CS 383/613 – Machine Learning

Decision Trees

Slides adapted from material created by E. Alpaydin
Prof. Mordohai, Prof. Greenstadt, Pattern Classification (2nd Ed.),
Pattern Recognition and Machine Learning

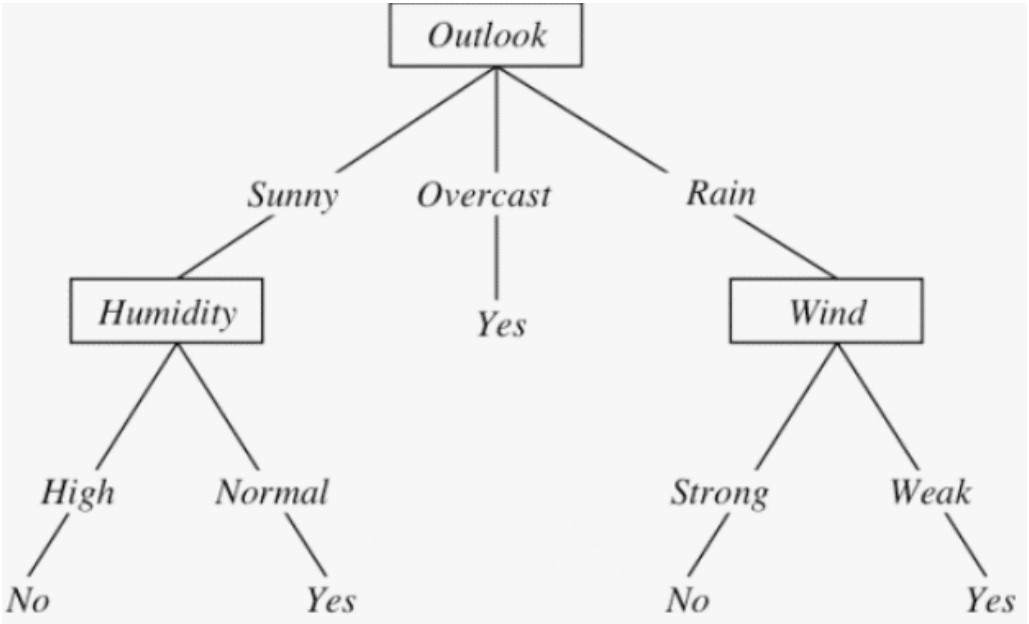
Objectives

- Decision Trees

Decision Trees

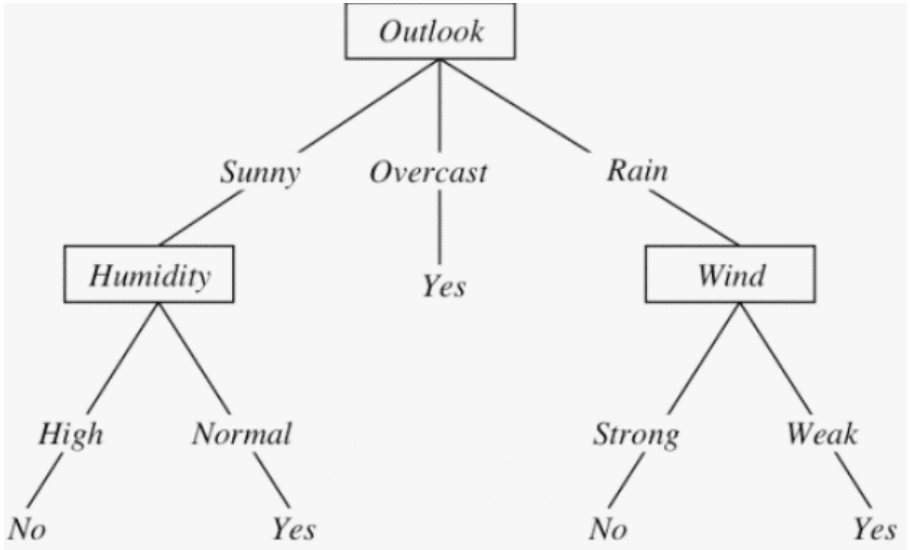
Decision Trees

- Building Decision Trees: Hierarchical and Recursive partitioning of the feature space
- Example: Play tennis?



