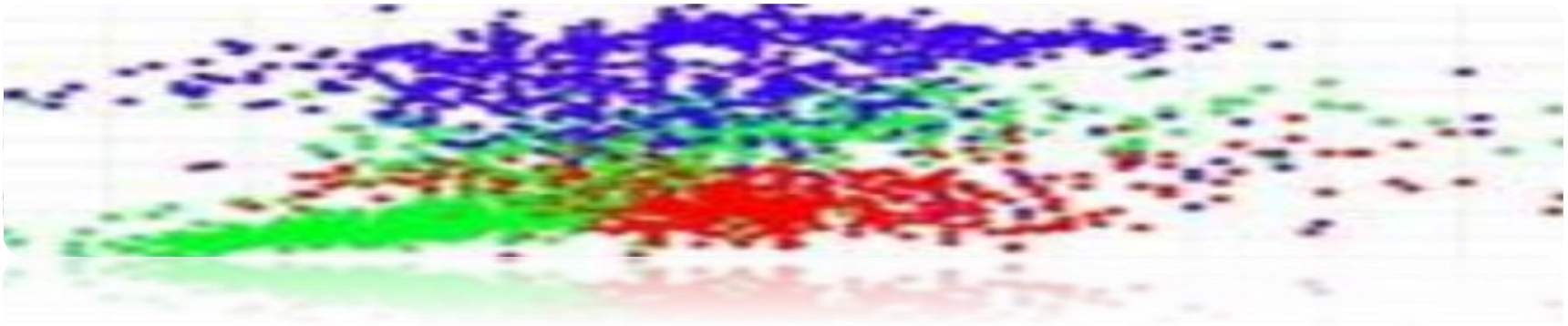


NON-LINEAR CLASSIFICATION MODELS

Outline

- k-Nearest Neighbors
- Rule-based classification
- Decision trees
- Random forests
- Boosting

Distance metric

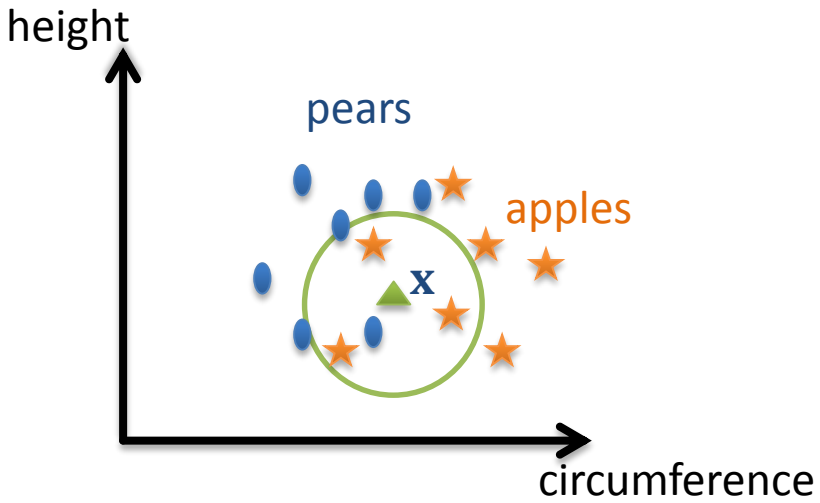
- Let $\boldsymbol{X} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots\}$ be the instance space

- A function $d: \boldsymbol{X} \times \boldsymbol{X} \to \mathbb{R}$ is called a **distance metric** if for every $\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k \in \boldsymbol{X}$ (i.e., the metric space):
  1. $d(\mathbf{x}_i, \mathbf{x}_j) = 0$ only if $i = j$, otherwise $d(\mathbf{x}_i, \mathbf{x}_j) > 0$
  2. $d(\mathbf{x}_i, \mathbf{x}_j) = d(\mathbf{x}_j, \mathbf{x}_i)$ (i.e., **symmetry**)
  3. $d(\mathbf{x}_i, \mathbf{x}_k) \leq d(\mathbf{x}_i, \mathbf{x}_j) + d(\mathbf{x}_j, \mathbf{x}_k)$ (**triangle inequality**)

In a metric space M with distance d the similarity between any $x, y \in M$ can be defined as $sim(x, y) := \frac{1}{1+d(x,y)}$ or $sim(x, y) := \frac{1}{e^{d(x,y)}}$

Metric distance	Definition
Euclidean	$\|\mathbf{x} - \mathbf{y}\| = \sqrt{\sum_i (x_i - y_i)^2}$
Manhattan	$\|\mathbf{x} - \mathbf{y}\|_1 = \sum_i \|x_i - y_i\|$
Maximum	$\|\mathbf{x} - \mathbf{y}\|_\infty = \max_i \|x_i - y_i\|$
Mahalanobis	$d_{maha}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i \left(\frac{x_i - y_i}{\sigma_i}\right)^2}$ (for normally distributed data)

