Supervised Learning:
"basic" learners: Decision Trees Classifiers and Regression Trees
Supervised Classifiers and Regressors: definition of the task

- Let D be a dataset representing historical data about a given domain
- Let’ assume that D is a list of records, called instances \( x_i: (x_{i1}\ldots x_{in}) \)
- Every \( x_{ij} \) represents the value of an attribute, or feature, describing the instance (values can be discrete or continuous)
- One of these features is the target for which we want to learn a prediction model \( f(x) \)
- In classification tasks, the target is a discrete variable (binary or multi-valued), called the «class» \( c \)
- In classifiers, the prediction model to be learned is often denoted as \( c(x) \)
- If the target is continuous, then we denote the predictor as a regressor, and the prediction function with \( f(x) \)
Illustrating Classification Task

<table>
<thead>
<tr>
<th>Tid</th>
<th>Attrib1</th>
<th>Attrib2</th>
<th>Attrib3</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>Large</td>
<td>125K</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Medium</td>
<td>100K</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>Small</td>
<td>70K</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>Medium</td>
<td>120K</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>No</td>
<td>Large</td>
<td>95K</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>Medium</td>
<td>80K</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>Yes</td>
<td>Large</td>
<td>220K</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>No</td>
<td>Small</td>
<td>85K</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>No</td>
<td>Medium</td>
<td>75K</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>No</td>
<td>Small</td>
<td>90K</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Training Set

<table>
<thead>
<tr>
<th>Tid</th>
<th>Attrib1</th>
<th>Attrib2</th>
<th>Attrib3</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>No</td>
<td>Small</td>
<td>55K</td>
<td>?</td>
</tr>
<tr>
<td>12</td>
<td>Yes</td>
<td>Medium</td>
<td>80K</td>
<td>?</td>
</tr>
<tr>
<td>13</td>
<td>Yes</td>
<td>Large</td>
<td>110K</td>
<td>?</td>
</tr>
<tr>
<td>14</td>
<td>No</td>
<td>Small</td>
<td>95K</td>
<td>?</td>
</tr>
<tr>
<td>15</td>
<td>No</td>
<td>Large</td>
<td>67K</td>
<td>?</td>
</tr>
</tbody>
</table>

Test Set

Learning algorithm

Induction

Apply Model

Deduction

Model
Examples of Classification Task

- Predicting tumor cells as benign or malignant (class is, e.g., $\text{benign}(x)$ with values $\text{pos}$ and $\text{neg}$)
- Classifying credit card transactions as legitimate or fraudulent (class is, e.g. fraudulent(x), with values pos and neg)
- Classifying secondary structures of protein as alpha-helical, beta-sheet, or random coil
- Categorizing news stories as finance, weather, entertainment, sports, etc (class is $\text{category}(x)$ with values weather, sport, finance.. )
Simple example of training set for object classification

<table>
<thead>
<tr>
<th>Instances</th>
<th>Size</th>
<th>Color</th>
<th>Shape</th>
<th>C(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>small</td>
<td>red</td>
<td>circle</td>
<td>positive</td>
</tr>
<tr>
<td>x2</td>
<td>large</td>
<td>green</td>
<td>circle</td>
<td>positive</td>
</tr>
<tr>
<td>x3</td>
<td>small</td>
<td>red</td>
<td>triangle</td>
<td>negative</td>
</tr>
<tr>
<td>x4</td>
<td>large</td>
<td>blue</td>
<td>square</td>
<td>negative</td>
</tr>
</tbody>
</table>
Note that instances can be represented as “records” in a table but also as vectors (or points) in a multi-dimensional space. Instances are often denoted also as feature vectors.

