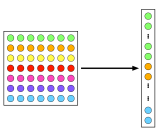
Some Notation

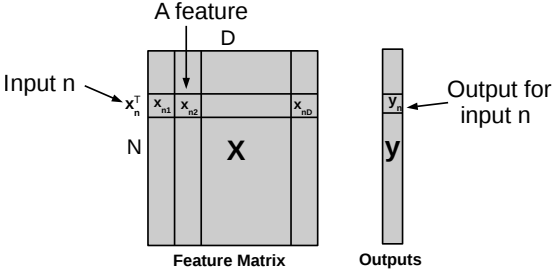
- Supervised Learning requires training data given as a set of input-output pairs $\{(\boldsymbol{x}_n, y_n)\}_{n=1}^N$
- Unsupervised Learning requires training data given as a set of inputs $\{\boldsymbol{x}_n\}_{n=1}^N$
- Each input $\boldsymbol{x}_n$ is (usually) a vector containing the values of the features or attributes or covariates that encode properties of the data it represents, e.g.,
  - Representing a 7×7 image: $\boldsymbol{x}_n$ can be a 49×1 vector of pixel intensities



  - Note: Good features can also be learned from data (feature learning) or extracted using hand-crafted rules defined by a domain expert. **Having a good set of features is half the battle won!**
- Each y_n is the output or response or label associated with input $\boldsymbol{x}_n$
  - The output y_n can be a scalar, a vector of numbers, or a structured object (more on this later)

Some Notation

- Will assume each input $\boldsymbol{x}_n$ to be a $D \times 1$ column vector (its transpose $\boldsymbol{x}_n^\top$ will be row vector)
- x_{nd} will denote the d-th feature of the n-th input
- We will use $\mathbf{X}$ ($N \times D$ feature matrix) to collectively denote all the N inputs
- We will use $\boldsymbol{y}$ ($N \times 1$ output/response/label vector) to collectively denote all the N outputs



- Note: If each y_n itself is a vector (we will see such cases later) then we will use a matrix $\mathbf{Y}$ to collectively denote all the N outputs (with row n containing y_n) and also use boldfaced $\boldsymbol{y}_n$

Types of Features and Types of Outputs

- Features (in vector $\boldsymbol{x}_n$) as well as outputs y_n can be real-valued, binary, categorical, ordinal, etc.
- Real-valued: Pixel intensity, house area, house price, rainfall amount, temperature, etc
- Binary: Male/female, adult/non-adult, or any yes/no or present/absent type values
- Categorical/Discrete: Pincode, bloodgroup, or any "which one from this finite set" type values
- Ordinal: Grade (A/B/C etc.) in a course, or any other type where relative values matters
- Often, the features can be of mixed types (some real, some categorical, some ordinal, etc.)
- Appropriate handling of different types of features may be very important (even if you algorithm is designed to "learn" good features, given a set of heterogeneous features)
- In Sup. Learning, different types of outputs may require different type of learning models

Supervised Learning

- Supervised Learning comes in many flavors. The flavor depends on the type of each output y_n
- Regression: $y_n \in \mathbb{R}$ (real-valued scalar)
- Multi-Output Regression: $\boldsymbol{y}_n \in \mathbb{R}^M$ (real-valued vector containing M outputs)

0.3	0.1	0.2	0.8	0.4

Illustration of a 5-dim output vector
for a multi-output regression problem

- Binary Classification: $y_n \in \{-1, +1\}$ or $\{0, 1\}$ (output in classification is also called "label")
- Multi-class Classification: $y_n \in \{1, 2, \ldots, M\}$ or $\{0, 1, \ldots, M-1\}$ (one of M classes is correct label)

0	0	0	1	0

Illustration of a 5-dim one-hot label vector
for a multi-class classification problem

- Multi-label Classification: $y_n \in \{-1, +1\}^M$ or $\{0, 1\}^M$ (a subset of M labels are correct)

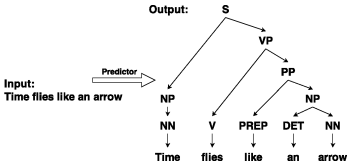
1	0	1	0	0

Illustration of a 5-dim binary label vector
for a multi-label classification problem
(unlike one-hot, there can be multiple 1s)

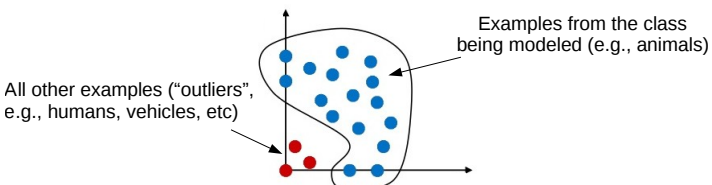
- Note: Multi-label classification is also informally called "tagging" (especially in Computer Vision)

Supervised Learning (Contd.)

- Structured-Prediction (a.k.a. Structured Output Learning): Each y_n is a structured object



- One-Class Classification (a.k.a. outlier/anomaly/novelty detection): y_n is "1" or "everything else"



- Ranking: Each y_n is a ranked list of relevant stuff for a given input/query $\boldsymbol{x}$

Computing Distances/Similarities

- Assuming all real-valued features, an input $\boldsymbol{x}_n \in \mathbb{R}^{D\times 1}$ is a point in a D dim. vector space of reals
- Standard rules of vector algebra apply on such representations, e.g.,
  - Euclidean distance b/w two points (say two images or two documents) $\boldsymbol{x}_n \in \mathbb{R}^D$ and $\boldsymbol{x}_m \in \mathbb{R}^D$

$$d(\boldsymbol{x}_n, \boldsymbol{x}_m) = ||\boldsymbol{x}_n - \boldsymbol{x}_m|| = \sqrt{(\boldsymbol{x}_n - \boldsymbol{x}_m)^\top (\boldsymbol{x}_n - \boldsymbol{x}_m)} = \sqrt{\sum_{d=1}^{D} (x_{nd} - x_{md})^2}$$

  - Inner-product similarity b/w $\boldsymbol{x}_n$ and $\boldsymbol{x}_m$ (cosine, $\boldsymbol{x}_n$, $\boldsymbol{x}_m$ are unit-length vectors)

$$s(\boldsymbol{x}_n, \boldsymbol{x}_m) = \langle \boldsymbol{x}_n, \boldsymbol{x}_m \rangle = \boldsymbol{x}_n^\top \boldsymbol{x}_m = \sum_{d=1}^{D} x_{nd} x_{md}$$

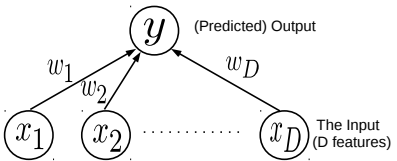
  - ℓ_1 distance between two points $\boldsymbol{x}_n$ and $\boldsymbol{x}_m$

$$d_1(\boldsymbol{x}_n, \boldsymbol{x}_m) = ||\boldsymbol{x}_n - \boldsymbol{x}_m||_1 = \sum_{d=1}^{D} |x_{nd} - x_{md}|$$