k-Nearest Neighbors

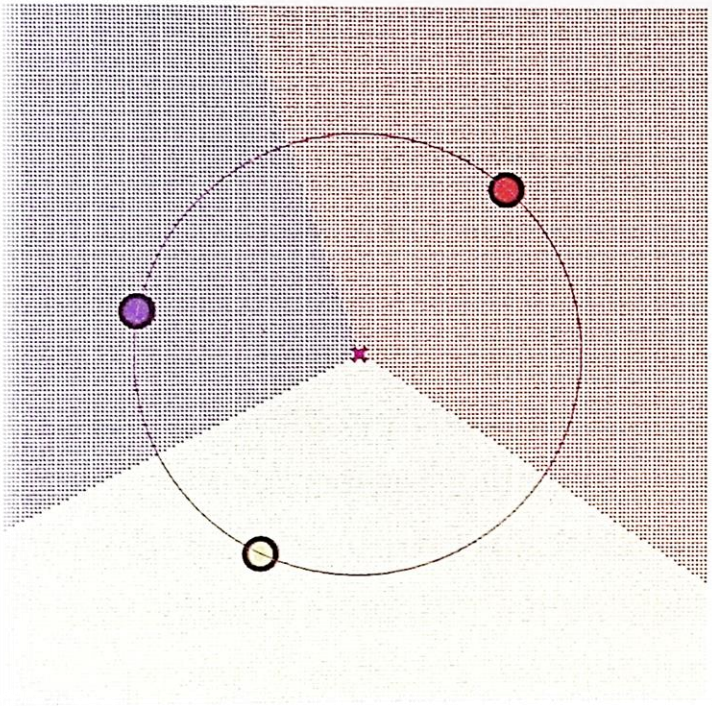


- Let $\mathbf{x}$ be an instance and $\boldsymbol{L} = \{c_1, \dots, c_m\}$ the possible classes
- Let $\mathrm{N}_k(\mathbf{x}) = \{\mathbf{x}_1, \dots, \mathbf{x}_k\}$ be the k nearest labeled neighbors of $\mathbf{x}$ (according to some metric)
- Classify $\mathbf{x}$ as

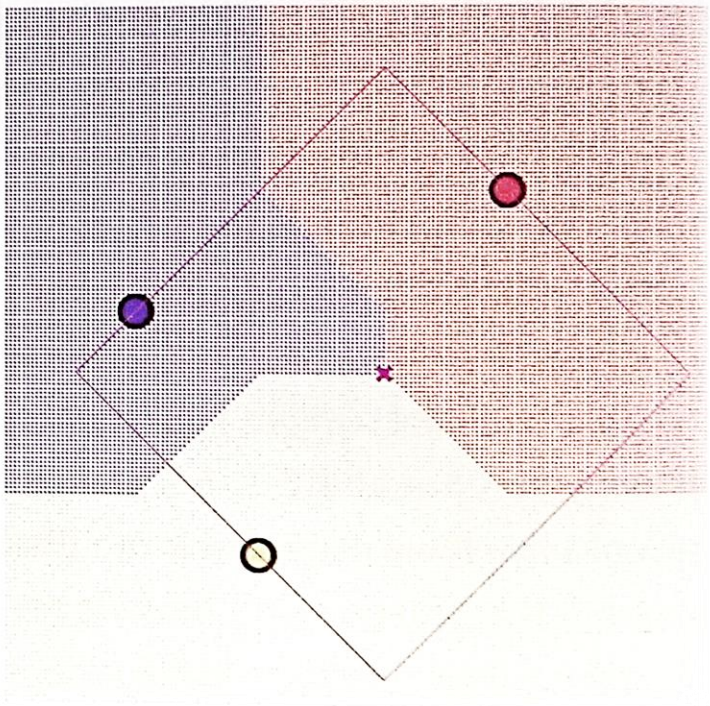
$$\underset{c \in \boldsymbol{L}}{\operatorname{argmax}} \sum_{\mathbf{x}_i \in \mathrm{N}_k(\mathbf{x})} w_i [\![l(\mathbf{x}_i) = c]\!]$$

$$w_i := \frac{1}{d(\mathbf{x}, \mathbf{x}_i)^2}$$

Example: Euclidean vs. Manhattan Distance



Linear decision regions



Non-convex decision regions

Source: Machine Learning by P. Flach

Remarks on k-NN

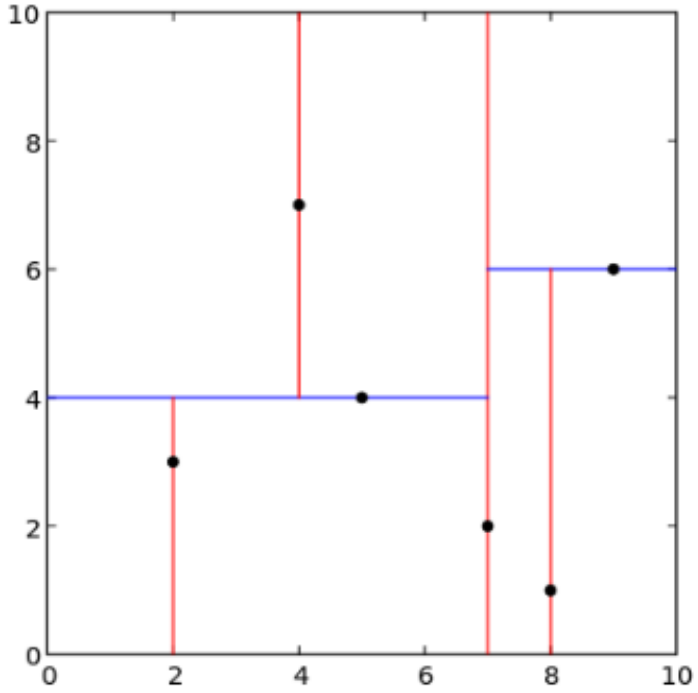
- "Lazy" multi-class classifier, almost Bayes optimal (with twice Bayes error rate as error upper bound)
- Very good performance if training instances from each class approach the true distribution of the class
- Can be computationally intensive (efficient computation of k-nearest neighbors is needed)
- All features/attributes are equally important (but in practical learning situations some features are more important than others)
- Highly susceptible to the "curse of dimensionality"

