. In this case there are 2^{2^m} different ways of labeling all instances, and each labeling corresponds to a different underlying boolean function that can be represented as a decision tree. This means that we have a very large search space, for example given ten features we have about 2^{1024} possible decision trees. Therefore, searching for the best tree becomes an intractable task, and we resort to different heuristics that are able to learn 'reasonably' good trees from training data.

2.2 What is a good decision tree?

Given a choice between two decision trees, which one should we choose? In other words how can we define the "goodness" of a decision tree? Consistency with training data i.e., how accurately does the decision tree predict the labels of the training instances, is one such measure. But, we can always build a decision such that each instance has its own leaf node, in which case the decision tree will have 100% accuracy. The problem with such a tree is that it will not be able to *generalize* to unseen data. This shows that training error by itself is not a suitable criterion for judging the effectiveness of a decision tree. To address this problem, we define a good decision tree as the one that is as small as possible while being consistent with the training data.

2.3 Inducing decision trees from training data:

Finding the smallest decision tree consistent with training data is an NP-complete problem and therefore we use a heuristic greedy approach to iteratively grow a decision tree from training data. Algorithm-1 outlines the general set of steps. We start the explanation of the proposed algorithm by first defining what is meant by a split and how to choose the "best" feature to create a split.

2.4 Tree splitting:

In Figure-1, each internal node of the decision tree defines a split of the training data based on one single feature. Splits based on only one feature are also known as *univariate* splits. For example the root node splits the training data into three subsets corresponding to the different values that the *Patrons* feature can take, so that the left-most child has all the instances whose feature value is 'None'. The type of split that a feature defines depends on how many possible values the feature can take. A binary feature will split the data into two sub-groups while a feature with k distinct values will define a k-ary (multiway) split resulting in multiple sub-groups.

2.4.1 Which feature is the best?

Let the training data for a classification task consist of N instances defined by m features, and let y denote the output/response variable, so that the training can be represented as (x_i, y_i) with $x_i = [x_{i1}, x_{i2}, \ldots, x_{im}]$, and $y_i \in \{1, 2, \ldots, k\}$ for $i = 1, 2, \ldots, N$. Consider a node q in the tree, that has N_q instances, then the proportion of the instance belonging to class k can be estimated as:

$$p_{qk} = \frac{1}{N_q} \sum_{i=1}^{N_q} \mathbb{I}(y_i = k) \tag{1}$$

where $\mathbb{I}(.)$ is an indicator function which is 1 when the arguments evaluate to true and 0 otherwise. In the above equation the indicator function is used simply to count the number of instances belonging to class k. We classify all the instances in q to the majority class i.e., $class(q) = \arg\max_k p_{qk}$.

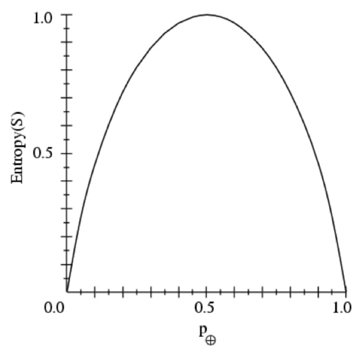
At node q we need to select the next best feature which will define the next split. One possible criterion for feature selection would be to measure the classification error of the resulting split.

$$Error(q) = \frac{1}{N_q} \sum_{i=1}^{N_q} \mathbb{I}(y_i \neq class(q)) \tag{2}$$

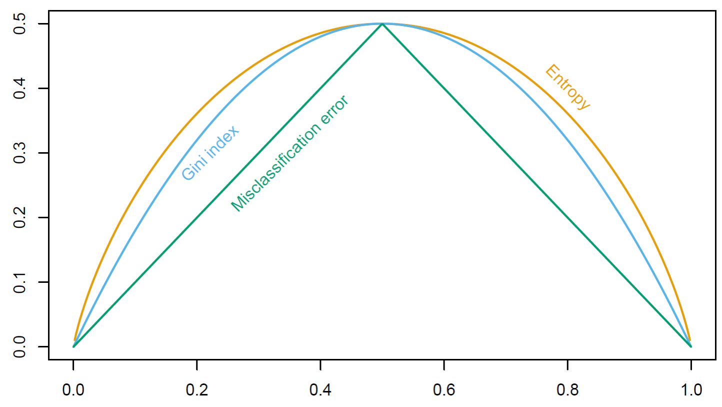
A feature that minimizes the classification error would be considered as the best feature to split node q.

For example consider the training data shown in Figure-1, and we decide that *Bar* is the feature that we should use to define our first split (root of the decision tree). This would produce a binary split resulting in two new nodes that we will represent by $\{x_1, x_2, x_4, x_5, x_8, x_{11}\} \in Bar_{no}$ and $\{x_3, x_6, x_7, x_9, x_{10}, x_{12}\} \in Bar_{yes}$. Before splitting the data the misclassification error was 50% and for both Bar_{no} and Bar_{yes} it is still 50%. So *Bar* is a bad choice as it has no effect on the classification error.