Linear Models

- Consider learning to map an input $\boldsymbol{x} \in \mathbb{R}^D$ to its output y (say real-valued)
- Assume the output to be a linear weighted combination of the D input features

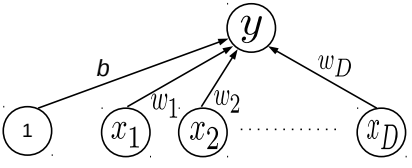
$$y = \sum_{d=1}^{D} w_d x_d = \boldsymbol{w}^\top \boldsymbol{x}$$



- This is an example of a linear model with D parameters $\boldsymbol{w} = [w_1, w_2, \ldots, w_D]$
- Inspired by **linear models of neurons**
- $\boldsymbol{w} \in \mathbb{R}^D$ is also known as the weight vector
- Here w_d denotes how important the d-th input feature is for predicting y
- The above is basically a linear model for simple regression (single, real-valued output y)
- This basic model can also be used as building blocks in many more complex models

Linear Models with Offset (Bias) Parameter

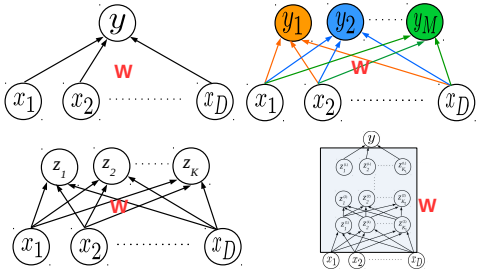
- Some linear models use an additional bias parameter b



$$y = \sum_{d=1}^{D} w_d x_d + b = \boldsymbol{w}^\top \boldsymbol{x} + b$$

- Can append a constant feature "1" for each input and rewrite as $y = \boldsymbol{w}^\top \boldsymbol{x}$, with $\boldsymbol{x}, \boldsymbol{w} \in \mathbb{R}^{D+1}$
- We will assume the same and omit the explicit bias for simplicity of notation

Learning Linear Models



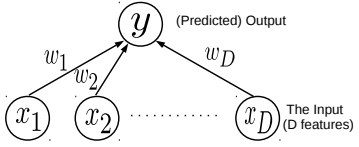
? How do we learn them from data

For linear models, learning = Learning the model parameters (the weights) We will formulate learning as an optimization problem w.r.t. these parameters

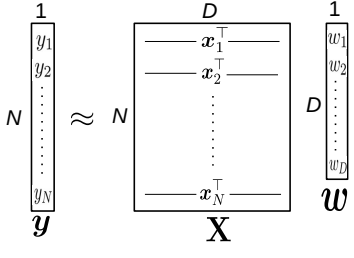
Learning a Linear Model for Regression

- Let's focus on learning the simplest linear model for now: Linear Regression

$$y = \sum_{d=1}^{D} w_d x_d = \boldsymbol{w}^\top \boldsymbol{x}$$

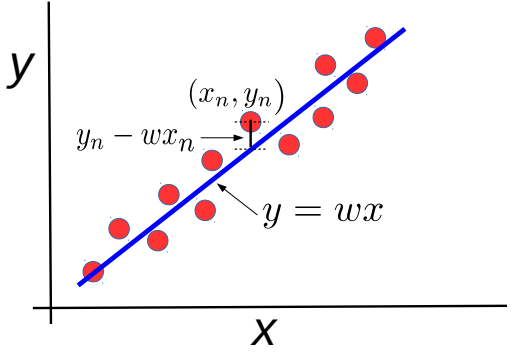


- Suppose we are given regression training data $\{(\boldsymbol{x}_n, y_n)\}_{n=1}^{N}$ with $\boldsymbol{x}_n \in \mathbb{R}^D$, and $y_n \in \mathbb{R}$
- Let's model the training data using $\boldsymbol{w}$ and assume $y_n \approx \boldsymbol{w}^\top \boldsymbol{x}_n$, $\forall n$ (equivalently $\boldsymbol{y} \approx \mathbf{X}\boldsymbol{w}$)



:Linear Regression

- With one-dimensional inputs, linear regression would look like



- Error of the model for an example $= y_n - \boldsymbol{w}^\top \boldsymbol{x}_n$ ($= y_n - wx_n$ for scalar input case)

Linear Regression

- Define the total error or "loss" on the training data, when using $\boldsymbol{w}$ as our model, as

$$\mathcal{L}(\boldsymbol{w}) = \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

- Note: Squared loss chosen for simplicity. Can define other type of losses too (more on this later)
- The best $\boldsymbol{w}$ will be the one that minimizes the above error (requires optimization w.r.t. $\boldsymbol{w}$)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

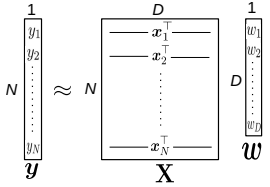
- This is known as "least squares" linear regression (Gauss/Legendre, early 18th century)
- Taking derivative (gradient) of $\mathcal{L}(\boldsymbol{w})$ w.r.t. $\boldsymbol{w}$ and setting to zero

$$\sum_{n=1}^{N} 2(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n) \frac{\partial}{\partial \boldsymbol{w}} (y_n - \boldsymbol{x}_n^\top \boldsymbol{w}) = 0 \quad \Rightarrow \quad \sum_{n=1}^{N} \boldsymbol{x}_n (y_n - \boldsymbol{x}_n^\top \boldsymbol{w}) = 0$$

- Simplifying further, we get a solution for $\boldsymbol{w} \in \mathbb{R}^D$

$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top)^{-1} \sum_{n=1}^{N} y_n \boldsymbol{x}_n = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{y}$$

Linear Regression



- Consider the closed form solution we obtained for linear regression based on least squares

$$\boldsymbol{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- The above closed form solution is nice but has some issues
  - The $D \times D$ matrix $\mathbf{X}^\top \mathbf{X}$ may not be invertible
  - Based solely on minimizing the training error $\sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 \Rightarrow$ can overfit the training data
  - Expensive inversion for large D: Can used iterative optimization techniques (will come to this later)

Regularized Linear Regression (a.k.a. Ridge Regression)

- Consider regularized loss: Training error + ℓ_2-squared norm of $\boldsymbol{w}$, i.e., $||\boldsymbol{w}||_2^2 = \boldsymbol{w}^\top \boldsymbol{w} = \sum_{d=1}^{D} w_d^2$

$$\mathcal{L}_{reg}(\boldsymbol{w}) = \left[\sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}\right]$$

- Minimizing the above objective w.r.t. $\boldsymbol{w}$ does two things
  - Keeps the training error small
  - Keeps the ℓ_2 norm of $\boldsymbol{w}$ small (and thus also the individual components of $\boldsymbol{w}$): **Regularization**
- There is a trade-off between the two terms: The regularization hyperparam $\lambda > 0$ controls it
  - Very small λ means almost no regularization (can overfit)
  - Very large λ means very high regularization (can underfit - high training error)
  - Can use cross-validation to choose the "right" λ
- The solution to the above optimization problem is: $\boldsymbol{w} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$
- Note that, in this case, regularization also made inversion possible (note the $\lambda \mathbf{I}_D$ term)

How ℓ_2 Regularization Helps Here?