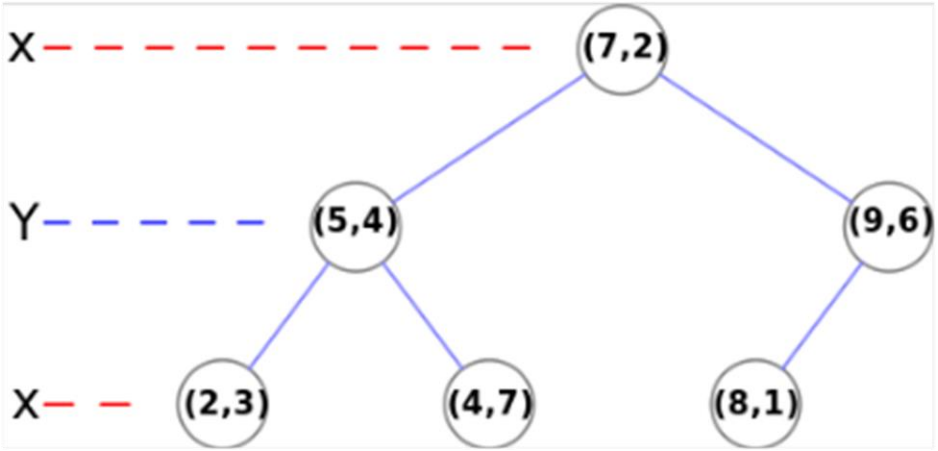
Efficient techniques for finding k-NN

- Naive space partitioning and exhaustive search in near-by partitions
- *K*-d trees
  - Space partitioning in-memory data structure for high-dimensional data
- R-trees (if data points have spatial extension)
  - On-disc data structure for indexing an overlapping partitioning of the data space; search is done based on bounding boxes and spatial joins
- Locality sensitive hashing for approximate k-NN search
  - Data points are mapped to lower dimensional space so that topology (according to some metric or similarity) is maintained

K-d trees

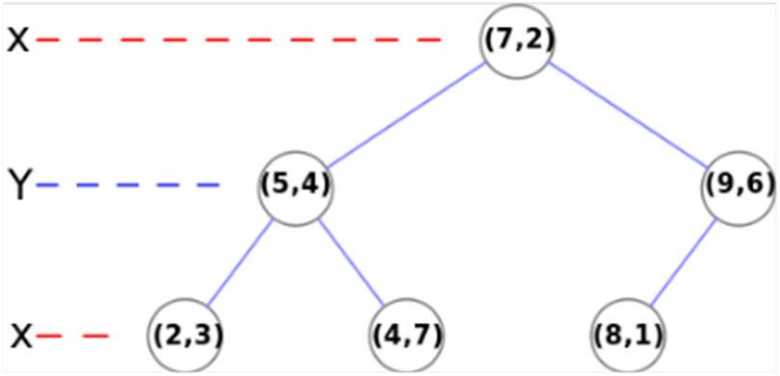
- Iterate over the dimensions and find the best splitting point(s) in each dimension

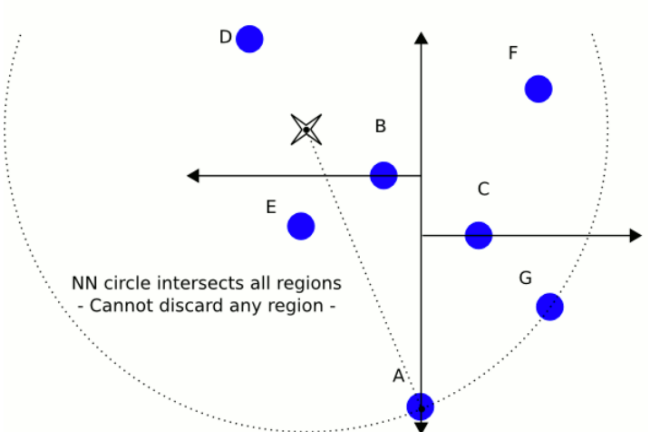




Source: Wikipedia

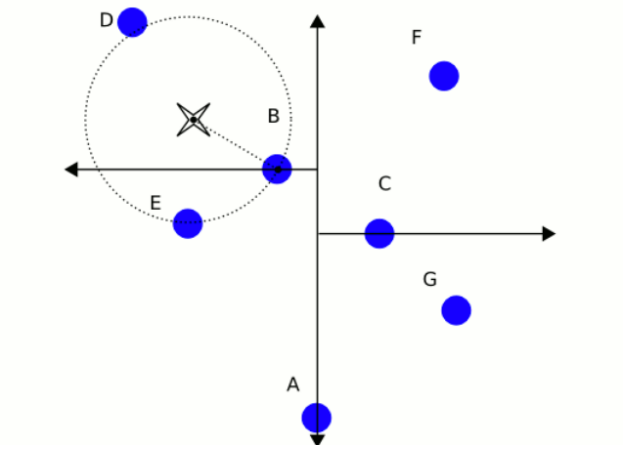
kNN search with K-d trees





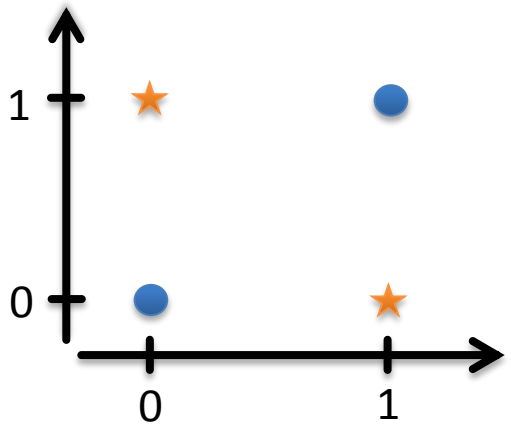
- k-NN search based on
  Euclidean distance by iteratively
  decreasing a bounding sphere
  until k nearest neighbors are found

- What is the complexity of k-NN search?



Source: Wikipedia

Rule-based classification



$$\big(x(e) \wedge \neg y(e)\big) \vee \big(y(e) \wedge \neg x(e)\big) \Rightarrow e = \bigstar$$
$$\big(\neg y(e) \wedge \neg x(e)\big) \vee \big(x(e) \wedge y(e)\big) \Rightarrow e = \bullet$$

$$(color(e) = red\ ) \Rightarrow e = apple$$
$$\big((color(e) = yellow) \wedge (weight(e) < 150g)\big) \Rightarrow e = pear$$
$$\big((color(e) = yellow) \wedge (weight(e) \geq 150g)\big) \Rightarrow e = apple$$

How do we know whether rules will perform well for a given classification task?