<table>
<thead>
<tr>
<th>Instances</th>
<th>Size</th>
<th>Color</th>
<th>Shape</th>
<th>C(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>small</td>
<td>red</td>
<td>circle</td>
<td>positive</td>
</tr>
<tr>
<td>x2</td>
<td>large</td>
<td>green</td>
<td>circle</td>
<td>positive</td>
</tr>
<tr>
<td>x3</td>
<td>small</td>
<td>red</td>
<td>triangle</td>
<td>negative</td>
</tr>
<tr>
<td>x4</td>
<td>large</td>
<td>blue</td>
<td>square</td>
<td>negative</td>
</tr>
</tbody>
</table>

In the cartesian space, each symbol is associated with a number, e.g. circle=0, triangle=1 square =2 in the «shape» axis.
Summary of this presentation

- Decision tree algorithm
- Ordering nodes (Information Gain)
- Regression Trees (Residual Sum of Squares)
- Testing and pruning

NOTE: decision trees are a very «old» ML algorithm, one of the first developed. Regression Trees are more widely used, and the basic principles are similar to DTs. Both DTs and RTs are at the basis of Random Forest, a very popular Ensemble method (will see with Ensambles)
Decision Trees

- The model output is a tree structure. **Nodes** are tests on features values, there is one branch for each value of the feature, and **leaf** nodes are “decisions”, they specify the class label.

- A decision tree can represent any boolean function $c(x)$, i.e., a classification function over discrete-valued feature vectors.

- The tree can be rewritten as a set of rules, i.e. disjunctive normal form (DNF). Example (for the left tree):

  - red $\land$ circle $\rightarrow$ pos
  - red $\land$ circle $\rightarrow$ A
  - blue $\rightarrow$ B; red $\land$ square $\rightarrow$ B
  - green $\rightarrow$ C; red $\land$ triangle $\rightarrow$ C
Every test node is a test on the value (or range) of one feature. For each possible outcome of the test, an edge is created that links to a subsequent test node or to a leaf node.

Leaf nodes are decisions concerning the value of the classification.
Toy example

- Decision tree to decide whether to go home by bus or walking
- Binary class: values are either walk or bus
- Decision is taken based on the values of just 3 features: wather (W), time (T), and hungry (H)
- Features are either discrete or discretized (e.g., >30m or <30m)
- The decision tree can be re-written in terms of a decision function $c(x)$ in first order logic: $c(x(W,T,H))=(\text{IF}(W=\text{Sun} \ \text{AND} \ T>30) \ \text{THEN} \ \text{Walk}) \ \text{OR} \ (\text{IF} \ W=\text{Sun} \ \text{AND} \ T<30) \ \text{THEN} \ \text{Bus}) \ \text{OR} \ (\text{IF} \ W=\text{Cloud} \ \text{AND} \ H=\text{Yes}) \ \text{THEN} \ \text{Walk}) \ \text{OR} \ (\text{etc. etc})$
A real example:
character recognition based on visual features

Can’t be visualized without zooming, but boxes (decisions) are alphabet letters, circles are tests on values of several visual features
How do we learn a decision tree?

- We use historical data for which the value of the class is known
- The basic process is **greedy recursive partition** of the decision space
How does it work: Top-Down Decision Tree Induction

- Recursively build a tree top-down by **divide and conquer**.

At each step, we aim to find the **“best split”** of our data. What is a good split? **One which reduces the uncertainty of classification** for “some” split!

Learning best splits (best ordering of tests on features, and best split over its values) is learning the DT **parameters**.
Top-Down Decision Tree Induction

• Recursively build a tree top-down by divide and conquer.

\[
\begin{align*}
\text{big, red, circle} &: + \\
\text{small, red, circle} &: + \\
\text{small, red, square} &: - \\
\text{big, blue, circle} &: -
\end{align*}
\]

The process ends when we can output decisions (= the class labels), but:
How do we decide the order in which we test attributes?
How do we decide the class of nodes for which we have no examples?

Let’s ignore for now these 2 issues and describe the algorithm first.
Decision Tree Induction Pseudocode

Algorithm $\text{DTree}(\text{examples } D, \text{features } F)$ returns a tree:

a) If all $\text{examples } D$ are in one category, return a leaf node with that category label

b) Else if the set of $\text{features } F$ is empty, return a leaf node with the category label that is the most common in examples.

Else pick a feature $f$ in $F$ and create a node $R$ for it

For each possible value $x_i$ of $f$:

Let $s_i$ be the subset of examples that have value $x_i$ for $f$

Add an outgoing edge $E$ to node $R$ labeled with the value $x_i$.

If $s_i$ is empty

then attach a leaf node to edge $E$ labeled with the category that is the most common in examples

else call $\text{DTree}(s_i, \text{features} - \{f\})$ and attach the resulting tree as the subtree under edge $E$.