- We saw that ℓ_2 regularization encourages the individual weights in $\boldsymbol{w}$ to be small
- Small weights ensure that the function $y = f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$ is smooth (i.e., we expect similar $\boldsymbol{x}$'s to have similar y's). Below is an informal justification:
- Consider two points $\boldsymbol{x}_n \in \mathbb{R}^D$ and $\boldsymbol{x}_m \in \mathbb{R}^D$ that are exactly similar in all features except the d-th feature where they differ by a small value, say ϵ
- Assuming a simple/smooth function $f(\boldsymbol{x})$, y_n and y_m should also be close
- However, as per the model $y = f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$, y_n and y_m will differ by ϵw_d
- Unless we constrain w_d to have a small value, the difference ϵw_d would also be very large (which isn't what we want).
- That's why regularizing (via ℓ_2 regularization) and making the individual components of the weight vector small helps

Regularization: Some Comments

- Many ways to regularize ML models (for linear as well as other models)
- Some are based on adding a norm of $\boldsymbol{w}$ to the loss function (as we already saw)
  - Using ℓ_2 norm in the loss function promotes the individual entries $\boldsymbol{w}$ to be small (we saw that)
  - Using ℓ_0 norm encourages very few non-zero entries in $\boldsymbol{w}$ (thereby promoting "sparse" $\boldsymbol{w}$)

$$||\boldsymbol{w}||_0 = \#\text{nnz}(\boldsymbol{w})$$

  - Optimizing with ℓ_0 is difficult (NP-hard problem); can use ℓ_1 norm as an approximation

$$||\boldsymbol{w}||_1 = \sum_{d=1}^{D} |w_d|$$

  - Note: Since they learn a sparse $\boldsymbol{w}$, ℓ_0 or ℓ_1 regularization is also useful for doing feature selection ($w_d = 0$ means feature d is irrelevant). We will revisit ℓ_1 later to formally see why ℓ_1 gives sparsity
-

Linear Regression as Solving System of Linear Equations

- Solving $\boldsymbol{y} = \mathbf{X}\boldsymbol{w}$ for $\boldsymbol{w}$ is like solving for D unknowns $w_1, \ldots, w_D$ using N equations

$$\begin{aligned} y_1 &= x_{11}w_1 + x_{12}w_2 + \ldots + x_{1D}w_D \\ y_2 &= x_{21}w_1 + x_{22}w_2 + \ldots + x_{2D}w_D \\ &\vdots \\ y_N &= x_{N1}w_1 + x_{N2}w_2 + \ldots + x_{ND}w_D \end{aligned}$$

- Can therefore view the linear regression problem as a system of linear equations
- However, in linear regression, we would rarely have $N = D$, but $N > D$ or $D > N$
- $N > D$ case is an overdetermined system of linear equations (# equations > # unknowns)
- $D > N$ case is an underdetermined system of linear equations (# unknowns > # equations)
- Thus methods to solve over/underdetermined systems can be used to solve linear regression as well
  - Many of these don't require a matrix inversion (will provide a separate note with details)

General Supervised Learning as Optimization

- We saw that regularized least squares regression required solving

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}_{reg}(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} \left[\sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \frac{\lambda}{2} \boldsymbol{w}^\top \boldsymbol{w} \right]$$

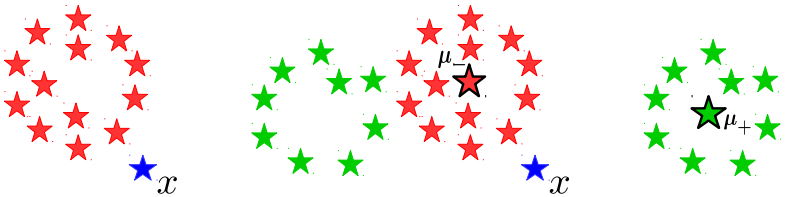
- This is essentially the training loss (called "empirical loss"), plus the regularization term
- In general, for supervised learning, the goal is to learn a function f, s.t. $f(\boldsymbol{x}_n) \approx y_n, \forall n$
- Moreover, we also want to have a simple f, i.e., have some regularization
- Therefore, learning the best f amounts to solving the following optimization problem

$$\hat{f} = \arg\min_{f} \mathcal{L}_{reg}(f) = \arg\min_{f} \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n)) + \lambda R(f)$$

  where $\ell(y_n, f(\boldsymbol{x}_n))$ measures the model f's training loss on $(\boldsymbol{x}_n, y_n)$ and $R(f)$ is a regularizer
- For least squares regression, $f(\boldsymbol{x}_n) = \boldsymbol{w}^\top \boldsymbol{x}$, and $R(f) = \boldsymbol{w}^\top \boldsymbol{w}$, and $\ell(y_n, f(\boldsymbol{x}_n)) = (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$
- As we'll see later, different supervised learning problems differ in the choice of f, $R(.)$, and ℓ

Prototype based Classification

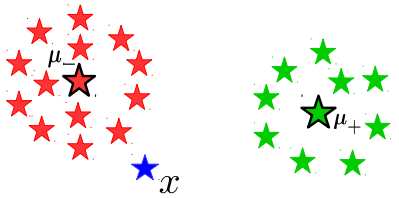
- Given: N labeled training examples $\{\boldsymbol{x}_n, y_n\}_{n=1}^{N}$ from two classes
  - Assume green is positive and red is negative class
  - N_+ exampes from positive class, N_- examples from negative class
- Our goal: Learn a model to predict label (class) y for a new test example $\boldsymbol{x}$



- A simple "distance from means" model: predict the class that has a closer mean
- Note: The basic idea easily generalizes to more than 2 classes as well

Prototype based Classification: More Formally

- What does the decision rule look like, mathematically ?



