Our first learning algorithm

• How would you classify the next example?

kNN Algorithm (simple form)

• At "training time" do nothing.
• Store examples.
• When given new example:
  - find k nearest neighbors
  - Predict L= majority vote of their labels

kNN Algorithm

• Theoretical basis + intuition:
  • "in the limit", when the dataset is dense, this should pick up "all important regions"
  • Very flexible classifier: no prior commitment to the shape, density, or distribution of regions

kNN: problems and extensions

• Noisy labels in training data
• Expensive test time/application: Complexity of finding the neighbors
• How to choose k?
• The effect of "close neighbors" and "far neighbors"

kNN: problems and extensions

• Completely dependent on the distance metric and representation
• E.g., Euclidean distance:
  - May require Normalization of features
  - Treats all dimensions equally
  - Sensitive to high dimension/ irrelevant features

• Solutions?
kNN: problems and extensions

- Can we predict real values instead of discrete labels (i.e., solve the regression problem)?

• Solutions?

Our second algorithm

Let's look at a simple dataset for motivation:

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temp</th>
<th>Humidity</th>
<th>Windy</th>
<th>Play</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Cool</td>
<td>Normal</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Mid</td>
<td>High</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Mid</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Overcast</td>
<td>Mid</td>
<td>High</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>Overcast</td>
<td>Hot</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Mid</td>
<td>High</td>
<td>True</td>
<td>No</td>
</tr>
</tbody>
</table>

• Class: Play tennis
• Attributes:
  – Outlook
  – Temp
  – Humidity
  – Windy

Decision Trees

• Give a different way to identify regions in instance space:

  • Recursively split on values of features to define regions that have single label

• What does this look like in Euclidean space?

• Decision trees can represent any discrete function of the attributes!

  • Why?

Decision Trees

• Given training set, can we build a tree that agrees with the data? (yes: easy; why?)

• What is a good decision tree?

  • Given training set, how can we build a good tree?
Several selection criteria have been proposed and used.

Information gain is commonly used (C4.5, J48)

We need to learn about entropy ...

\[
\text{Entropy}(p_1, \ldots, p_n) = \sum_{i} p_i \log \frac{1}{p_i} = - \sum_{i} p_i \log p_i
\]

For 2 classes \( p_1 = p, p_2 = 1-p \) and this simplifies to

\[
\text{Entropy}(p) = - \sum p \log p - (1-p) \log (1-p)
\]

\[
\text{Gain}(\text{Split}) = \text{Ent}(S) - \sum_{j} \frac{|S_j|}{|S|} \text{Ent}(S_j)
\]

Heuristic for wide splits

\[
\text{SplitInfo} = \sum_{j} \frac{|S_j|}{|S|} \log \frac{|S|}{|S_j|}
\]

\[
\text{GainRatio} = \frac{\text{Gain}}{\text{SplitInfo}}
\]

Outlook = Sunny:

\[
\text{info}(2,3) = \text{entropy}(2/5,3/5) = -2/5 \log(2/5) - 3/5 \log(3/5) = 0.971 \text{ bits}
\]

Outlook = Overcast:

\[
\text{info}(4,0) = \text{entropy}(1,0) = -1 \log(0) - 0 \log(0) = 0 \text{ bits}
\]

Outlook = Rainy:

\[
\text{info}(3,2) = \text{entropy}(3/5,2/5) = -3/5 \log(3/5) - 2/5 \log(2/5) = 0.971 \text{ bits}
\]

Expected information for attribute:

\[
\text{info}(3,2,4,0,3,2) = (5/14) \times 0.971 + (4/14) \times 0 + (5/14) \times 0.971 = 0.693 \text{ bits}
\]
### Decision Trees

\[
\text{gain(Outlook)} = \text{info}(\{9, 5\}) - \text{info}(\{2, 3\}, \{4, 0\}, \{3, 2\})
= 0.940 - 0.693
= 0.247 \text{ bits}
\]

\[
\text{gain(Temperature)} = 0.029 \text{ bits}
\]

\[
\text{gain(Humidity)} = 0.152 \text{ bits}
\]

\[
\text{gain(Windy)} = 0.048 \text{ bits}
\]