Return the subtree rooted at $R$.

a) and b) are the termination conditions
Example

Instances:
<big, red, circle>: +  <small, red, circle>: +
<small, red, square>: −  <big, blue, circle>: −

Features:
- dimension, shape, color

1. Pick a feature $f$ and create a node $R$ for it, eg. Color

2. For each possible value $x_i$ of $f$ (red for example):
   1. Let $s_i$ be the subset of examples that have value $v_i$ for $f$.
   2. Add an outgoing edge $E$ to node $R$ labeled with the value $x$.
   3. if (...) else call \( \text{DTree}(\text{example}(s_i), \text{features} - \{f\}) \) and attach the resulting tree as the subtree under edge $E$.

call the algorithm on the subset for the feature red:
\[
\text{DTree}(<\text{big, red, circle}>: + , <\text{small, red, circle}>: + , <\text{small, red, square}>: −)
\]
Example

4. **Pick** a feature $f$ and create a node $R$ for it, eg. **shape**

5. **If** all $s_i$ are in one category, return a leaf node with that category label.
6. **If** $s_i$ is empty, then attach a leaf node to edge $E$ labeled with the category that is the most common in examples.
Example: Backtrack to color (blue)

<big, blue, circle>: –

Now we know how to decide the class when we have no examples, but how do we decide the order in which we create nodes?
Summary

- Decision tree algorithm
- **Ordering nodes (Information Gain)**
- Regression Trees (Residual Sum of Squares)
- Fine-tuning the tree
Picking a Good Split Feature

- The goal is to have the resulting tree be **as small as possible**, per Occam’s razor.
- Finding a **minimal decision** tree (nodes, leaves, or depth) is an **NP-hard** optimization problem.
- The top-down divide-and-conquer method does a greedy search for a simple tree but does not guarantee to find the smallest.
  - The **general lesson in ML**: “Greed is good.”
- Want to pick a feature that creates subsets of examples that are relatively “pure” in a single class so they are “closer” to being leaf nodes.
- There are a variety of methods for picking a good test, a popular one is based on information gain that originated with the ID3 system of Quinlan (1979). The choice of the method to be used is an hyperparameter of the DT model.
About “purity”

● Look at the image below and think which group can be described easily. Intuitively, the answer is C because it requires less information, as all values are similar (=blue).

● On the other hand, B requires more information to describe it, and A requires the maximum information. In other words, we can say that C is a “pure” node, B is impure and A is more impure than B.
Entropy and Binary Entropy

- **Entropy** (disorder, impurity) of a set of examples, D, relative to binary classification is:

\[
E(D) = -p_1 \log_2(p_1) - p_0 \log_2(p_0) = -p_1 \log_2(p_1) - (1 - p_1) \log_2(1 - p_1)
\]

- where \( p_1 \) is the fraction of positive examples in D and \( p_0 \) is the fraction of negatives. (Notice that if S is a sample of a population, Entropy(S) is an estimate of the population entropy).

- If all examples are in one category (as for node C of the previous example), entropy is zero (we define \( 0 \cdot \log(0) = 0 \)).

- If examples are equally mixed (\( p_1 = p_0 = 0.5 \)), entropy is a maximum of 1.
Entropy can be viewed as the number of bits required on average to encode the class of an example in $D$. It is also an estimate of the initial “disorder” or “uncertainty” about a classification, given the set D.

**General Formula of Entropy:** For multi-class problems with $C$ category values, entropy generalizes to:

$$E(D) = - \sum_{i=0}^{C} p_i \log_2(p_i)$$
Example: Entropy Computation

- We have two class labels, blue and yellow.
- In group C, we have 18 objects, and they are all blue:
  \[
  \text{Entropy}(C) = -p_{\text{blue}} \log(p_{\text{blue}}) - p_{\text{yellow}} \log(p_{\text{yellow}}) \\
  = -\frac{18}{18} \times \log(1) - \frac{0}{18} \times \log(0) = 0
  \]
- In group A, we have 20 objects, 9 are yellow, 11 are blue:
  \[
  \text{Entropy}(A) = -p_{\text{blue}} \log(p_{\text{blue}}) - p_{\text{yellow}} \log(p_{\text{yellow}}) \\
  = -\frac{11}{20} \times \log\left(\frac{11}{20}\right) - \frac{9}{20} \times \log\left(\frac{9}{20}\right) = 0.328 + 0.36 = 0.688
  \]
Entropy Plot for Binary Classification (only 2 class labels)
The **Information Gain** (IG, Gain) of a feature \( f \) is the \textbf{expected} reduction in entropy resulting from splitting on this feature.