- The mean of each class is given by

$$\boldsymbol{\mu}_- = \frac{1}{N_-} \sum_{y_n=-1} \boldsymbol{x}_n \quad \text{and} \quad \boldsymbol{\mu}_+ = \frac{1}{N_+} \sum_{y_n=+1} \boldsymbol{x}_n$$

- Euclidean Distances from each mean are given by

$$\begin{aligned} ||\boldsymbol{\mu}_- - \boldsymbol{x}||^2 &= ||\boldsymbol{\mu}_-||^2 + ||\boldsymbol{x}||^2 - 2\langle \boldsymbol{\mu}_-, \boldsymbol{x} \rangle \\ ||\boldsymbol{\mu}_+ - \boldsymbol{x}||^2 &= ||\boldsymbol{\mu}_+||^2 + ||\boldsymbol{x}||^2 - 2\langle \boldsymbol{\mu}_+, \boldsymbol{x} \rangle \end{aligned}$$

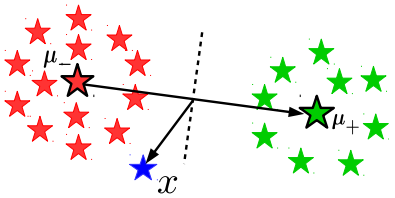
- **Decision Rule:** If $f(\boldsymbol{x}) := ||\boldsymbol{\mu}_- - \boldsymbol{x}||^2 - ||\boldsymbol{\mu}_+ - \boldsymbol{x}||^2 > 0$ then predict +1, otherwise predict -1

Prototype based Classification: The Decision Rule

- We saw that our decision rule was

$$f(\boldsymbol{x}) := ||\boldsymbol{\mu}_- - \boldsymbol{x}||^2 - ||\boldsymbol{\mu}_+ - \boldsymbol{x}||^2 = 2\langle \boldsymbol{\mu}_+ - \boldsymbol{\mu}_-, \boldsymbol{x} \rangle + ||\boldsymbol{\mu}_-||^2 - ||\boldsymbol{\mu}_+||^2$$

- **Imp.:** $f(\boldsymbol{x})$ effectively denotes a hyperplane based classification rule $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$ with the vector $\boldsymbol{w} = \boldsymbol{\mu}_+ - \boldsymbol{\mu}_-$ representing the direction normal to the hyperplane



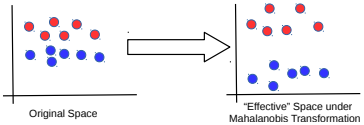
- **Imp.:** Can show that the rule is equivalent to $f(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n \langle \boldsymbol{x}_n, \boldsymbol{x} \rangle + b$, where α's and b can be estimated from training data **(try this as an exercise)**
  - This form of the decision rule is very important. Decision rules for many (in fact most) supervised learning algorithms can be written like this (weighted sum of similarities with all the training inputs)

Be Careful when Computing Distances

- Euclidean distance $d(\boldsymbol{x}_n, \boldsymbol{x}_m) = \sqrt{(\boldsymbol{x}_n - \boldsymbol{x}_m)^\top (\boldsymbol{x}_n - \boldsymbol{x}_m)}$ may not always be appropriate
- Another alternative (still Euclidean-like) can be to use the Mahalanobis distance

$$d_M(\boldsymbol{x}_n, \boldsymbol{x}_m) = \sqrt{(\boldsymbol{x}_n - \boldsymbol{x}_m)^\top \mathbf{M} (\boldsymbol{x}_n - \boldsymbol{x}_m)}$$

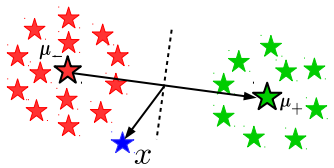
- Shown below is an illustration of what $\mathbf{M} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$ will do (note: figure not to scale)



- How do I know what's the right $\mathbf{M}$ for my data?Some options
  - Set it based on some knowledge of what you data looks like
  - Learn it from data (called Distance Metric Learning[1] - a whole reseach area in itself)
- Distance Metric Learning is one of the many approaches for feature learning from data

[1] Distance Metric Learning. See "A Survey on Metric Learning for Feature Vectors and Structured Data" by Ballet *et al*

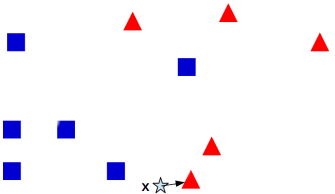
Prototype based Classification: Some Comments



- A very simple supervised learner. Works for any number of classes. Trivial to implement. :-)
- This simple approach, if using Euclidean distances, can only learn linear decision boundaries
  - A reason: The basic approach implicitly assumes that classes are roughly spherical and equi-sized
- Several nice improvements/generalizations possible (some of which we will see in coming lectures)
  - Instead of a point (mean), model classes by prob. distributions (to account for class shapes/sizes)
  - Instead of Euclidean distances, can use non-Euclidean distances, distance metric learning, or "kernels"
- Another limitation: Needs plenty of training data from each class to reliably estimate the means
  - But with a good feature learner, even ONE (or very few) example per class may be enough (a state-of-the-art "Few-Shot Learning" model actually uses Prototype based classification)

Nearest Neighbor

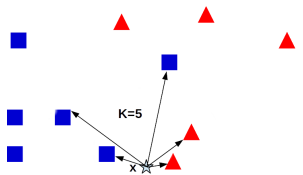
- Another classic distance-based supervised learning method



- The label y for $\boldsymbol{x} \in \mathbb{R}^D$ will be the label of its nearest neighbor in training data. Also known as one-nearest-neighbor (1-NN)
- Euclidean/Mahalanobis distance can be used to find the nearest neighbor (or can use a learned distance metric)
- We typically use more ($K > 1$) neighbors in practice
- Note: The method is widely applicable - works for both classification and regression problems

K-Nearest Neighbors (K-NN)

- Makes one-nearest-neighbor more robust by using more than one neighbor
- Test time simply does a majority vote (or average) of the labels of K closest training inputs



- For a test input $\boldsymbol{x}$, the averaging version of the prediction rule for K-nearest neighbors

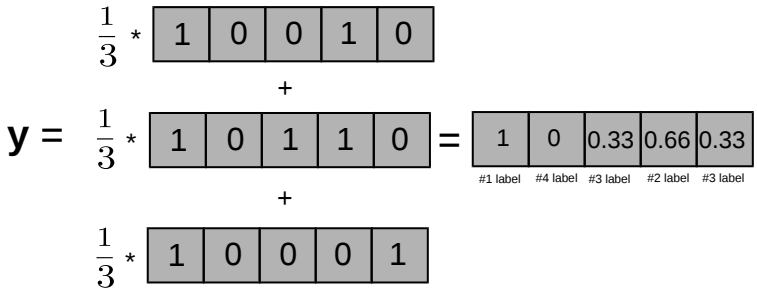
$$\boldsymbol{y} = \frac{1}{K} \sum_{n \in \mathcal{N}_K(\boldsymbol{x})} \boldsymbol{y}_n$$

.. where $\mathcal{N}_K(\boldsymbol{x})$ is the set of K closest training inputs for $\boldsymbol{x}$

- Above assumes the K neighbors have equal $(1/K)$ weights. Can also use distance-based weights
- Note: The rule works for multi-label classification too where each $\boldsymbol{y}_n \in \{0,1\}^M$ is a binary vector
  - Averaging will give a real-valued "label score vector" $\boldsymbol{y} \in \mathbb{R}^M$ using which we can find the best label(s)