### Continuing to Split

\[
\begin{align*}
\text{gain(Temperature)} & = 0.571 \text{ bits} \\
\text{gain(Humidity)} & = 0.971 \text{ bits} \\
\text{gain(Windy)} & = 0.020 \text{ bits}
\end{align*}
\]

### Final Decision Tree

![Decision Tree Diagram]

### Decision Tree Learning Algorithm

- If data has a pure class
  - Make leaf node with that class
- Otherwise
  - Pick feature to split on
  - Divide data into sub-datasets \( s_j \) according to the feature’s values
  - Recursively build a tree for each subset

### Other Criteria / Tasks

- The Gini Criterion = \( 4p(1 - p) \)
- The \([KM]\) Criterion = \( 2\sqrt{p(1 - p)} \)

\[
\text{Criterion for Regression} = \frac{1}{t} \sum_{i=1}^{t} (v_i - \bar{v})^2
\]

\[
\bar{v} = \frac{1}{t} \sum_{i=1}^{t} v_i
\]

### Real Valued Attributes

- Naïve treatment makes a very side split with possibly one example per branch.
  - Is this good?
  - Alternative picks threshold \( t \) and tests \( (\text{feature} >= t) \) to get a binary split.
  - How can we pick \( t \)?
The Bad News

• This does not quite work …

![Graph showing accuracy vs. size of tree](chart)

Overfitting in DT

• Why Does this happen?

- Few examples at lower levels in tree
- Quantities calculated “not reliable” in this case
- Even worse with “noisy data”
- And when features not sufficiently rich

• Solutions?

Overfitting in DT

• Solution 1 (C4.5, J48): uses a confidence interval based on class ratio at leaf and number of examples in the training set.
• This is not fully justified but works well in practice.
• Solution 2: use a validation set. Known as reduced error pruning (REP)

REP Example

<table>
<thead>
<tr>
<th>color</th>
<th>speed</th>
<th>Traffic density</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>&gt;75</td>
<td>Y</td>
</tr>
<tr>
<td>black</td>
<td>&lt;=75</td>
<td>N</td>
</tr>
<tr>
<td>blue</td>
<td>&gt;75</td>
<td>Y</td>
</tr>
<tr>
<td>black</td>
<td>&lt;=75</td>
<td>N</td>
</tr>
<tr>
<td>red</td>
<td>&gt;75</td>
<td>Y</td>
</tr>
<tr>
<td>black</td>
<td>&lt;=75</td>
<td>N</td>
</tr>
<tr>
<td>red</td>
<td>&gt;75</td>
<td>Y</td>
</tr>
<tr>
<td>black</td>
<td>&lt;=75</td>
<td>N</td>
</tr>
<tr>
<td>red</td>
<td>&gt;75</td>
<td>Y</td>
</tr>
<tr>
<td>black</td>
<td>&lt;=75</td>
<td>N</td>
</tr>
<tr>
<td>red</td>
<td>&gt;75</td>
<td>Y</td>
</tr>
<tr>
<td>black</td>
<td>&lt;=75</td>
<td>N</td>
</tr>
<tr>
<td>red</td>
<td>&gt;75</td>
<td>Y</td>
</tr>
<tr>
<td>black</td>
<td>&lt;=75</td>
<td>N</td>
</tr>
</tbody>
</table>

REP Example

<table>
<thead>
<tr>
<th>Decision Errors</th>
<th>Keep</th>
<th>Prune</th>
</tr>
</thead>
<tbody>
<tr>
<td>2Y, 4N</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>2Y, 1N</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>2Y, 5N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4Y, 2N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1Y, 9N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2Y, 8N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Missing Attribute Values

- Common in real data
- We can handle this in a way that works across algorithms (that is, also for kNN).
- How?
- But we can do better with a solution tailored for decision trees. **How?**

### Recap

- We already know two learning algorithms
- And many variants or improvements over their basic forms
- **Given new application:**
  - How do we know if one of these (say kNN) is doing well?
  - And which one is doing better?