Association rules

- Rules of the form $\mathcal{A} \Rightarrow \mathcal{C}$ (for classification: $\mathcal{A}$ is a set of feature values, $\mathcal{C}$ is a single class or a set of feature values), $\mathcal{A}$ is called **antecedent**, $\mathcal{C}$ is called **consequent**

- **Support/coverage of an association rule** $\mathcal{A} \Rightarrow \mathcal{C}$: Relative number of cases for which implication is true, denoted by $supp(\mathcal{A} \Rightarrow \mathcal{C}) = \frac{\#(\mathcal{A} \cup \mathcal{C})}{n}$, where n is the number of all cases

- **Confidence/accuracy of an association rule** $\mathcal{A} \Rightarrow \mathcal{C}$:

$$conf(\mathcal{A} \Rightarrow \mathcal{C}) = \frac{supp(\mathcal{A} \Rightarrow \mathcal{C})}{supp(\mathcal{A} \Rightarrow *)}$$

- **Lift of an association rule** $\mathcal{A} \Rightarrow \mathcal{C}$:

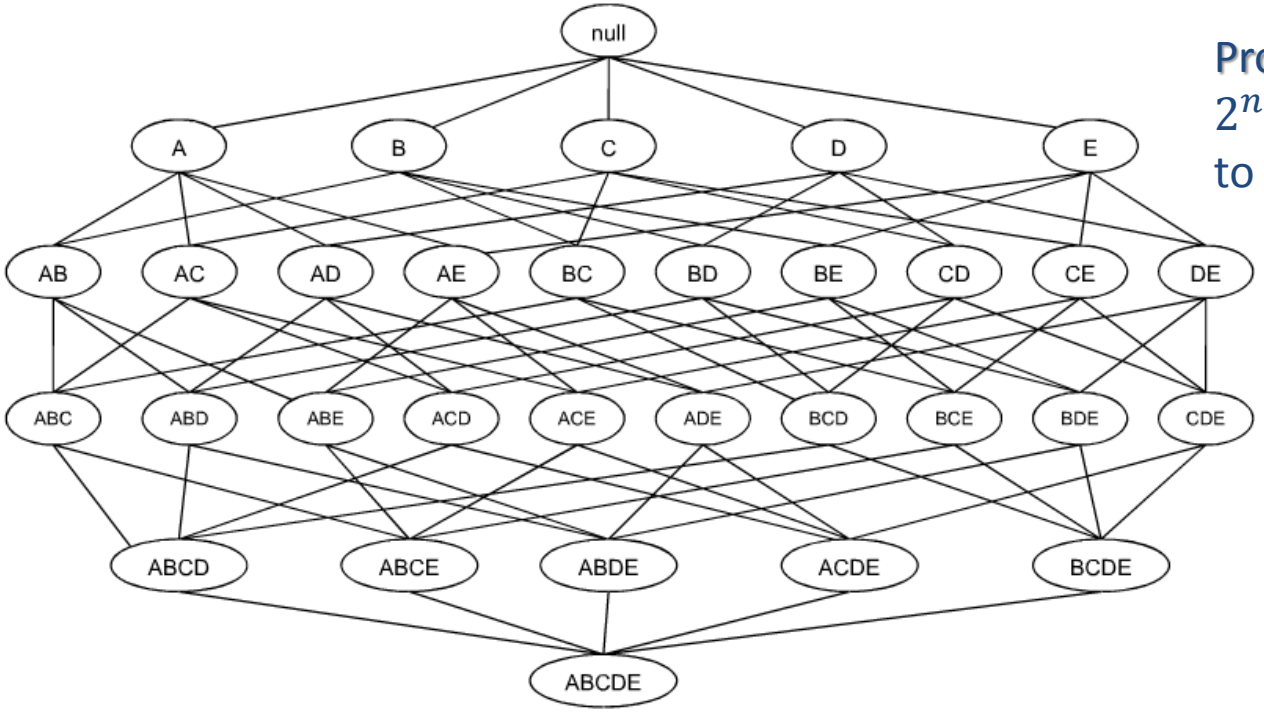
$$lift(\mathcal{A} \Rightarrow \mathcal{C}) = \frac{supp(\mathcal{A} \Rightarrow \mathcal{C})}{supp(\mathcal{A} \Rightarrow *)supp(\mathcal{C} \Rightarrow *)}$$

- Association rules can be used to predict classes from feature values or associations between feature values

Association rule mining

- Association rule-mining techniques consist of two steps
- Let I be the set of all feature values and classes, i.e., the **item set**
  1. Find all frequent subsets S of I (i.e. with $supp(S) \geq \theta$)
  2. Generate association rules R from the frequent subsets with $conf(R) > \gamma$

  For step 1, generate **occurrence lattice** of items and identify frequent subsets:

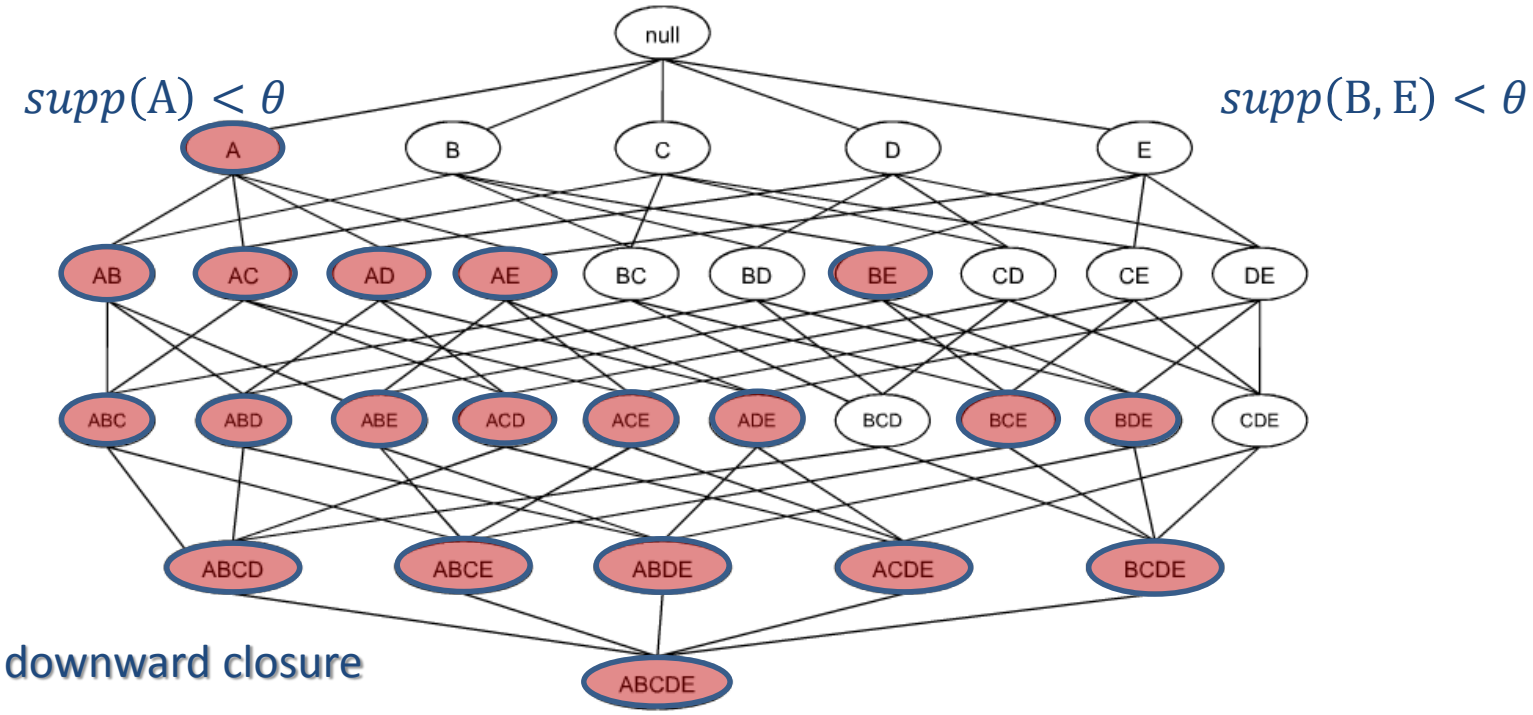


Problem:
$2^n - 1$ subsets to be checked!

Association rule mining

- Association rule-mining techniques consist of two steps
- Let I be the set of all feature values and classes, i.e., the item set
  1. Find all frequent subsets S of I (i.e. with $supp(S) \geq \theta$)
  2. Generate association rules R from the frequent subsets with $conf(R) > \gamma$

  For step 1, generate occurrence lattice of items and identify frequent subsets:



Apriori algorithm

- Find all frequent subsets (i.e., with support $\geq \theta$) by exploiting **downward closure** property:

```
Set l := 1;
Find frequent subsets of size l;
While subsets S of size l with supp(S) ≥ θ are found
    l := l + 1;
    Generate subsets S of size l with supp(S) ≥ θ
    by combining subsets of size l − 1
```

- Generate association rules R from the frequent subsets with $conf(R) > \gamma$

```
For each frequent subset S
    Find all non-empty subsets A of S such that
    conf(A ⇒ S\A) > γ
```

Remarks on the Apriori algorithm

- For a subset of feature values $\mathcal{A} = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$:
  $conf(\{\mathrm{A}, \mathrm{B}, \mathrm{C}\} \Rightarrow \{\mathrm{D}\}) \geq conf(\{\mathrm{A}, \mathrm{B}\} \Rightarrow \{\mathrm{C}, \mathrm{D}\}) \geq conf(\{\mathrm{A}\} \Rightarrow \{\mathrm{B}, \mathrm{C}, \mathrm{D}\})$
- This kind of "downward closure property" can be used again for pruning

- Although on many practical datasets, the algorithm runs efficiently, its runtime complexity (for $|I| = n$) is $O(2^n)$

- Runtime in practice is highly sensitive to the choice of θ and γ

- Algorithm can be used for different goals
  - Derive horn clauses (i.e., conjunctive deduction rules) for class prediction
  - Grouping of features by association
  - Finding most salient features for representing a class

1-R algorithm

- Much simpler than association rules
- Idea: Construct one-level rules

```
For each feature
    For each value of that feature
        Find most frequent class
        Make rule assign class to this value
        Calculate error rate of rule
    Calculate error rate for feature
Choose rules from feature with smallest relative error rate
```

- Can also be used to discretize features in a supervised fashion
- How?

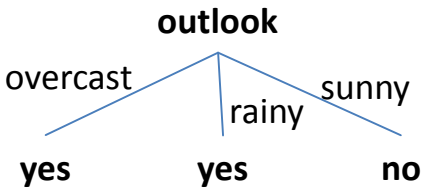
Example of rule learning with the 1-R algorithm

- Source: Data Mining, Practical Machine Learning Tools and Techniques by I. Witten, E. Frank, M. Hall

Will event X take place?

feature	rules	error	error per feature
outlook	sunny → no	2/5	4/14
	overcast → yes	0/4	
	rainy → yes	2/5	
temperature	hot → no	2/4	5/14
	mild → yes	2/6	
	cool → yes	1/4	
humidity	high → no	3/7	4/14
	normal → yes	1/7	
windy	false → yes	2/8	5/14
	true → no	3/6	