K-NN for Multi-Label Learning: Pictorial Illustration

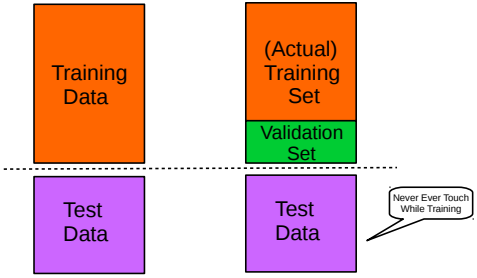
- Suppose $K = 3$. The label averaging for a multi-label learning problem will look like



- Note that we can use the final $\boldsymbol{y}$ to rank the labels based on the real-valued scores
  - Can use it to predict the best, best-2, best-3, and so on..
  - Note: This is why multi-label learning is often used in some ranking problems where we wish to predict a ranking of the possible labels an input can have

How to Select K: Cross-Validation

- We can use cross-validation to select the "optimal" value of K
- Cross-validation - Divide the training data into two parts: actual training set and a validation set



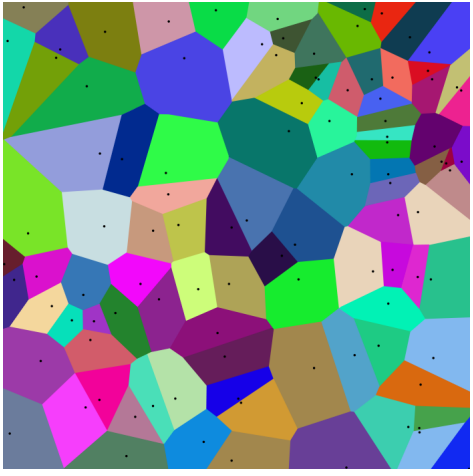
- Try different values of K and look at the accuracies on the validation set
  - Note: For each K, we typically try multiple splits of train and validation sets
- Select the K that gives the best accuracy on the validation set
- Never touch the test set (even if you have access to it) during training to choose the best K

Some Aspects about Nearest Neighbor

- A simple yet very effective method in practice (if given lots of training data)
  - *Provably* has an error-rate that is no worse than twice of the "Bayes optimal" classifier which assumes knowledge of the true data distribution for each class
- Also called a memory-based or instance-based or non-parametric method
- No "model" is learned here. Prediction step uses all the training data
- Requires lots of storage (need to keep all the training data at test time)
- Predction can be slow at test time
  - For each test point, need to compute its distance from all the training points
  - Clever data-structures or data-summarization techniques can provide speed-ups
- Need to be careful in choosing the distance function to compute distances (especially when the data dimension D is very large)
- The 1-NN can suffer if data contains outliers (we will soon see a geometric illustration), or if amount of training data is small. Using more neighbors ($K > 1$) is usually more robust

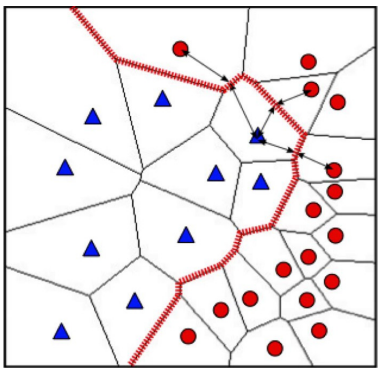
Geometry of 1-NN

- 1-NN splits the planes into a finite set of regions of the input space



The Decision Boundary of 1-NN (for binary classification)

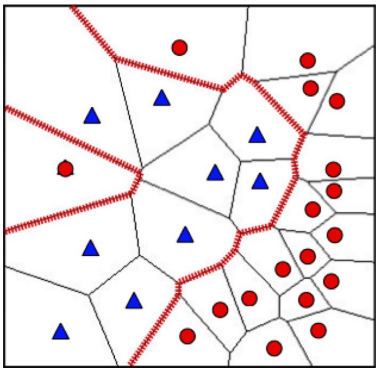
- The decision boundary is composed of hyperplanes that form perpendicular bisectors of pairs of points from different classes



Pic credit: Victor Lavrenko

Effect of Outliers on 1-NN

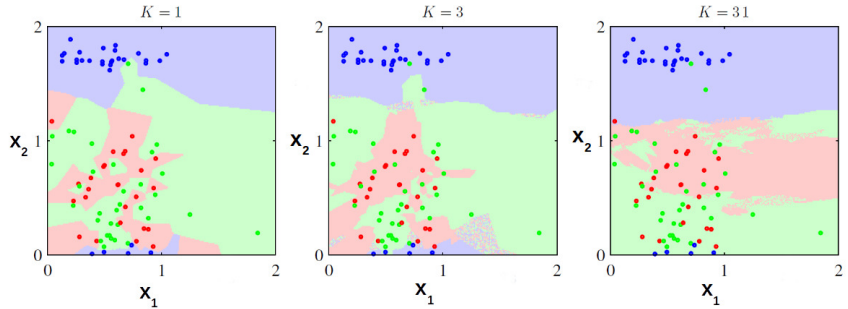
- How the decision boundary can drastically change when the data contains some outliers



Pic credit: Victor Lavrenko

Effect of Varying K

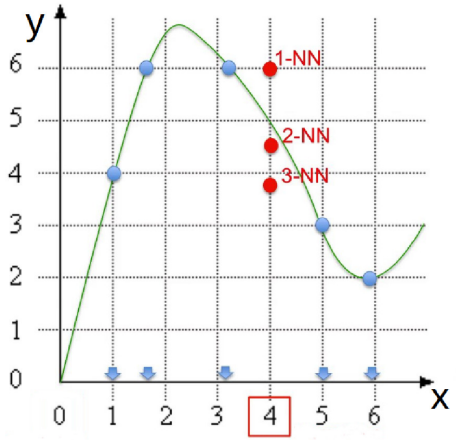
- Larger K leads to smoother decision boundaries



Too small K (e.g., $K = 1$) can lead to overfitting, too large K can lead to underfitting

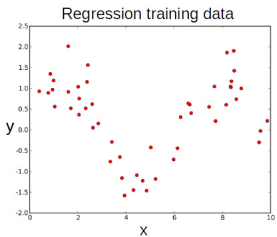
Pic credit: Chris Bishop (PRML)

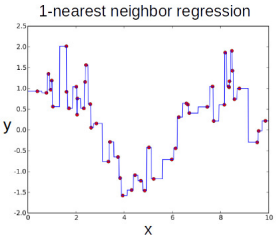
K-NN Behavior for Regression

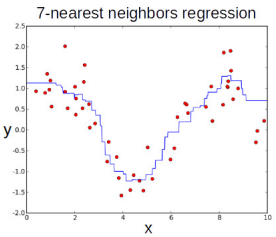


Pic credit: Victor Lavrenko

K-NN Behavior for Regression







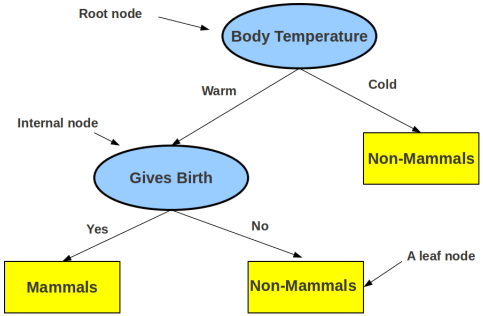
Pic credit: Alex Smola and Vishy Vishwanathan

Summary

- Looked at two distance-based methods for classification/regression
  - A "Distance from Means" Method
  - Nearest Neighbors Method
- Both are essentially "local" methods (look at local neighborhood of the test point)
- Both are simple to understand and only require knowledge of basic geometry
- Have connections to other more advanced methods (as we will see)
- Need to be careful when computing the distances (learned Mahalanobis distance metrics, or "learned features" + Euclidean distance can often do wonders!)