Although, classification error can be used to grow a decision tree, it has some shortcomings: it does not favor pure nodes i.e., nodes that have instances belonging to only one class. As an example assume that we have a binary classification task with each class having 50 instances, represented as $(50, 50)$. Consider two splits, one which results in $(10, 40)$ and $(40, 10)$ and another that produces $(50, 20)$ and $(0, 30)$. In both cases the classification error is 20%, but the latter split produces a pure node which should be preferred over the former. Therefore, instead of looking at classification error we use other measures that characterize the *purity/impurity* of a node, so that our goal is to prefer splits that produce pure nodes. It should be noted here that the *impurity of a node is measured in terms of the target variable i.e., the class labels and not the feature itself.*



(a)



(b)

Figure 5: (a) The entropy and its comparison to other node impurity measures (b) for binary classification tasks. The x-axis in both cases is the probability that an instance belongs to class 1, and in (b) the entropy function has been scaled down for ease of comparison.

2.4.2 Node Impurity Measures

Entropy: The entropy for a node q having N_q instances is defined as:

$$H(q) = -\sum_{i=1}^{k} p_{qk} \log_2 p_{qk} \tag{3}$$

where k is the number of classes. Entropy measures the impurity of a given node: if all the instances belong to one class then entropy is zero (for this we would need to assume that $0 \log_2 0 = 0$). Similarly, for a uniform distribution of class labels i.e., there is an equal number of instances belonging to each class entropy reaches its maximum value. This can be seen in the case of binary classification in Figure-5a, where the entropy function has a maximum at $p = 0.5$.

In order to select the best feature to further split a node we want the maximum reduction in entropy, i.e., the child nodes should be as pure as possible (or less impure than their parent). If we split node q based on feature V, that has $|V|$ distinct values (resulting in a $|V|$-way split) the reduction in entropy is calculated as:

$$IG(q, V) = H(q) - \sum_{i=1}^{|V|} \frac{N_i}{N_q} H(i) \tag{4}$$

where N_i is the number of instances in the i^{th} child node and $H(i)$ is its entropy calculated using (3). (4) is known as information gain, and we would select the feature with the maximum information gain to split node q.

Gini Index: The Gini index for a node q having N_q instances is defined as:

$$Gini(q) = \sum_{i=1}^{k} p_{qk}(1 - p_{qk}) \tag{5}$$

where k is the number of classes. Gini index is another impurity measure that is extensively used and is similar to entropy as can be seen from Figure-5b. It reaches its maximum value when the instances are equally distributed among all classes, and has a value of zero when all instances belong to one class. Similar to information gain (4), we can measure the gain in Gini index when splitting on feature A:

$$Gain_{Gini}(q, V) = Gini(q) - \sum_{i=1}^{|V|} \frac{N_i}{N_q} Gini(i) \tag{6}$$

where N_i is the number of instances in the i^{th} child node and $Gini(i)$ is calculated using (5). The feature with the maximum gain will be used to split node q.

2.4.3 Binary Splits vs Multiway Splits:

Recursive binary splits are more desirable than multiway splits, because multiway splits can fragment the training data during the early tree growing process and leave less data for the later stages. Since, any k-ary split can be represented equivalently as a sequence of binary splits (*homework problem*), using a binary splitting strategy is preferable.

- Ordinal Features: The values taken by an ordinal feature are discrete but can be ordered. The features *Price*, *Patrons* and *WaitEstimate* are examples of ordinal features. To create a binary split based on an ordinal feature $x_{(i)}$ with 'k' distinct values, we can select one of the feature values as the threshold θ and then split the data into two subsets corresponding to $x_{(i)} < \theta$ and $x_{(i)} \geq \theta$. Figure-6 shows an example of creating a binary split for an ordinal feature.

