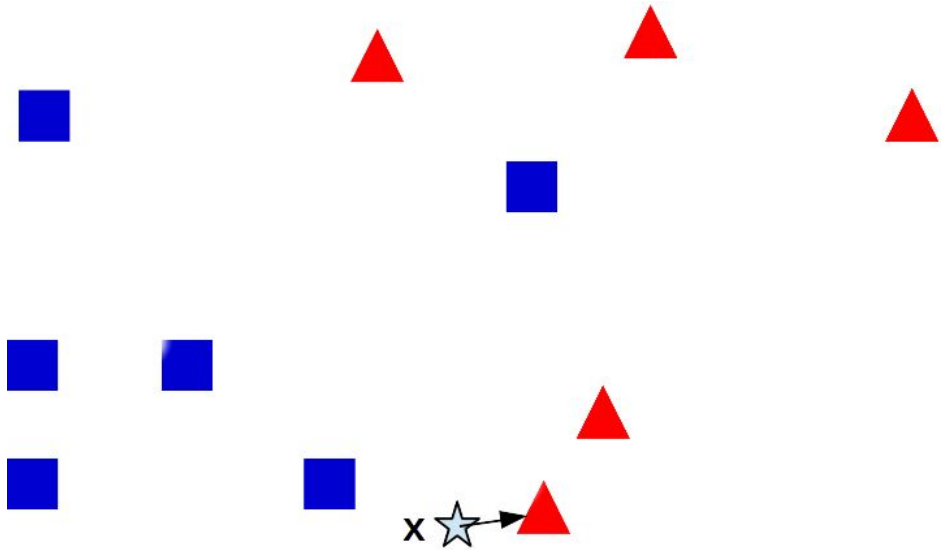
Supervised Learning

K-Nearest Neighbour, Linear, Logistic...

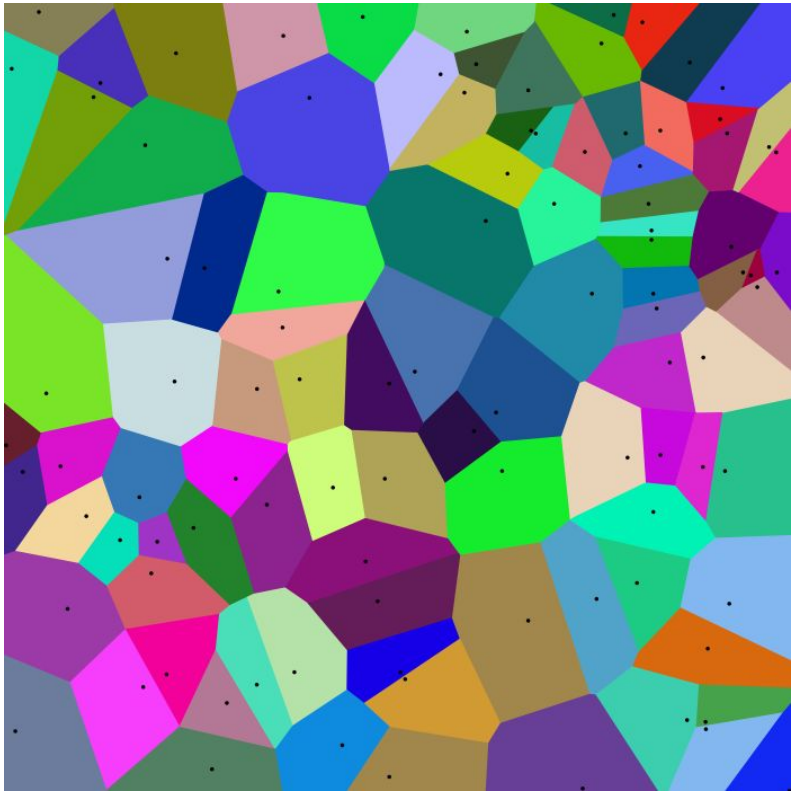
Learning By Computing Distances - kNN

- The label y for $x \in R^D$ will be label of its **nearest neighbour in training data.**
- Euclidean Distance can be used to find the nearest neighbour.
- NOTE : This method also applies to regression.