Decision Tree

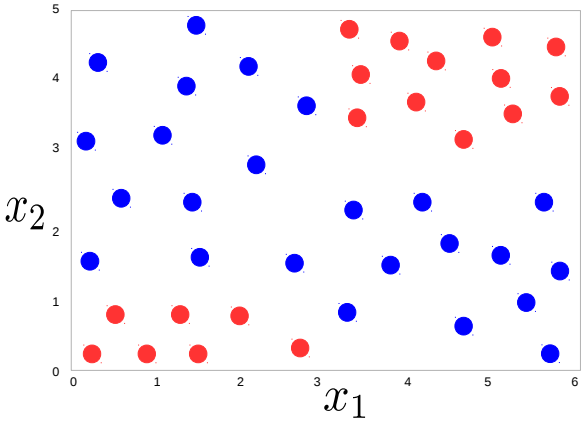
- Defines a tree-structured hierarchy of rules
- Consists of a root node, internal nodes, and leaf nodes



- Root and internal nodes contain the rules
- Leaf nodes define the predictions
- Decision Tree (DT) learning is about learning such a tree from labeled training data

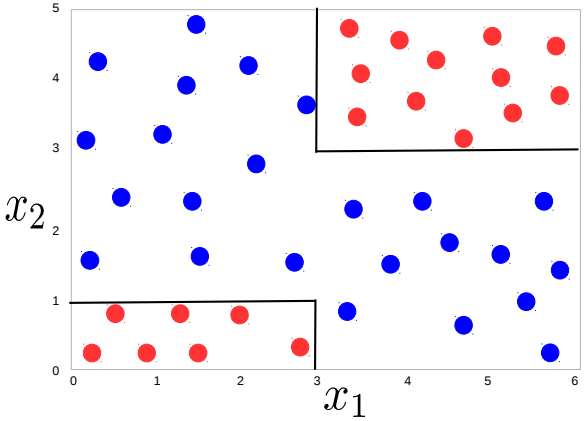
A Classification Problem

Consider binary classification. Assume training data with each input having 2 features (x_1, x_2)



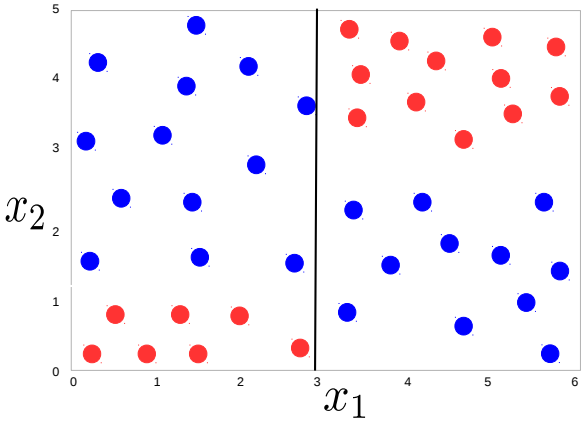
A Classification Problem

The "expected" decision boundary given this training data. Let's learn this!



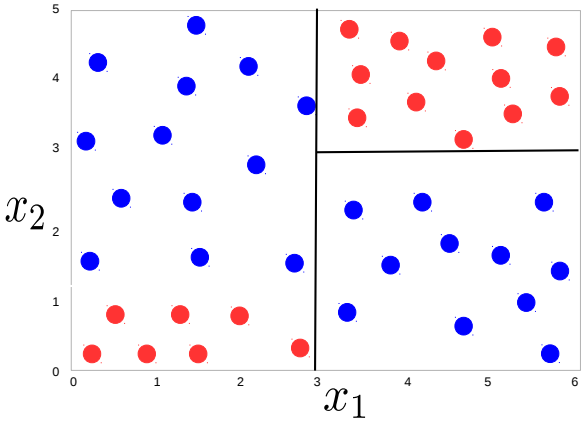
Learning by Asking Questions!

Is x_1 (feature 1) greater than 3 ?



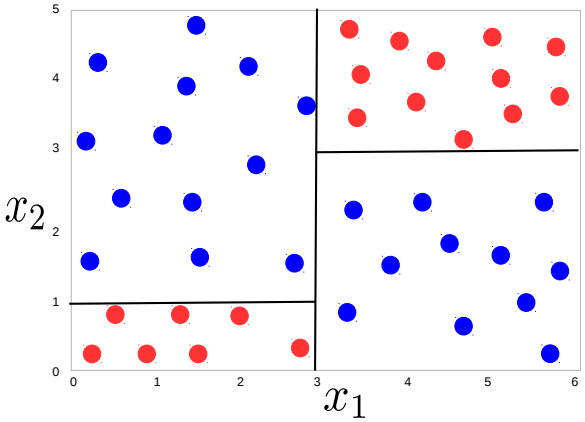
Learning by Asking Questions!

Given $x_1 > 3$, is feature 2 (x_2) greater than 3?



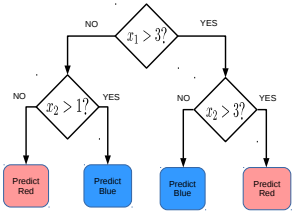
Learning by Asking Questions!

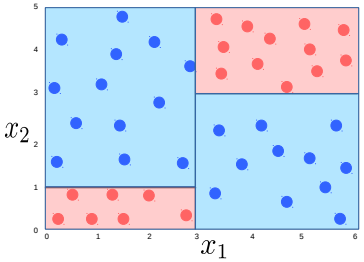
Given $x_1 < 3$, is feature 2 (x_2) greater than 1?