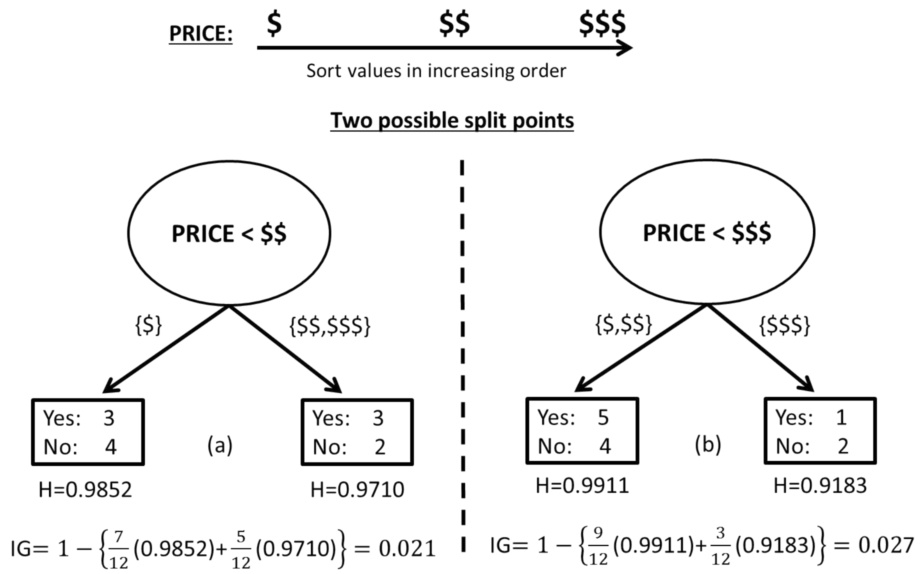

PRICE:
$
$$
$$$
Sort values in increasing order
Two possible split points
PRICE < $$
{$}
{$$,$$$}
Yes: 3
No: 4
(a)
Yes: 3
No: 2
H=0.9852
H=0.9710
IG= 1 − {7/12 (0.9852)+5/12 (0.9710)} = 0.021
PRICE < $$$
{$,$$}
{$$$}
Yes: 5
No: 4
(b)
Yes: 1
No: 2
H=0.9911
H=0.9183
IG= 1 − {9/12 (0.9911)+3/12 (0.9183)} = 0.027

Figure 6: Defining a binary split for an ordinal variable using the *Price* feature from Example-1. The left branch contains feature values that are < the testing value, and the right branch has all values that are ≥ the testing value.

- Numerical Features: A multiway split for a continuous feature, can result in an N-ary split, where N is the total number of training instances, so that each child has only one instance. To avoid this situation we transform a continuous feature into a binary feature by defining a suitable threshold θ, and then splitting the data as $x_{(i)} < \theta$ and $x_{(i)} \geq \theta$. Let $x_{1i}, x_{2i}, \ldots, x_{Ni}$ be the sorted values for the i^{th} feature. Then the optimal threshold can be found by setting the threshold as each of the $N-1$ mid-points i.e., $\frac{x_{ji}+x_{(j+1)i}}{2}$; $\forall j \in \{1, 2, \ldots, N-1\}$, and identifying the threshold value that maximizes the gain.

- Nominal (Categorical) Features: Defining a binary split for nominal features entails partitioning the set of all possible feature values in two non-empty and non-overlapping subsets. For a nominal feature $A \in \{a_1, a_2, \ldots, a_v\}$ this requires evaluating $2^{v-1} - 1$ possible binary partitions. For example consider the *Type* feature from Example-1. An exhaustive search over the possible binary partitions would entail evaluating the following splits:

  1. {Burger},{French,Thai}
  2. {French},{Burger,Thai}
  3. {Thai},{Burger,French}

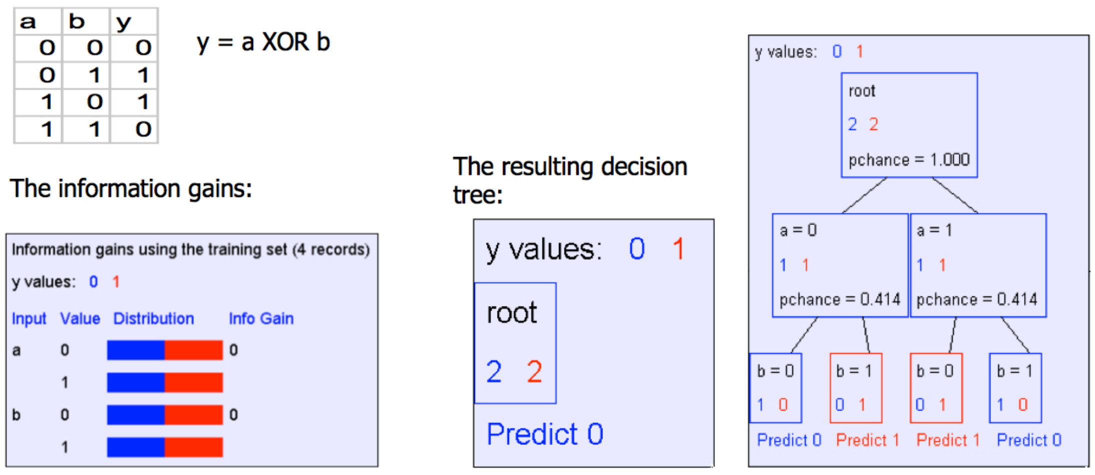
  One possible strategy to avoid an exhaustive search is to replace the original feature by v dummy boolean features, where the i^{th} dummy feature represents the indicator function $\mathbb{I}(x_{ji} = v_i)$; $\forall j \in \{1, 2, \ldots, N\}$.