\[
Gain(D, f) = Entropy(D) - \sum_{v \in \text{Values}(f)} \frac{|D_v|}{|D|} \text{Entropy}(D_v)
\]

where \( D_v \) is the subset of \( D \) having value \( v \) for feature \( f \) (e.g., if \( f = \text{color} \) and \( v = \text{red} \))
Information Gain

- Entropy of each resulting subset weighted by its relative size...

**Example:**

\[
\begin{align*}
\text{<big, red, circle>: +} & & \text{<small, red, circle>: +} \\
\text{<small, red, square>: −} & & \text{<big, blue, circle>: −}
\end{align*}
\]

\[
2+, 2 −: E=1 \\
\text{size}
\]

\[
\begin{align*}
\text{big:} & & \text{small:} \\
1+,1− & & 1+,1−
\end{align*}
\]

\[
\begin{align*}
E_{\text{big}} &= 1 \\
E_{\text{small}} &= 1
\end{align*}
\]

\[
\text{Gain} = 1 - (0.5 \cdot 1 + 0.5 \cdot 1) = 0
\]

\[
\text{<big, red, circle>: +} & & \text{<small, red, circle>: +} \\
\text{<small, red, square>: −} & & \text{<big, blue, circle>: −}
\end{align*}
\]

\[
2+, 2 −: E=1 \\
\text{color}
\]

\[
\begin{align*}
\text{red:} & & \text{blue:} \\
2+,1− & & 0+,1−
\end{align*}
\]

\[
\begin{align*}
E_{\text{red}} &= 0.918 \\
E_{\text{blue}} &= 0
\end{align*}
\]

\[
\text{Gain} = 1 - (0.75 \cdot 0.918 + 0.25 \cdot 0) = 0.311
\]

\[
\text{<big, red, circle>: +} & & \text{<small, red, circle>: +} \\
\text{<small, red, square>: −} & & \text{<big, blue, circle>: −}
\end{align*}
\]

\[
2+, 2 −: E=1 \\
\text{shape}
\]

\[
\begin{align*}
\text{circle:} & & \text{square:} \\
2+,1− & & 0+,1−
\end{align*}
\]

\[
\begin{align*}
E_{\text{circle}} &= 0.918 \\
E_{\text{square}} &= 0
\end{align*}
\]

\[
\text{Gain} = 1 - (0.75 \cdot 0.918 + 0.25 \cdot 0) = 0.311
\]

Initial Entropy is 1
New pseudo-code

DTree(examples, features) returns a tree

a) If all examples are in one category, return a leaf node with that category label.
b) Else if the set of features is empty, return a leaf node with the category label that is the most common in examples.

Else pick the best feature f according to IG and create a node R for it

For each possible value $x_i$ of $f$:
   Let $s_i$ be the subset of examples that have value $x_i$ for $f$
   Add an outgoing edge $E$ to node $R$ labeled with the value $x_i$

If $s_i$ is empty
   then attach a leaf node to edge $E$ labeled with the category that is the most common in examples.
   else call DTree($s_i$, features – {$f$}) and attach the resulting tree as the subtree under edge $E$.

Return the subtree rooted at $R$. 
A complete example
A Decision Tree example: “play Tennis”

- Data Example: “When do you play tennis?”