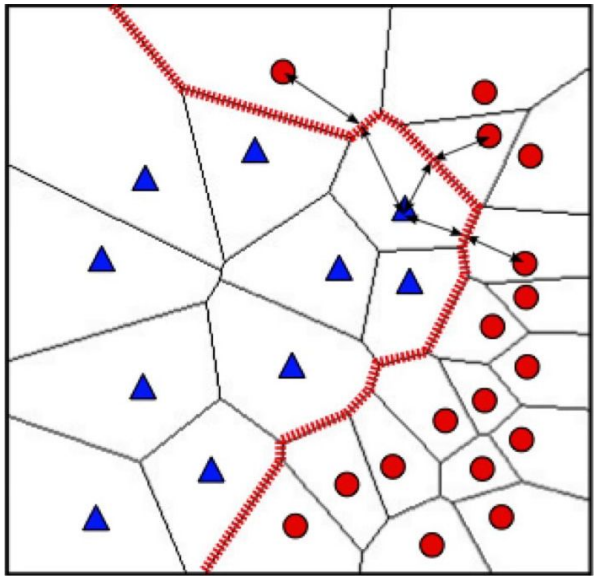
- A simple yet very effective method in practice (if given lots of training data).
- Also called a memory-based or instance-based or non-parametric method.
- No "model" is learned here. Prediction step uses all the training data.
- Requires lots of storage (need to keep all the training data at test time).
- Prediction 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