2.4.4 When to stop splitting?

- All training instances at a given node have the same target/decision value. REMARK: $H(Y) = 0$

- No attributes can further distinguish records. REMARK : $H(Y|X) = H(Y) \rightarrow IG = 0$ for all features X

Is it a good idea to stop splitting if the information gain is zero?



a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

y = a XOR b

The information gains:

Information gains using the training set (4 records)

y values: 0 1

Input	Value	Distribution	Info Gain
a	0	🟦🟥	0
	1	🟦🟥	
b	0	🟦🟥	0
	1	🟦🟥	

The resulting decision tree:

Figure 7: "Do not split" VS "split" when Information gain is 0

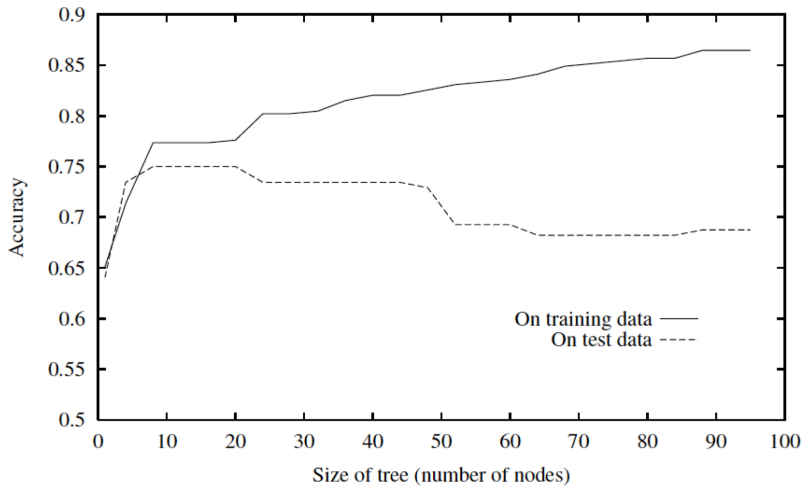


Figure 8: Overfitting.

2.5 Overfitting

Overfitting occurs when the decision tree has low classification error on the training data, but considerably higher classification error on test (previously unseen) data. Figure-8 shows an example of overfitting and we can see that as the number of node in the tree grow its accuracy increases on training data but the accuracy on test data gets lowered. As the size of the tree grows, the decision tree starts to capture the peculiarities of the training data itself such as measurement noise instead of learning a general trend from the data which better characterizes the underlying target concept. There are a number of methods that can be used to avoid growing larger and deeper trees.

- **Pre-Pruning (Early Stopping Criterion):**We can incorporate the preference of learning shorter trees within the tree growing process by imposing a limit on:

    1. Maximum number of leaf nodes
    2. Maximum depth of the tree
    3. Minimum number of training instances at a leaf node

    Remark: Care should be taken while setting these limit, because stringent limits can result in a shallow tree that is unable to adequately represent the target concept we are trying to learn. There are no general rules to how these limits should be set and usually these limits are dataset specific.

- **Reduced Error Post-Pruning:** Another strategy to avoid overfitting in decision trees is to first grow a full tree, and then prune it based on a previously held-out validation dataset. One pruning strategy is to propagate the errors from the leaves upwards to the internal nodes, and replace the sub-tree rooted at the internal node by a single leaf node. By labeling this new leaf node as predicting the majority class, we can calculate the reduction in validation error for every node in the tree. The subtrees rooted at the node with error reduction can be pruned and replaced by a leaf node. The algorithm is summarized as:

```
**Data**: Validation Data $D_{vd}$ and fully grown tree $T_{full}$
**Result**: Pruned Decision Tree $T_{pruned}$
T=$T_{full}$;
while all nodes of T are tested do
    select an inner node $t \in T$;
    construct $T'$: replace $t$ with a leaf node using training data;
    if $error(T', D_{vd}) \leq error(T, D_{vd})$ then
        $T = T'$;
    end
end
```

Algorithm 2: Algorithm for reduced error post-pruning of decision trees.

- **Rule-based Post-Pruning:** In rule-based pruning, a decision tree is converted into an equivalent set of rules. This is done by generating a separate rule for each path (root to leaf) in the decision tree. For example, in the case of the decision tree shown in Figure-4 the right-most path translated into the rule:

  IF (*Patrons* = 'Full') **AND** (*WaitEstimate* = '0-10') **THEN** (*WillWait* = 'Yes')

  The pruning strategy is then to refine each rule by removing any preconditions or antecedents that do not increase the error on validation data. So, in the first step for the rule given above the following two rules will be tested:

  1. **IF** (*Patrons* = 'Full') **THEN** (*WillWait* = 'Yes')
  2. **IF** (*WaitEstimate* = '0-10') **THEN** (*WillWait* = 'Yes')

  If any of these rules have an error that is lower than the original rule, then we would replace the original rule with the new pruned rule. The pruning process will recurse over the new (shorter) rules and halts when all the pruned rules are worst than their un-pruned version. At the end of the pruning procedure all the rules are sorted based on their accuracy over the validation data, and are used in this sequence to classify new instances.