Decision Trees

- Representation
  - Each internal node tests an attribute
  - Each branch is an attribute value
  - Each leaf assigns a classification



Consistent Decision Trees

- If we have D binary features and a binary classification system then there are 2^{2^D} decision trees
  - Yikes
- If our data is noiseless and without any randomness, then there is at least one trivially **consistent** decision tree for the data set
  - Fits the data perfectly
- In fact, there's many potential ones
- But we'd prefer to find a *compact* one.
  - That is one with a minimal height.

Example

- Example: Situations for which I will/won't play tennis
- **Exercise:** Let's (painfully) build a consistent decision tree from these examples
  - There's a lot of different possible ones

Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes

Decision Tree Learning

- In practice it may be impossible to build a consistent decision tree
  - There may be noise
  - And/or there may be randomness
  - Or we don’t have enough information/features.
- Ultimately, we’d like to find a small (compact) tree consistent with as many training examples as possible.

Decision Tree Learning

- Idea: (recursively) choose "most significant" attribute/feature as root of (sub)tree
- Here's a greedy algorithm to do that:

function DTL(X,Y,$features$,$probs$)
    if X is empty then return a leaf node **with *probs* as the probability for each class**.
    elseif all Y have same values, return a leaf node with value Y_1
    elseif $features$ is empty return a leaf node with **probabilities for eacn cıass**.
    else
        $best$ = ChooseAttribute($X, Y, features$)
        $tree$ = new internal node testing feature $best$
        for each value v_i in $best$ do
            $[X^{(i)}, Y^{(i)}]$ = elements of $[X, Y]$ with $best = v_i$
            $subtree$ = DTL($X^{(i)}, Y^{(i)}, features - best, prob(Y)$)
            add a branch to $tree$ with label v_i and subtree $subtree$
        return $tree$

Decision Tree Learning

- Our initial call will be:
  - `root = DTL(Xtrain,Ytrain,[0:D-1],probs(Ytrain))`
- Additional things to note:
  - $[0, D-1]$ gives a list of the column (feature) numbers.
  - $ChooseAttribute$ returns the ID of the feature that is best among those remaining
  - $features - best$ removes feature number $best$ from the set of available ones in $features$.
  - $probs(Ytrain)$ returns the class probabilities from $Ytrain$

function DTL(X,Y,$features$,$probs$)
    if X is empty then return a leaf node with **with *probs* as the probability for each class**.
    elseif all Y have same values, return a leaf node with value Y_1
    elseif $features$ is empty return a leaf node with **probabilities for each class**.
    else
        $best$ = ChooseAttribute($X, Y, features$)
        $tree$ = new internal node testing feature $best$
        for each value v_i in $best$ do
            $[X^{(i)}, Y^{(i)}]$ = elements of $[X, Y]$ with $best = v_i$
            $subtree$ = DTL($X^{(i)}, Y^{(i)}, features - best, prob(Y)$)
            add a branch to $tree$ with label v_i and subtree $subtree$
        return $tree$

Binary Decision Tree

- If each feature is binary, then this can be made a bit simpler by implementing a binary tree:

function DTL(X,Y,$features$,$probs$)
    if X is empty then return a leaf node with **with *probs* as the probability for each class**.
    elseif all Y have same values, return a leaf node with value Y_1
    elseif $features$ is empty return a leaf node with **probabilities for each class**
    else
        $best$ = **ChooseAttribute($X, Y, features$)**
        $tree$ = new internal node testing feature $best$

        $[X^{(T)}, Y^{(T)}]$ = elements of $[X, Y]$ with $best = True$
        $leftChild = DTL(X^{(T)}, Y^{(T)}, features - best, prob(Y))$
        Set $leftChild$ to be the left child of $tree$.

        $[X^{(F)}, Y^{(F)}]$ = elements of $[X, Y]$ with $best = False$
        right$Child = DTL(X^{(F)}, Y^{(F)}, features - best, prob(Y))$
        Set $rightChild$ to be the right child of $tree$.

        return $tree$

Choosing an Attribute

- Idea: A good attribute splits the examples into subsets that contain (ideally) observations from just one class.
  - Zero randomness in class labels.
- Which feature is the most useful then for splitting the data at first?
  - Outlook?
  - Temperature?
  - Humidity?
  - Windy?
- While there's many ideas on how to choose the best attribute, one common decision tree learning algorithms, ID3, uses *class entropy.*

Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

Entropy

- Given probability of events $v_1, \dots, v_K$ as $P(v_1), \dots, P(v_K)$ we can compute the ***entropy*** of this system as

$$H\big(P(v_1), \dots, P(v_K)\big) = \sum_{i=1}^{K} (-P(v_i) \log_{\boldsymbol{K}} P(v_i))$$

- Entropy measures the randomness of the data
- Example: Tossing a fair coin
  - $v_1 = heads, v_2 = tails,$
  - $P(v_1) = 0.5, P(v_2) = 0.5$
  - $H\left(\frac{1}{2}, \frac{1}{2}\right) = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1$

Weighted Average Entropy

- Returning to decision trees….
- When deciding what feature to split on, we'll compute the *weighed average entropy* of the class labels of the subsets it creates.
  - And select the one that has the lowest.
- To compute this for a given feature:
  - Let the feature split the data coming into the current tree node into *subsets* based on their value of that feature.
  - Compute class entropy of each subset.
  - Return the weighed average of these entropies (weighed by the percentage of samples in each subset).

Weighted Average Entropy

- Let
  - H_i be the entropy of subset i
  - $|C_i|$ be the number of observations in subset i.
  - S be the number of subsets created by the current feature.
  - N be the number of observations.
- We can then define the average entropy as:

$$\mathbb{E} = \sum_{i=1}^{S} \frac{|C_i|}{N} H_i$$

Example

- For reference:
  - $H\big(P(v_1), \ldots, P(v_K)\big) = \sum_{i=1}^{K}(-P(v_i)\log_{\boldsymbol{K}} P(v_i))$
  - $\mathbb{E} = \sum_{i=1}^{S} \frac{|C_i|}{N} H_i$
- Data:

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 3 \\ 2 & 2 \\ 1 & 2 \\ 3 & 2 \\ 2 & 2 \end{bmatrix}, Y = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

- Let's figure out which feature provides the lowest average post-split entropy!

Example

$$H\big(P(v_1),\dots,P(v_K)\big)=\sum_{i=1}^{K}\left(-P(v_i)\log_{\boldsymbol{K}}P(v_i)\right)$$

$$\mathbb{E}=\sum_{i=1}^{S}\frac{|C_i|}{N}H_i$$

- Feature 1

$$\mathbb{E}_1=\frac{3}{6}\left(-\frac{2}{3}\log_2\frac{2}{3}+-\frac{1}{3}\log_2\frac{1}{3}\right)+\frac{2}{6}\left(-\frac{1}{2}\log_2\frac{1}{2}+-\frac{1}{2}\log_2\frac{1}{2}\right)+\frac{1}{6}(0\log(0)-1\log(1))=0.7925$$

- Feature 2

$$\mathbb{E}_2=\frac{1}{6}\left(-\frac{1}{1}\log_2 1+-0\log_2 0\right)+\frac{4}{6}\left(-\frac{1}{4}\log_2\left(\frac{1}{4}\right)+-\frac{3}{4}\log_2\left(\frac{3}{4}\right)\right)+\frac{1}{6}\left(-\frac{1}{1}\log\left(\frac{1}{1}\right)-\frac{0}{1}\log\left(\frac{0}{1}\right)\right)=0.5409$$

$$X=\begin{bmatrix}1&1\\1&3\\2&2\\1&2\\3&2\\2&2\end{bmatrix},Y=\begin{bmatrix}1\\1\\1\\0\\0\\0\end{bmatrix}$$

More than 2 classes?

$$H\big(P(v_1), \ldots, P(v_K)\big) = \sum_{i=1}^{K} (-P(v_i) \log_{\boldsymbol{K}} P(v_i))$$

$$\mathbb{E} = \sum_{i=1}^{S} \frac{|C_i|}{N} H_i$$

- Think about how you can change the equations for work for more than two classes…
- The equations hold!

$$H\big(P(v_1), \ldots, P(v_K)\big) = \sum_{i=1}^{K} (-P(v_i) \log_{K} P(v_i))$$

$$\mathbb{E} = \sum_{i=1}^{S} \frac{|C_i|}{N} H_i$$

Example

- Ok. Let's try to use all this to build our ID3 decision tree!