What We Learned

- A Decision Tree (DT) consisting of a set of rules **learned** from training data



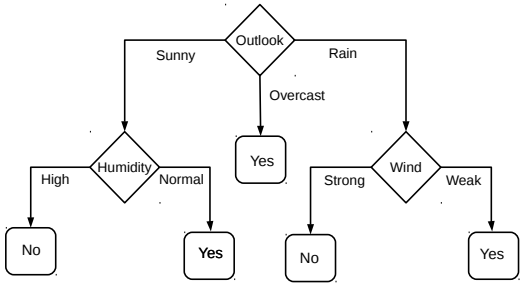


- These rules perform a recursive partitioning of the training data into "homogeneous" regions
  - Homogeneous means that the outputs are same/similar for all inputs in that region
- Given a new test input, we can use the DT to predict its label
- A key benefit of DT: Prediction at test time is very fast (just testing a few conditions)

Decision Tree for Classification: Another Example

- Deciding whether to play or not to play Tennis on a Saturday
  - Each input (a Saturday) has 4 categorical features: Outlook, Temp., Humidity, Wind
  - A binary classification problem (play vs no-play)
  - Left: Training data, Right: A decision tree constructed using this data

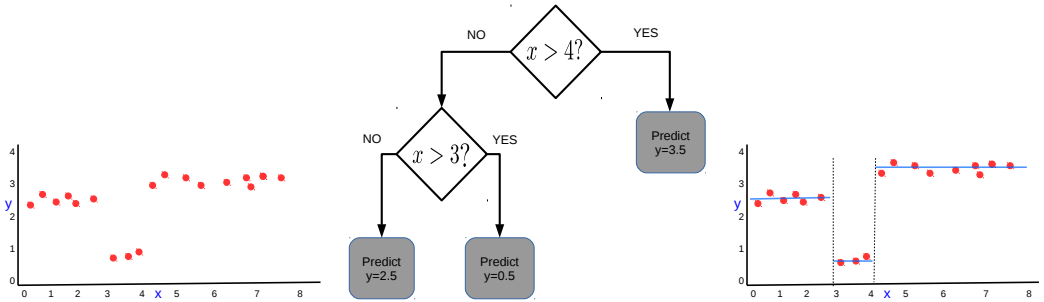
day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



Example credit: Tom Mitchell

A Decision Tree for Regression

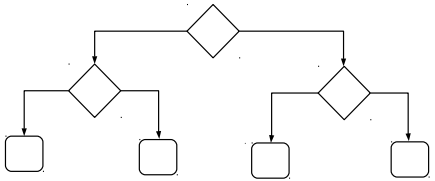
Decision Trees can also be used for regression problems



Here too, the DT partitions the training data into homogeneous regions (inputs with similar outputs)

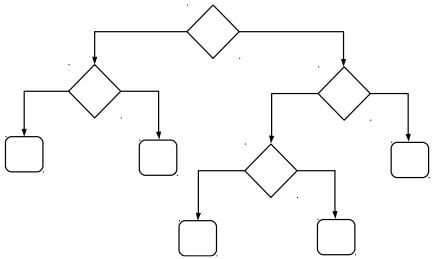
Some Considerations: Shape/Size of DT

- What should be the size/shape of the DT?
  - Number of internal and leaf nodes
  - Branching factor of internal nodes
  - Depth of the tree



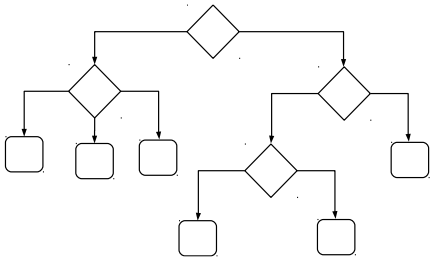
Some Considerations: Shape/Size of DT

- What should be the size/shape of the DT?
  - Number of internal and leaf nodes
  - Branching factor of internal nodes
  - Depth of the tree



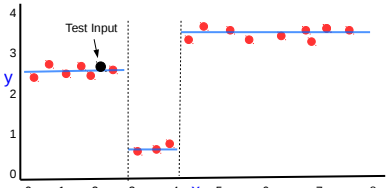
Some Considerations: Shape/Size of DT

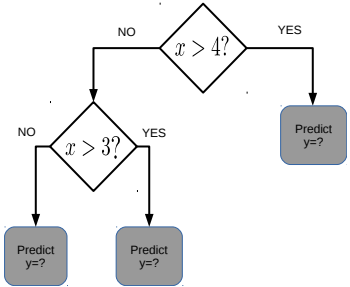
- What should be the size/shape of the DT?
  - Number of internal and leaf nodes
  - Branching factor of internal nodes
  - Depth of the tree



Some Considerations: Leaf Nodes

- What to do at each leaf node (the goal: make predictions)? Some options:
  - Make a constant prediction (majority/average) for every test input reaching that leaf node?
  - Use a nearest neighbors based prediction using all training inputs on that leaf node?

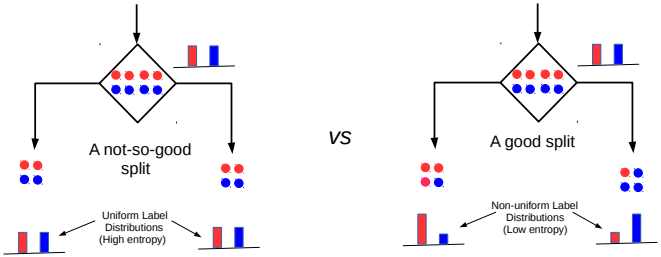




  - (Less common) Predict using an ML model learned using training inputs that belong to that leaf node?
- Constant prediction is the fastest at test time (and gives a piece-wise constant prediction rule)

Some Considerations: Internal Nodes

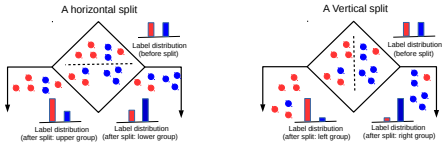
- How to split at each internal node (the goal: split the training data received at that node)?
- No matter how we split, the goal should be to have splits that result in groups as "pure" as possible



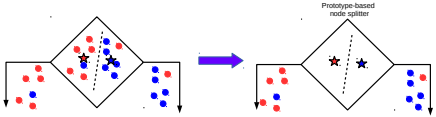
- For classification problems, entropy of the label distribution is a measure of purity
  - Low entropy $\Rightarrow$ high purity (less uniform label distribution)
  - Splits that give the largest reduction (before split vs after split) in entropy are preferred (this reduction is also known as "information gain")
- For regression, entropy doesn't make sense (outputs are real-valued). Typically variance is used.

Some Considerations: Internal Nodes (Contd.)