2.6 Missing Values

In most real world datasets, there are missing feature values in the data. There are multiple reasons that can account for missing data, for example device failure, unanswered survey question, etc. In the case of missing feature values in our training data, we can:

- throw away incomplete instances (if we have enough training data)
- throw away the feature if a very high number of instances (e.g., 70%) have missing feature values
- fill in the missing values based on the overall feature mean
- fill in the missing values based on the class mean for the missing feature

- in certain cases where a missing value has its own specific meaning for example in the case of a medical test where a certain hormone is not detected we can create a new feature value to represent that the value is missing (not measured)
- if we come to an internal node and the feature value is missing, we can distribute the instance to all the child nodes with diminished weights that reflect the proportion of instances going down to each child

2.7 Different misclassification costs

Consider the task of classifying between spam and non-spam email messages. In this case classifying something that is non-spam as spam entails a higher cost than labeling a spam message as non-spam. Similarly, for problems arising in the medical diagnosis/evaluation domain false negatives have a higher cost than false positives. Assume that we have a binary classification task, and have used a decision tree to predict the labels for our training dataset. The results can be summarized using a confusion matrix:

		Predicted Class	
		P	**N**
Actual Class	**P**	True Positive(TP)	False Negative(FN)
	N	False Positive(FP)	True Negative(TN)

Figure 9: Confusion matrix for a binary classification task. We have two classes positive (P) and negative (N).

So far, we have dealt with uniform costs i.e., the cost of predicting a false positive or a false negative is equal (unit cost, every mistake counts as one). However, in domains where we care about what errors are more costlier than others, we need to incorporate this information into the tree induction process, so that the final decision tree gives more attention to reducing the types of error that have a high cost. To this end we can specify the misclassification costs for a $k-$class classification problem using a $(k \times k)$ loss/cost matrix that is defined as follows:

$$\begin{bmatrix} 0 & L_{12} & L_{13} & \dots & L_{1k} \\ L_{21} & 0 & L_{23} & \dots & L_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{k1} & L_{k2} & L_{k3} & \dots & 0 \end{bmatrix} \tag{7}$$

where, L_{ij} represent the cost of misclassifying an instance belonging to the i^{th} class as belonging to class j. The diagonal entries represent the cost associated with predicting the correct label which is zero. These costs can be integrated into Gini index by observing that we can equivalently represent (5) as $Gini(q) = \sum_{k \neq k'} p_{qk} p_{qk'}$ (*Homework problem*), and then modify it as:

$$Gini(q) = \sum_{k \neq k'} L_{kk'} p_{qk} p_{qk'} \tag{8}$$

This approach works only for classification problems that have more than two classes.

3 Regression trees

Lets say that for each node m, χ_m is the set of datapoints reaching that node.
Estimate a predicted value per tree node

$$g_m = \frac{\sum_{t \in \chi_m} y_t}{|\chi_m|}$$

Calculate mean square error

$$E_m = \frac{\sum_{t \in \chi_m} (y_t - g_m)^2}{|\chi_m|}$$

How to choose the next split. If $E_m < \theta$, then stop splitting. Otherwise choose the split that realizes the maximum drop in error for all all brances. Say we are considering feature X with branches $x_1, x_2, ..., x_k$, and lets call χ_{mj} the subset of χ_m for which $X = x_j$.

$$g_{mj} = \frac{\sum_{t \in \chi_{mj}} y_t}{|\chi_{mj}|}$$

$$E'_m(X) = \frac{\sum_j \sum_{t \in \chi_{mj}} (y_t - g_{mj})^2}{|\chi_m|}$$

We shall choose X such that $E'_m(X)$ is minimized, or the drop in error is maximized.