<table>
<thead>
<tr>
<th>instance</th>
<th><strong>Outlook</strong></th>
<th><strong>Temperature</strong></th>
<th><strong>Humidity</strong></th>
<th><strong>Windy</strong></th>
<th>Play</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>x2</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>x3</td>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>x4</td>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>x5</td>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>x6</td>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>x7</td>
<td>Overcast</td>
<td>Cool</td>
<td>Normal</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>x8</td>
<td>Sunny</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>x9</td>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>x10</td>
<td>Rainy</td>
<td>Mild</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>x11</td>
<td>Sunny</td>
<td>Mild</td>
<td>Normal</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>x12</td>
<td>Overcast</td>
<td>Mild</td>
<td>High</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>x13</td>
<td>Overcast</td>
<td>Hot</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>x14</td>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>True</td>
<td>No</td>
</tr>
</tbody>
</table>
The Process of Constructing a Decision Tree

● Select an attribute to place at the root of the decision tree and make one branch for every possible value.
● Repeat the process recursively for each branch.
Information Gained by knowing the Result of a Decision

In the “play tennis” example, there are 9 instances of which the decision to play is “yes” and there are 5 instances of which the decision to play is “no’. Then, the initial data entropy is:

\[
\frac{9}{14} \times \left( -\log \frac{9}{14} \right) + \left( \frac{5}{14} \right) \times \left( -\log \frac{5}{14} \right) = 0.940
\]

The information initially required to correctly separate the data is 0.940 bits.
Information further required if “Outlook” is placed at the root

0.971 is the entropy of 5 instances of which 3 have a label and 2 have the other label.

0 is the entropy of a dataset where all instances have the same class label.

Probability of outlook = sunny

\[\text{Information further required} = \left(\frac{5}{14}\right) \times 0.971 + \left(\frac{4}{14}\right) \times 0 + \left(\frac{5}{14}\right) \times 0.971 = 0.693 \text{ bits.}\]
Information Gained by Placing Each of the 4 Attributes

- Gain(outlook) = 0.940 bits – 0.693 bits = 0.247 bits.
- Gain(temperature) = 0.029 bits.
- Gain(humidity) = 0.152 bits.
- Gain(windy) = 0.048 bits.
The Strategy for Selecting an Attribute to Place at a Node

- Select the attribute that gives us the largest information gain.
- In this example, it is the attribute “Outlook”.

```
<table>
<thead>
<tr>
<th>Outlook</th>
<th>sunny</th>
<th>overcast</th>
<th>rainy</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 “yes”</td>
<td>3 “no”</td>
<td>4 “yes”</td>
<td>3 “yes”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 “no”</td>
</tr>
</tbody>
</table>
```
The Recursive Procedure for Constructing a Decision Tree

- Apply to each branch recursively to construct the decision tree.
- For example, for the branch “Outlook = Sunny”, we evaluate the information gained by applying each of the remaining 3 attributes.

  ➢ Gain(Outlook=sunny; Temperature) = 0.971 – 0.4 = 0.571
  ➢ Gain(Outlook=sunny; Humidity) = 0.971 – 0 = 0.971
  ➢ Gain(Outlook=sunny; Windy) = 0.971 – 0.951 = 0.02
Recursive selection

• Similarly, we also evaluate the information gained by applying each of the remaining 3 attributes for the branch “Outlook = rainy”.

- Gain(Outlook=rainy;Temperature) = 0.971 – 0.951 = 0.02
- Gain(Outlook=rainy;Humidity) = 0.971 – 0.951 = 0.02
- Gain(Outlook=rainy;Windy) = 0.971 – 0 = 0.971
The Resulting Tree

Outlook
- sunny
- overcast
- rainy

humidity
- high
- normal

windy
- false
- true

- no
- yes
DT can be represented as a set of rules

- IF Outlook = sunny AND humidity = high  → no
- IF Outlook = sunny AND humidity = normal  → yes
- IF Outlook = overcast  → yes
- IF Outlook = rainy AND windy = false  → yes
- IF Outlook = rainy AND windy = true  → no
Support and Confidence

- Each rule has a **support** \( \frac{|D_v|}{|D|} \) (or “cover”) represented by the set of examples that satisfy the rule.

- Each rule has also a **confidence** which might or might not be equal to support. The confidence is the subset of \( D_v \) **which is correctly classified by the rule.**
Support and Confidence

Remember one of the 2 “exit” conditions in the algorithm:

- **Else if** the set of features is empty, return a leaf node with the category label that is the most common in examples.