- Note that splitting at internal node itself is like a classification problem
  - Data received at the internal node has to be routed along its outgoing branches
- Some common techniques for splitting an internal node
  - Splitting by testing a single feature (simplest/fastest; used in ID3, C4.5 DT algos). For example:



  - Splitting using a classifier learned using data on that node. For example, prototype based classifier

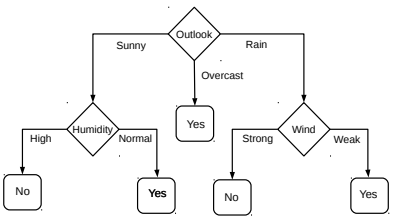


- The same splitting rule will be applied to route a test input that reaches this internal node

Decision Tree Construction

- As an illustration, let's look at one way of constructing a decision tree for some given data
- We will use the entropy/information-gain based splitting criterion for this illustration

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



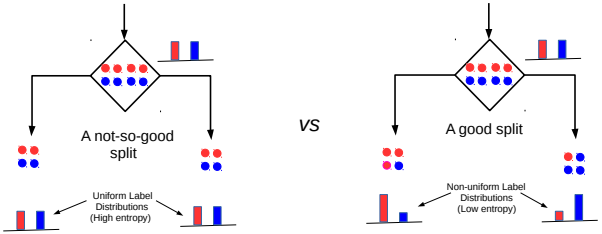
- Question: Why does it make more sense to test the feature "outlook" first?
- Answer: Of all the 4 features, it's most informative (highest information gain as the root node)
- **Analogy:** Playing the game 20 Questions (the most useful questions first)

Entropy and Information Gain

- Consider a set S of inputs with a total C classes, p_c = fraction of inputs from class/label c
- Entropy of the set S: $H(S) = -\sum_{c \in C} p_c \log_2 p_c$
- The difference in the entropy before and after the split is called information gain (IG)
- For one group S being split into two smaller groups S_1 and S_2, we can calculate the IG as follows

$$IG = H(S) - \frac{|S_1|}{|S|} H(S_1) - \frac{|S_2|}{|S|} H(S_2)$$

- For DT construction, entropy/IG gives us a criterion to select the best split for an internal node



DT Construction using IG Criterion

- Let's look at IG based DT construction for the Tennis example
- Let's begin with the root node of the DT and compute IG of each feature
- Consider feature "wind" $\in$ {weak,strong} and its IG at root

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no

- Root node: $S = [9+, 5-]$ (all training data: 9 play, 5 no-play)
- Entropy: $H(S) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.94$
- $S_{weak} = [6+, 2-] \Longrightarrow H(S_{weak}) = 0.811$, $S_{strong} = [3+, 3-] \Longrightarrow H(S_{strong}) = 1$

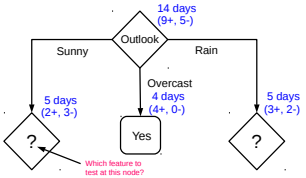
$$\begin{aligned} IG(S, \text{wind}) &= H(S) - \frac{|S_{weak}|}{|S|} H(S_{weak}) - \frac{|S_{strong}|}{|S|} H(S_{strong}) \\ &= 0.94 - 8/14 * 0.811 - 6/14 * 1 \\ &= 0.048 \end{aligned}$$

Likewise, $IG(S, \text{outlook}) = 0.246$, $IG(S, \text{humidity}) = 0.151$, $IG(S, \text{temperature}) = 0.029 \Rightarrow$ outlook chosen

DT Construction using IG Criterion: Growing the tree

- Having decided which feature to test at the root, let's grow the tree
- How to decide which feature to test at the next level (level 2) ?
- **Rule:** Iterate - for each child node, select the feature with the highest IG

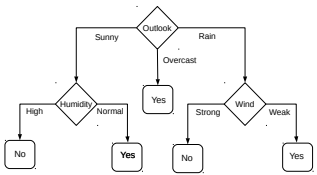
day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



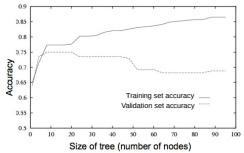
- Proceeding as before, for level 2, left node, we can verify that
  - $IG(S, \text{temperature}) = 0.570$, $IG(S, \text{humidity}) = 0.970$, $IG(S, \text{wind}) = 0.019$ (thus humidity chosen)
- No need to expand the middle node (already "pure" - all yes)
- Can also verify that wind has the largest IG for the right node
- Note: If a feature has already been tested along a path earlier, we don't consider it again

When to Stop Growing?

day	outlook	temperature	humidity	wind	play
1	sunny	hot	high	weak	no
2	sunny	hot	high	strong	no
3	overcast	hot	high	weak	yes
4	rain	mild	high	weak	yes
5	rain	cool	normal	weak	yes
6	rain	cool	normal	strong	no
7	overcast	cool	normal	strong	yes
8	sunny	mild	high	weak	no
9	sunny	cool	normal	weak	yes
10	rain	mild	normal	weak	yes
11	sunny	mild	normal	strong	yes
12	overcast	mild	high	strong	yes
13	overcast	hot	normal	weak	yes
14	rain	mild	high	strong	no



- Stop expanding a node further when
  - It consist of examples all having the same label (the node becomes "pure")
  - We run out of features to test along the path to that node
  - The DT starts to overfit (can be checked by monitoring the validation set accuracy)



Avoiding Overfitting: Decision Tree Pruning

- Desired: a DT that is not too big in size, yet fits the training data reasonably
- Note: An example of a very simple DT is "decision-stump"
  - A decision-stump only tests the value of a single feature
  - Not very powerful in itself but often used in large ensembles of decision stumps
- Mainly two approaches to prune a complex DT
  - Prune while building the tree (**stopping early**)
  - Prune after building the tree (**post-pruning**)
- Criteria for judging which nodes could potentially be pruned
  - Use a validation set (separate from the training set)
    - Prune each possible node that doesn't hurt the accuracy on the validation set
    - Greedily remove the node that improves the validation accuracy the most
    - Stop when the validation set accuracy starts worsening
  - Use model selection methods, such as Minimum Description Length (MDL); more on this later

Decision Tree: Some Comments

- Other alternatives to entropy for judging feature informativeness in DT classification?
  - Gini-index $\sum_{c=1}^{C} p_c(1 - p_c)$ is another popular choice
- For DT regression (Regression Trees[1]), can split based on the variance in the outputs, instead of using entropy (which doesn't make sense for real-valued inputs)
- Real-valued features (we already saw some examples) can be dealt with using thresholding
  - Need to be careful w.r.t. number of threshold points, how fine each range is, etc.
- More sophisticated decision rules at the internal nodes can also be used (anything that splits the inputs at an internal node into homogeneous groups; e.g., a machine learning classification algo)
- Need to take care handling training or test inputs that have some features missing

[1] Breiman, Leo; Friedman, J. H.; Olshen, R. A.; Stone, C. J. (1984). Classification and regression trees

Some Aspects about Decision Trees

Some key strengths:

- Simple and each to interpret
- Do not make any assumption about distribution of data
- Easily handle different types of features (real, categorical/nominal, etc.)
- Very fast at test time (just need to check the features, starting the root node and following the DT until you reach a leaf node)
- Multiple DTs can be combined via ensemble methods (e.g., Decision Forest)
  - Each DT can be constructed using a (random) small subset of features

Some key weaknesses:

- Learning the optimal DT is NP-Complete. The existing algorithms are heuristics (e.g., greedy selection of features)
- Can sometimes become very complex unless some pruning is applied