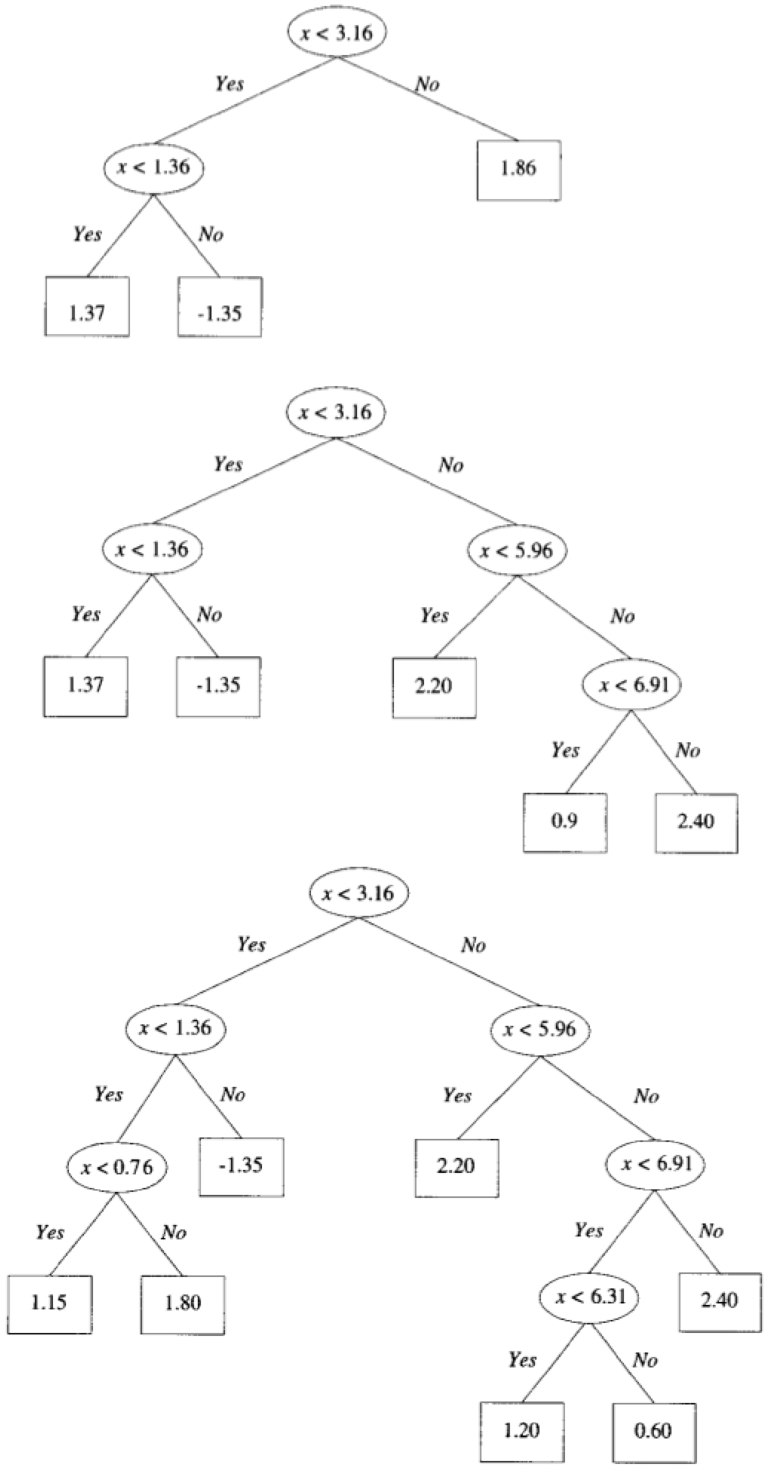


Figure 10: Regression tree

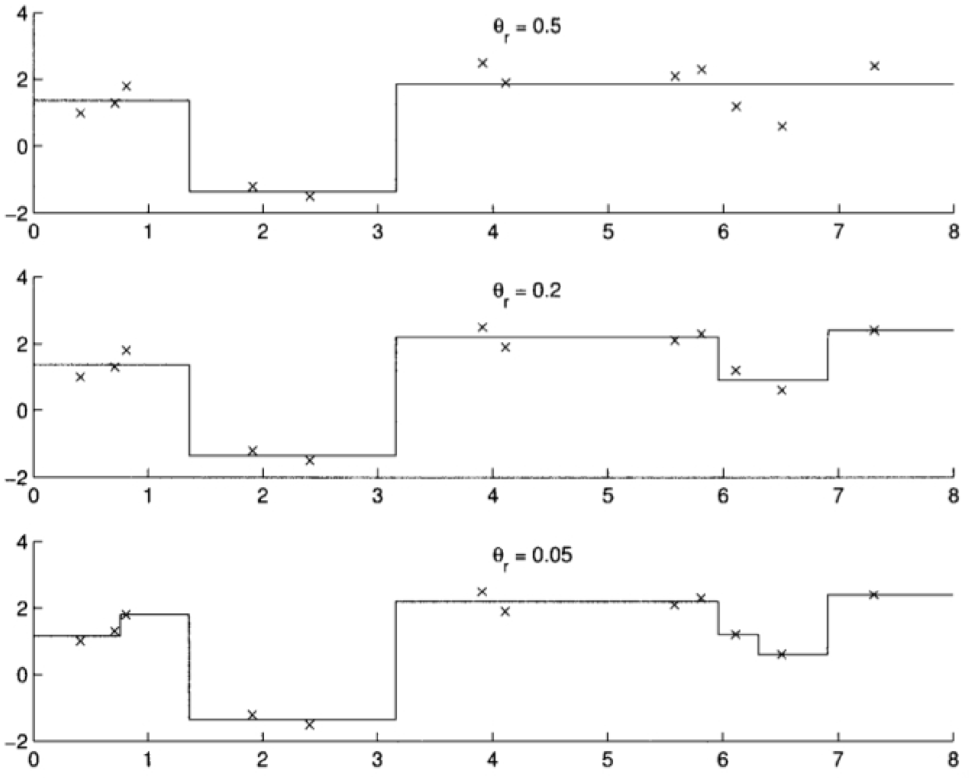


Figure 11: Regression fit

4 Multivariate Decision Trees (*Optional*)

The class of decision functions that a univariate decision tree can learn is restricted to axis-aligned rectangles, as shown in Figure-12, for two continuous features $x_1 \in \mathbb{R}$ and $x_2 \in \mathbb{R}$.