Hence if the set of examples |D_v| does not have a uniform classification, but, say, |D_v+| positive and |D_v-| negative, if |D_v+|>|D_v-|, we output:

- the label “positive”
- **support** is \[ \frac{|D_v+|}{|D|} \]
- **confidence** is \[ \frac{|D_v+|}{|D_v|} \]

For example if we “consume” all features in a branch of the tree and we remain with 5 examples, of which 3 positive and 2 negatives, we append the decision “positive” to the tree branch (and its associated rule), with support 3 (or 3/|D|) and confidence 3/5
Example case:
We have just one feature

RULE: IF Outlook=sunny THEN neg

Support=3/14
Confidence=3/5
Issues of Decision Tree Learning

- **Continuous (real-valued) features can be handled** by allowing nodes to split a real-valued feature into ranges based on a threshold (e.g. length < 3 and length ≥3).

- Algorithms for finding Dtrees are efficient for processing **large amounts of training data** for data mining tasks.

- Methods have been developed for handling **noisy training data** (both class and feature noise).

- Methods developed for handling **missing feature values**.

- Some of these issues will be analyzed during labs.

- Classification trees have discrete class labels at the leaves, **regression trees** allow real-valued outputs at the leaves.
Summary

- Decision tree algorithm
- Ordering nodes (Information Gain)
- **Regression Trees (Residual Sum of Squares)**
- Fine-tuning the tree
Regression Trees

- Regression trees handle both continuous features and non-categorical classification functions (reading on reg-trees: [link](#)).
- Regression trees output values, therefore they are regressors, as the name says, not classifiers.
Regression Trees

- In Dtrees we can discretize features, but this is part of **data pre-processing**
- In RT, creating splits on continuous features is part of the learning process (RT parameters)
- Every branch of the tree defines a region in the multi-dimensional space, the output (leaf nodes of the tree) is the mean value of the output of training data in the defined region

E.g., 43.43 is the mean value that the feature to be predicted has for all examples in the training set for which Industry is <748 and Population < 190
The regions in theory could have any shape. However, we choose to divide the feature space into high-dimensional rectangles or boxes (for simplicity and ease of interpretation of the resulting predictive model).

Our goal is to find boxes $R_1, \ldots, R_J$ that minimize the Residual Sum of Squares $RSS$ given by:

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where $\hat{y}_{R_j}$ is the mean observed value of the training samples within the $j$-th box, and $y_i$ is the value of each single observation in the box.

For example, given region $R_1$, we compute the average value of the output function $y$ for all points $x_i$ in the training set that fall into $R_1$. 
Regression trees (3)

- It is computationally unfeasible to consider every possible partition of the feature space into J boxes.
- Thus, we take a top-down, greedy approach called **recursive binary splitting**, called «top-down» since it begins at the top of the tree (all observations below to a single region) and then successively splits the predictor space.
- Each split is indicated via two new branches further down on the tree.
- It is *greedy* since at each step of the tree building process, the best split is made at that particular split (rather than looking ahead and picking a split that will lead to a better tree in a future split).
- It still requires scanning all the observed values of the training set (or region) at each split.
- You can learn more on RT algorithm at this [link](#) and this second [link](#).
Recursive binary splitting (1)

For a continuous or ordinal variable (feature) with \( m \) distinct values, \( x_1 \ x_2 \ldots \ x_m \) in the learning set, consider each value in turn. Each selection produces a partition consisting of two sets of values: \( x < x_i \) and \( x > x_i \), \( i = 1, 2, \ldots, m \).

Thus there are \( m - 1 \) possible partitions to consider.
Recursive binary splitting (2)

- For every candidate split, we compute the residual sum of squares \( \text{RSS} \) for \( y = f(x) \) in the two regions:
  \[
  \text{RSS(\text{split})} = \text{RSS}_1 + \text{RSS}_2 = \sum_{R_1} (y_i - \bar{y}_1)^2 + \sum_{R_2} (y_i - \bar{y}_2)^2
  \]

- We also compute the overall RSS
  \[
  \text{RSS}_0 = \sum (y_i - \bar{y})^2
  \]

- We choose the partition for which \( \text{RSS}_0 - \text{RSS(\text{split})} \) is the largest.