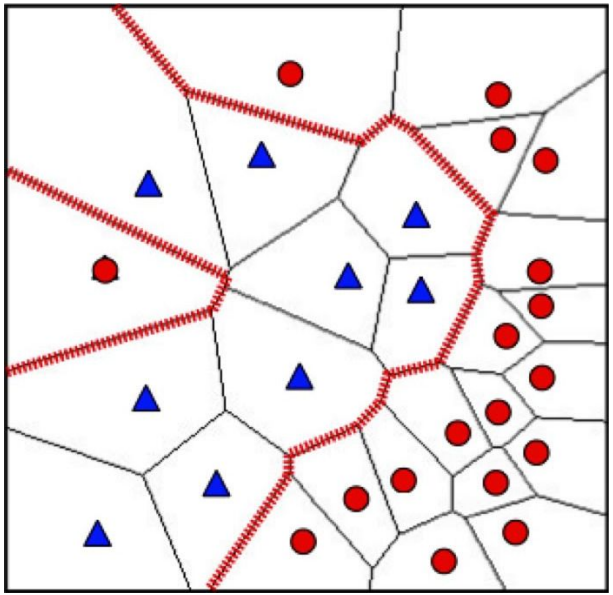
The Decision Boundary of 1-NN

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



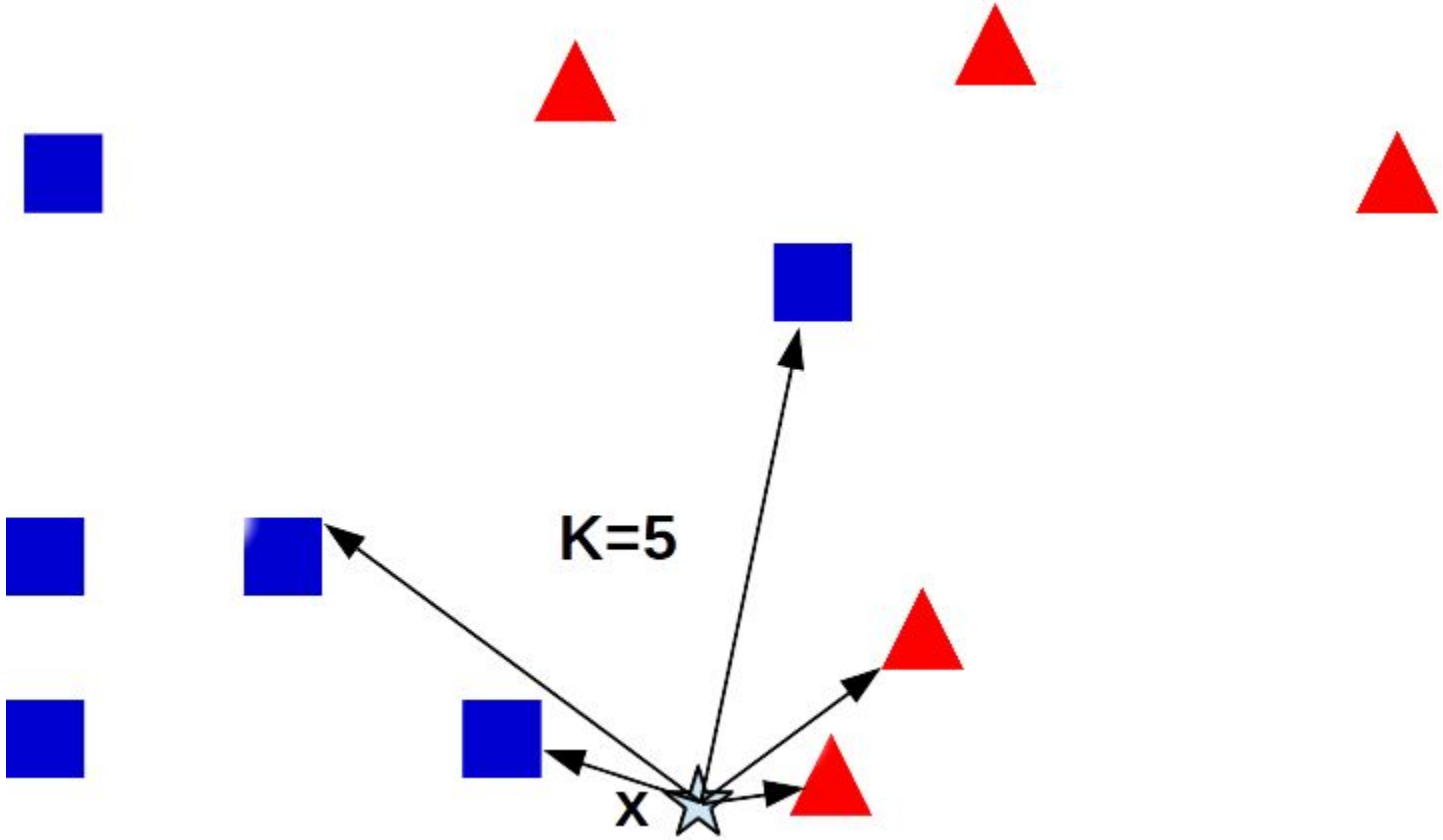
Effect of Outliers on 1-NN

An illustration of how the decision boundary can drastically change when the data contains some outliers



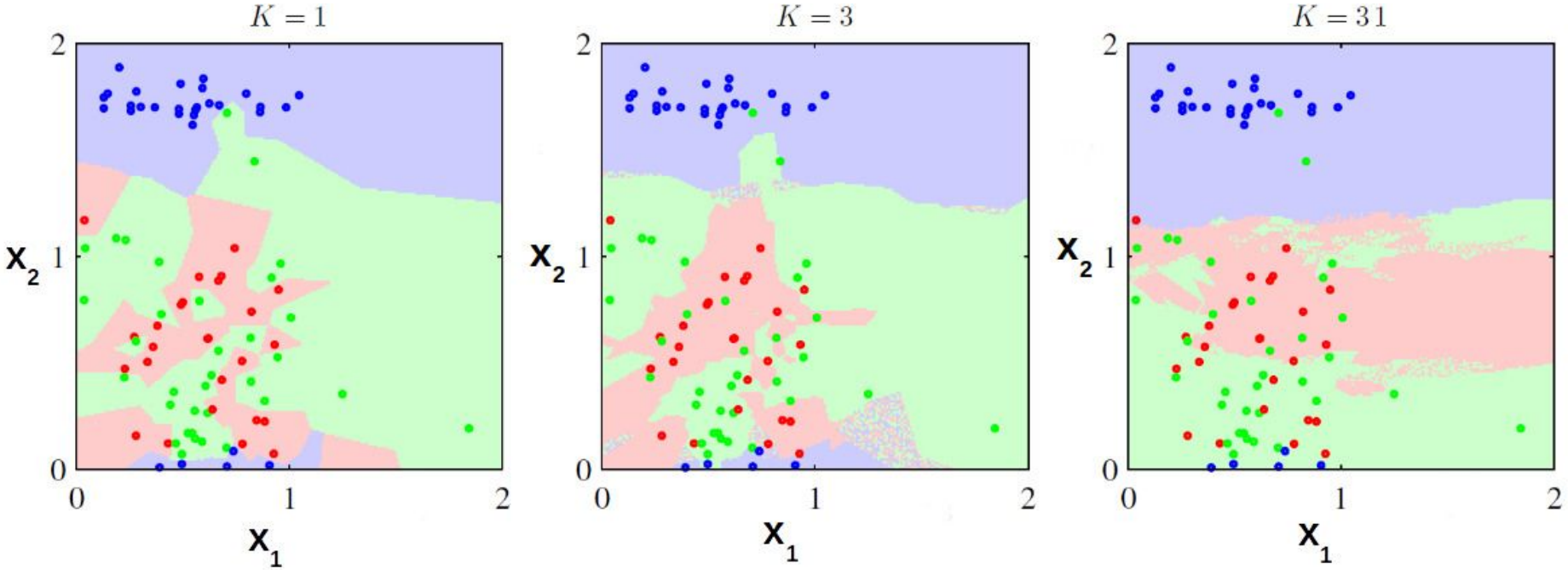
K -Nearest Neighbors (K -NN)