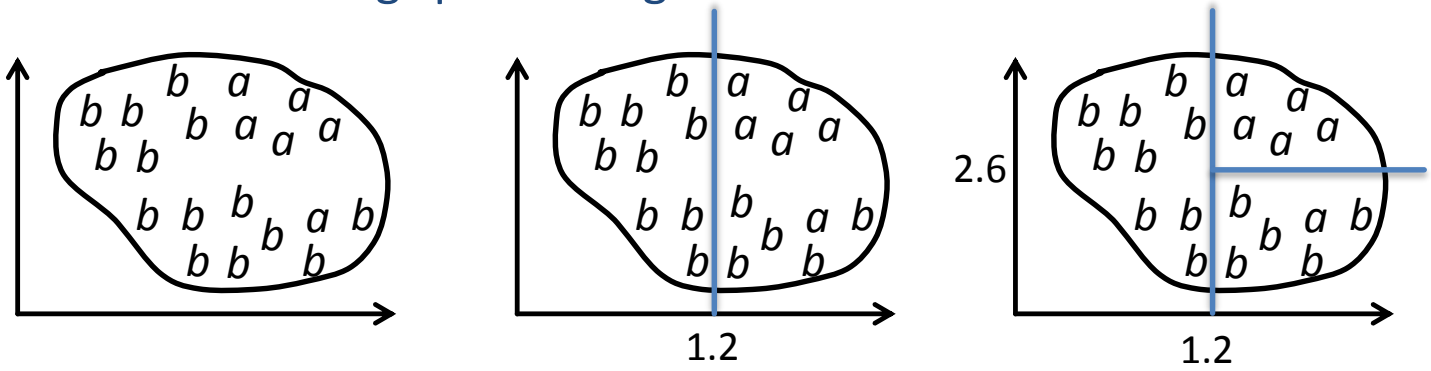
One-level decision tree



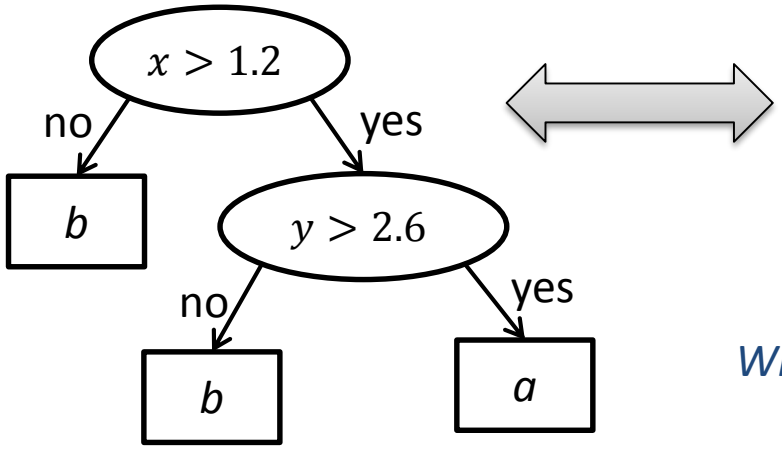
Decision tree stump

Hierarchical classifiers

- Divide the training space along feature dimensions:



- Construct **decision tree** for classes:



$R1: x \leq 1.2 \Rightarrow b$
$R2: x > 1.2 \wedge y \leq 2.6 \Rightarrow b$
$R3: \neg R1 \wedge \neg R2 \Rightarrow a$

Which dimension to choose for partitioning???

Rules can be inferred from tree paths

General decision tree algorithm

- Input: Set of labeled instances D; set of features F
- Output: Tree T with labelled leaves

```
growTree(D, F)
    If homogeneous(D)  //true if instances in D represent one class
        return class of D as labeled leaf
    S ← bestSplit(D, F)  //returns the most discriminative attribute
                         //of the instances in D
    Split D into subsets D_i  //according to the values of
                              //the attribute S
    For each i
        If D_i ≠ ∅ then T_i ← growTree(D_i, F)
        Else label T_i with the class of D
    Return tree
```

Finding the Best Split

- A set of instances D, is pure if all instances belong to the same class
- Let $Imp(D)$ denote the impurity of D
- For a partition $D_1, D_2, \dots, D_k$ of For D, the **purity gain** can be defined as

$$Imp(D) - \sum_{i=1}^{k} \frac{|D_i|}{|D|} Imp(D_i)$$

- For classes $c_1, \dots, c_m$ in D, popular measures for impurity:
  - **Entropy**

$$H(D) = \sum_{i=1}^{m} -\hat{p}_i \log \hat{p}_i$$

  - **Gini index**

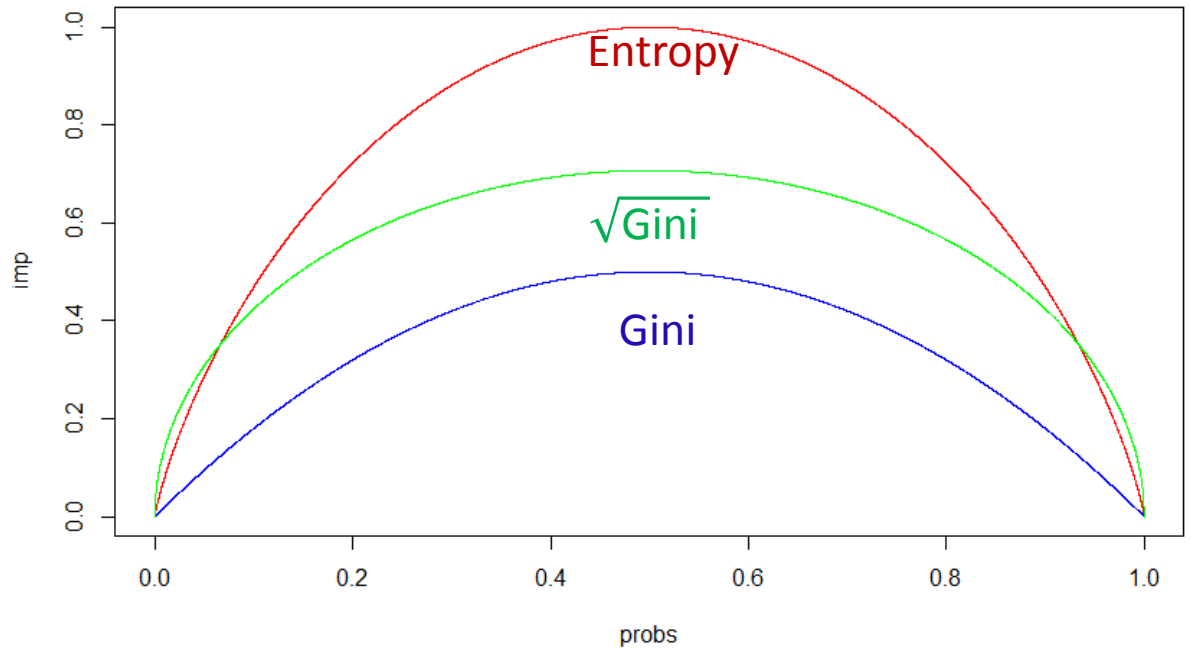
$$Gini(D) = \sum_{i=1}^{m} \hat{p}_i (1 - \hat{p}_i)$$