- The principle is very similar to Decision Trees, but we have RSS rather than IG!!!!

\( y_1 \) is the average value of the output function \( f(x) \) for member of R1
Quizzzzzzzzzz

- Remember, we said that learning implies tuning the model parameters and establishing an optimization problem.
- Which parameters we tune in DT and RT?
- What is the optimization function?
Regression trees visualized

http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html
Tree depth: 1
Tree Depth: 2
Tree depth 5
Tree depth: 6
Summary

- Decision tree algorithm
- Ordering nodes (Information Gain)
- Regression Trees (Residual Sum of Squares)
- Fine-tuning the tree
Fine-tuning the tree

- Decision or regression trees are not optimal, they are obtained as the result of a greedy process.
- A common problem is that they might be excessively bushy – this is a general problem (for all types of ML algorithms) denoted as OVERFITTING.
- Overfitting happens when a model learns too many details and even noise in the training data, causing a negative impact on the ability of the model to generalize. This means that the model may perform poorly on new data.
Overfitting decision trees

<table>
<thead>
<tr>
<th>Underfit tree</th>
<th>Optimal tree</th>
<th>Overfit tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy on training = 50%</td>
<td>Accuracy on training = 70%</td>
<td>Accuracy on training = 90%</td>
</tr>
<tr>
<td>Accuracy on test = 50%</td>
<td>Accuracy on test = 70%</td>
<td>Accuracy on test = 65%</td>
</tr>
</tbody>
</table>
Overfitting

- Learning a tree that classifies the training data perfectly may not lead to the tree with the best generalization to unseen data.
  - There may be noise in the training data that the tree is erroneously fitting.
  - The algorithm may be making poor decisions towards the leaves of the tree that are based on very little data and may not reflect reliable trends (e.g. reliable rules should be supported by “many” of examples, not just a handful).
- A hypothesis, $h$, is said to overfit the training data if there exists another hypothesis, $h'$, such that $h$ has less error than $h'$ on the training data but a greater error on independent test data.

![Graph showing accuracy vs. hypothesis complexity](image_url)
Overfitting more in general

- **Under-fitting**: (too simple to explain the variance)
- **Appropriate-fitting**:
- **Over-fitting**: (forcefitting -- too good to be true)
Overfitting Prevention in Dtrees: (Pruning) Methods

● Two basic approaches for decision trees:

➢ **Pre-pruning**: Stop growing tree as some point during top-down construction when there is no longer sufficient data to make reliable decisions (e.g. \( |D_v| < k \)).

➢ **Post-pruning**: Grow the full tree, then remove sub-trees that do not have sufficient evidence.

● Label leaf resulting from pruning with the majority class of the remaining data, or a class probability distribution.
Reduced Error Pruning

A post-pruning, cross-validation approach:

1. **Partition** training data \( D \) in “learning” \( L \) and “validation” \( V \) sets.
2. **Build a complete tree** from the \( L \) data.
3. Until accuracy on the validation set \( V \) decreases, do:
   1. for each non-leaf node, \( n \), in the tree do:
      1. **Temporarily prune** the subtree below \( n \) and replace it with a leaf labeled with the current majority class at that node.
      2. **Measure and record** the **accuracy** of the pruned tree on the validation set.
   2. Permanently **prune** the node that does not result in a significant increase in accuracy on the validation set.
The majority of “removed” examples under the pruned sub-tree is neg.

Is this pruned tree significantly worse than the full tree? If the same or similar accuracy, then choose the pruned tree.
Issues with Reduced Error Pruning

- The problem with this approach is that it potentially “wastes” training data on the validation set.
- The severity of this problem depends on where we are on the learning curve:

A learning curve shows the accuracy, e.g. the learning rate, as the number of training examples grows.
Pruning Regression Trees

- One possibility is to stop growing the tree when the «gain» in RSS is below a given threshold.
- A better strategy is to grow a very large tree $T_o$ and then prune it back to obtain a subtree.
- How to we find the best subtree?
- We want to select a subtree that leads to the lowest test error rate.
- One popular technique is called Cost Complexity Pruning (if interested, you can learn more here) – it is, again, similar to reduced post pruning..
Will see more on decision and regression trees..

- ... when we will introduce ensemble methods