function DTL(X,Y,$features$,$probs$)
  if X is empty then return a leaf node with **with *probs* as the probability for each class**.
  elseif all Y have same values, return a leaf node with value Y_1
  elseif $features$ is empty return a leaf node with **probabilities for each class**.
  else
    $best$ = **ChooseAttribute($X, Y, features$)**
    $tree$ = new internal node testing feature $best$
    for each value v_i in $best$ do
      $[X^{(i)}, Y^{(i)}]$ = elements of $[X, Y]$ with $best = v_i$
      $subtree$ = DTL($X^{(i)}, Y^{(i)}, features - best, prob(Y)$)
      add a branch to $tree$ with label v_i and subtree $subtree$
    return $tree$

Outlook	Temperature	Humidity	Windy	Play
Sunny	Hot	High	False	No
Sunny	Hot	High	True	No
Overcast	Hot	High	False	Yes
Rainy	Mild	High	False	Yes
Rainy	Cool	Normal	False	Yes
Rainy	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Sunny	Mild	High	False	No
Sunny	Cool	Normal	False	Yes
Rainy	Mild	Normal	False	Yes
Sunny	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Rainy	Mild	High	True	No

Predicting

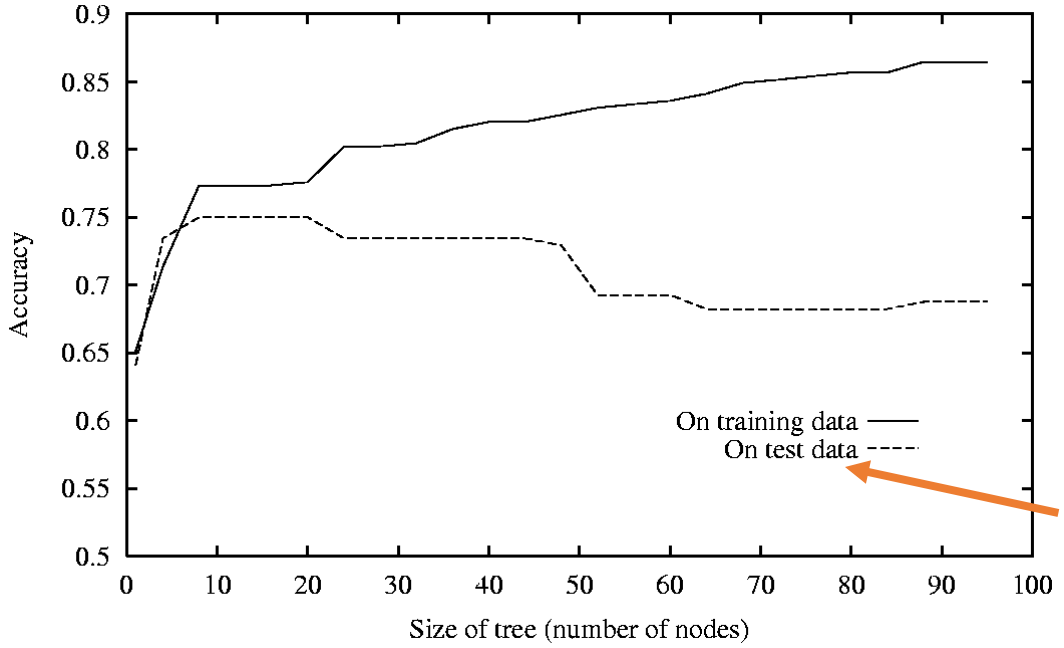
- Once we have a decision tree built, we can use it to make predictions.
- To do so just involves *traversing* the decision tree from the root down to a leaf node.
- Something like…

```
function predict(x,node)
    if isLeaf(node)
        return node.label
    else
        child = node.child[x[node.feature]]
        return predict(x,child)

yhat = predict(x,root)
```

Overfitting

- What's the problem with fitting our data as closely as possible?
  - We may overfit, the data!
- Since this is an iterative (or recursive) algorithm, the use of a validation set to decide between different versions is quite natural.



Note: Many people unfortunately use testing and validation sets interchangeably ☹.

http://jmvidal.cse.sc.edu/talks/decisiontrees/allslides.html

Reduced Error Approaches

- How can we use the validation set?
- Reduced Error Growing
  - When you look to split a node based on training data, evaluating after using *validation set*
  - If things get worse, stop
- Reduced Error Pruning
  - Grow fully out then….
  - Evaluate impact on *validation* set of pruning each possible node (plus those below it)
  - Greedily remove the one that most improve *validation* set accuracy
  - Continue until none help

Continuous Valued Inputs

- This is one of those algorithms that works most naturally/easily with categorical features!
- So, we could convert any continuous features to categorical ordinal ones.