  - **Square root of Gini index**

$$\sqrt{Gini(D)}$$

$\hat{p}_i$: probability estimate of the class c_i in D

Entropy vs. Gini index



- $\sqrt{\text{Gini}}$ is insensitive to fluctuations in the class distribution (i.e., the relative impurity of the child w.r.t. its parent, $\frac{Imp(child)}{Imp(parent)}$, does not change if class distribution changes)
- Entropy and Gini index are sensitive to such fluctuations
- **Information gain** is the purity gain in terms of Entropy

General algorithm for the best split

- Input: Set of data instances D, set of features F
- Output: Feature f to split on

```
bestSplit(D, F)
    ImpGain = 0
    split = ∅
    For each f ∈ F
        Partition D into D_1, ..., D_m according to values of f
        If Imp(D) − Imp(D_1, ..., D_m) > ImpGain
            ImpGain = Imp(D) − Imp(D_1, ..., D_m)
            split = {f, D_1, ..., D_m}
    Return split
```

Information gain

$$IG(D, f) = H(D) - H(D|f) = H(D) - \sum_{v \in V(f)} \frac{|D_v|}{|D|} H(D_v)$$

- $H(D)$: Entropy in partition D

$$H(D) = -\sum_j \frac{n_{D,j}}{|D|} \log\left(\frac{n_{D,j}}{|D|}\right)$$

- $V(f)$: Values of attribute f,
- D_v: Instances in D with value v for f
- $n_{D,j}$: Number of instances belonging to class j in partition D

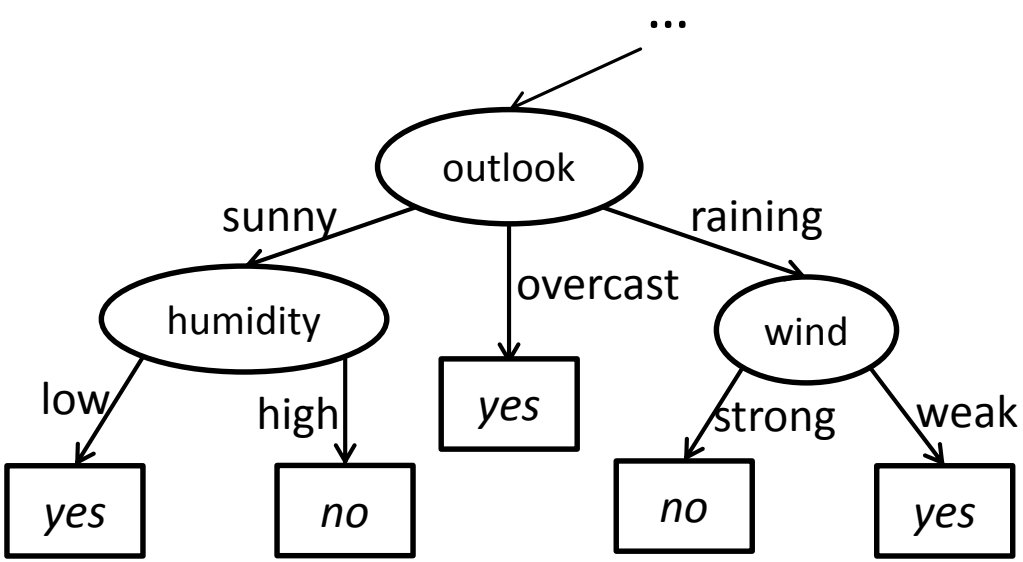
Split by information gain: Example

- Locally: Chosen split dimension should maximize information gain
- Globally: Tree should be as pure as possible, i.e., maximum purity is achieved when each leaf represents a single class

$$H(D) = -\frac{3}{5}\log\left(\frac{3}{5}\right) - \frac{2}{5}\log\left(\frac{2}{5}\right) = 0.971$$

$$H(D) - H(D|wind) = 0.971$$

$$H(D) - H(D|humidity) = 0.02$$



Training examples

class ↓

outlook	humidity	wind	golf
...			
rain	high	weak	yes
rain	normal	weak	yes
...			
rain	normal	strong	no
rain	normal	weak	yes
rain	high	strong	no

Decision tree algorithm for continuous attribute values

BuildTree(D, F) *//D: current node representing the data, F: features*

If D contains only training data of the same class

Terminate

Determine split dimension $f \in F$ //e.g. f maximizes inf. gain

Determine split value x of f //with respect to split value x

$D_1 = D \cap \{d \mid d.f \leq x\}$ and $D_2 = D \cap \{d \mid d.f > x\}$

BuildTree(D_1, F); BuildTree(D_2, F)

- Average complexity (for m attributes and n training examples) is $O(mn\log(n))$