Figure-13 shows an example where a univariate decision tree fails to capture the underlying linear decision function. In a multivariate tree, the splitting criteria can be a functional of more than one feature. For example, we can modify the root of the first tree shown in Figure-12 as having the following splitting criterion:

$$2 * x_1 + 1.5 * x2 < 0.5$$

More generally, a binary linear multivariate node m split can look like

$$w_1 x_1 + w_2 x_2 + ... w_m x_m + w_0 > 0$$

Such splits can be extremely powerful (if data is linearly separable, a single split at root can create a perfect classification); even more complex splits can be obtained using nonlinear split functions. However, finding a good multivariate split is not anymore a matter of brute force: there are $2^d \binom{N}{d}$ possible splits (or hyperplanes). Later on in the course we will discuss linear classification and how good hyperplanes can be obtained without an exhaustive search.

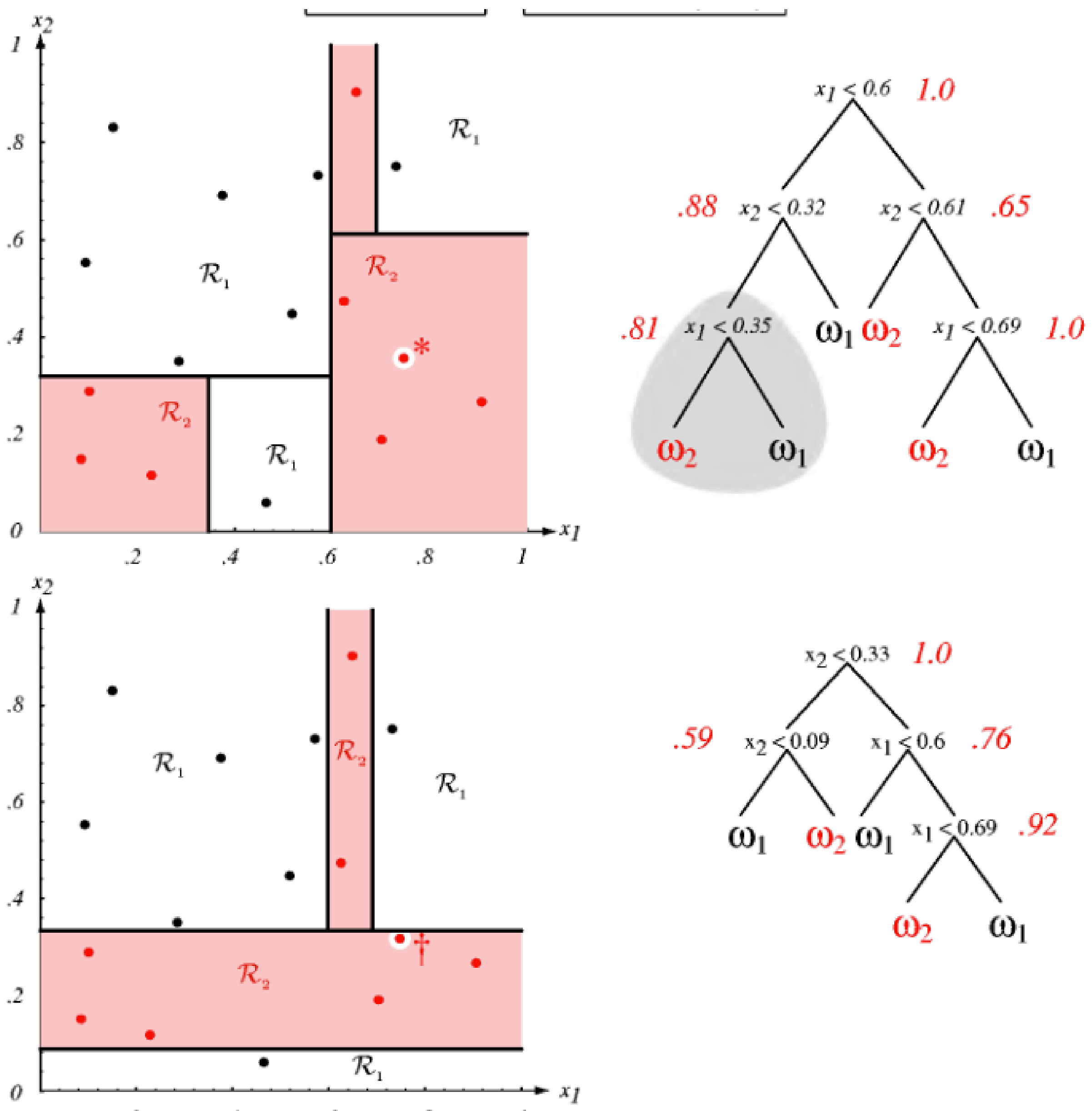


Figure 12: Real-valued information gain and decision tree

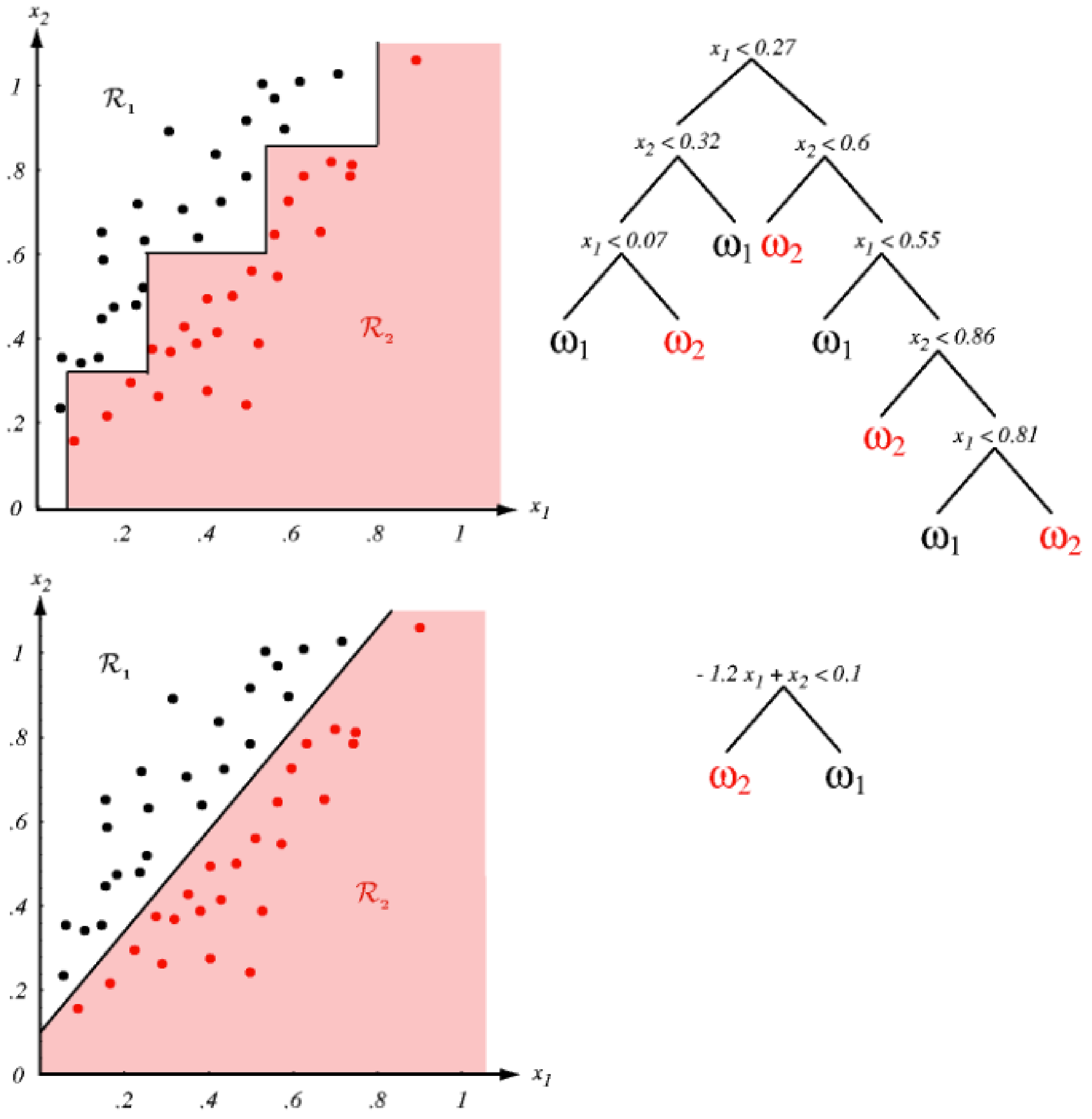


Figure 13: The target concept that we need to learn is a linear function of both x_1 and x_2. A univariate decision tree (*top*) is unable to capture the target concept. On the other hand a multivariate decision tree (*bottom*) learns the linear decision boundary using only a single split.