Decision tree pruning

- Important to mitigate overfitting

- **Bottom-up pruning strategy**
  1. Start at the leaves and replace a subtree with its most popular class
  2. If the prediction accuracy is not affected then the change is kept

- **Incomplete/impure tree induction**
  1. Build the tree in top-down fashion
  2. Test at each current leaf if impurity is below some threshold (alternatively, do not expand a new leaf if it does not increase prediction accuracy for the corresponding attribute)

Bottom-up pruning algorithm

For every internal node N
    $T_N \leftarrow$ tree rooted at N
    $D_N \leftarrow$ Data represented by N
    If accuracy of T_N over D_N is worse than accuracy of predicting majority class c^* in D_N
        Replace T_N in T by a leaf labelled with c^*
Return pruned version of T

- Average complexity (for tree size n training examples) is $O(n \log(n))$

Remarks on decision trees

- Can be turned into probabilistic ranking classifiers by ordering leaves in non-decreasing order of empirical class probabilities
- Laplace smoothing can be applied to make estimates more robust for small leaves

- Easy to interpret and explain (rules can be read off of paths)
- Very good performance on discrete attribute domains

- Danger of **overfitting** (e.g. high purity from few training samples)
- Performance degrades for continuous attribute domains
- Size can become relatively large

- Different implementations: ID3 (Entropy-based), C4.5 (Gini-based), MARS

Ensemble learners: Random forests (L. Breiman'01)

BuildForest(training points:$\mathbf{x}_1,..,\mathbf{x}_N$, features:$f_1,..,f_M$

Guess $m \ll M$

For each tree $T \in \{T_1, \dots, T_k\}$

Choose S_N out of N training points by sampling N times with replacement //i.e., bootstrapping

Grow(S_N, m, T)

Grow(S_N, m, T)

If $S_N = \emptyset$ return T

Randomly choose m features

//hopefully the best m decision features

Compute the best split on the S_N training points based on the m features //e.g. split that maximizes inf. //gain

Add new nodes S_{N_1} ,S_{N_2} as children of S_N

Grow(S_{N_1}, m, T); Grow(S_{N_2}, m, T)

Remarks on random forests

- The error rate of each tree is computed on the remaining test points
- For a new sample, push it down each tree and compute weighted average from all predictions

- More robust to noise than decision trees
- Less susceptible to overfitting
- No pruning is needed
- Relation between feature subsets and classification accuracy is revealed
- Can be easily parallelized
- Empirically shown to be one of the most accurate learning methods

- Difficult to interpret
- Maintenance and governance of large data structures

Ensemble learners: Boosting

- Combination of many simple/weak learners (typically binary classifiers which classify better than a random class assignment based on very few features) to a single strong learner

- **Example**
  - For data instance $\mathbf{x} = (x_1, \dots, x_m)$, a weak classifier M_j can predict as follows:

$$M_j(\mathbf{x}) = \begin{cases} 1, x_j > \theta_j \\ -1, x_j \leq \theta_j \end{cases}$$

  - Define $M := sign\left(\sum_j \alpha_j M_j(\mathbf{x})\right)$, where α_j is the weight of the *j*'th classifier

  - Let $\mathbf{x}_1, \dots, \mathbf{x}_n$ be training instances with ground truth labels $t_1, \dots, t_n \in \{-1,1\}$
  - Minimize $\sum_i w_{ji} e^{-t_i M_j(\mathbf{x}_i)}$ (the so-called **exponential loss function**)

Boosting algorithm

- Input: Training instances $\mathbf{x}_1, \ldots, \mathbf{x}_n$, ensemble size T
- Output: Ensemble model $M(\mathbf{x}) = sign\left(\sum_{j=1}^{T} \alpha_j M_j(\mathbf{x})\right)$

$w_{1i} \coloneqq \frac{1}{n}$ for each $\mathbf{x}_i$; $\alpha_j \coloneqq 1.0$ for each $M_j, 1 \leq j \leq T$

For $j = 1$ to T

Run $M_{curr}(\mathbf{x}_i) = sign\left(\sum_{k=1}^{j} \alpha_k M_k(\mathbf{x}_i)\right)$ on all $\mathbf{x}_i$

Calculate weighted error ϵ_j of M_j

If $\epsilon_j \geq \frac{1}{2}$ //$\epsilon_j \coloneqq \sum_i w_{ji} [\![t_i \neq M_j(\mathbf{x}_i)]\!]$

$j \coloneqq j - 1$ and break //previous ensemble was better

$\alpha_j \coloneqq \frac{1}{2} \ln \frac{1-\epsilon_j}{\epsilon_j}$ //confidence of the model

For each misclassified instance $\mathbf{x}_m$

$w_{(j+1)m} \coloneqq \frac{w_{jm}}{2\epsilon_j}$ //increase weight

For each correctly classified instance $\mathbf{x}_c$

$w_{(j+1)c} \coloneqq \frac{w_{jc}}{2(1-\epsilon_j)}$ //decrease weight

Boosting formal

- The algorithm greedily minimizes loss for each of the *j* classifiers

$$Z_j = \sum_{\mathbf{x}_i} w_{ji} \exp\left(-\alpha_j t_i M_j(\mathbf{x}_i)\right) \approx \epsilon_j \exp(\alpha_j) + (1-\epsilon_j)\exp(-\alpha_j)$$

- Taking first derivative for α_j, setting it to 0, and solving for α_j yields

$$\alpha_j := \frac{1}{2}\ln\frac{1-\epsilon_j}{\epsilon_j} \text{ and } Z_j = 2\sqrt{\epsilon_j(1-\epsilon_j)} \quad (\sqrt{\text{Gini}})$$

Remarks on Boosting

- Implicitly derives the “reliability” of features that should be considered for the classification task (e.g., in case that each single classifier decides based on one feature)
- Bias-reduction by adding in each iteration a new classifier that aims to correct previous misclassifications (the goal for an ensemble is to classify instances correctly on average)
- Easy to implement and efficient algorithm with good empirical performance