- Makes one-nearest-neighbor more robust by using more than one neighbor.
- The K -NN prediction rule: Take a majority vote (or average) of the labels of K > 1 neighbors in the training data.
- Works for both classification and regression
  - For classification, we usually take the majority labels from the K neighbors
  - For regression, we usually average the real-valued labels of the K neighbors
- The “right” value of K needs to be selected (e.g., via cross-validation)

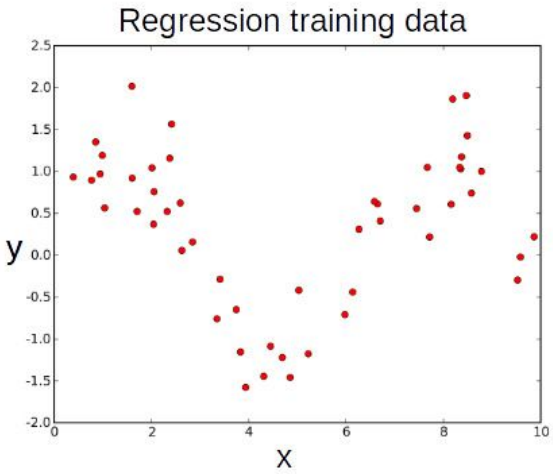

K=5
x

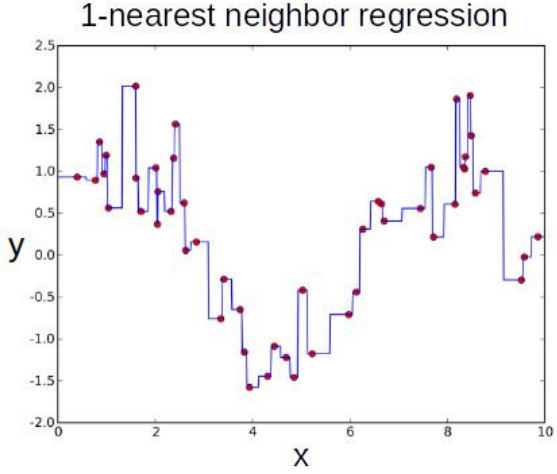
K -Nearest Neighbors: Decision Boundaries

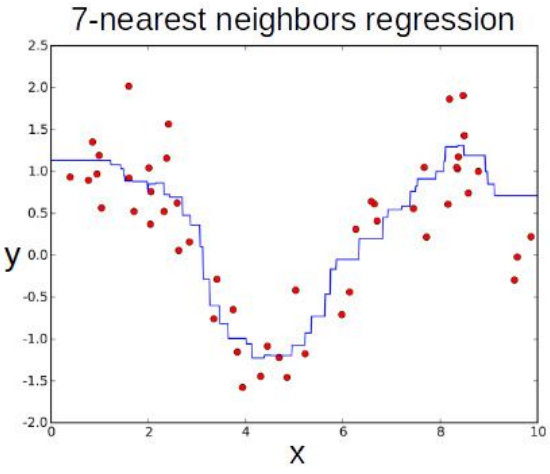
Larger K leads to smoother decision boundaries



K -NN Behavior for Regression



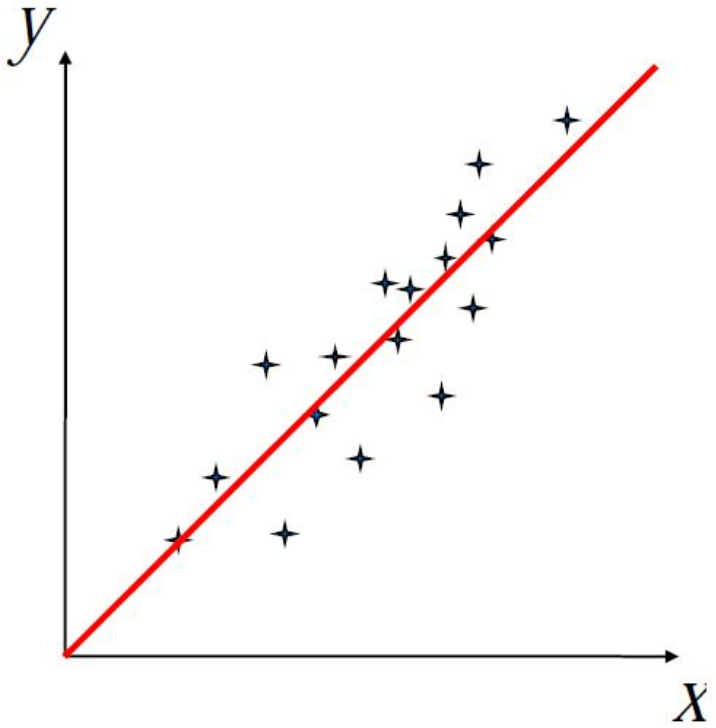




Linear Regression

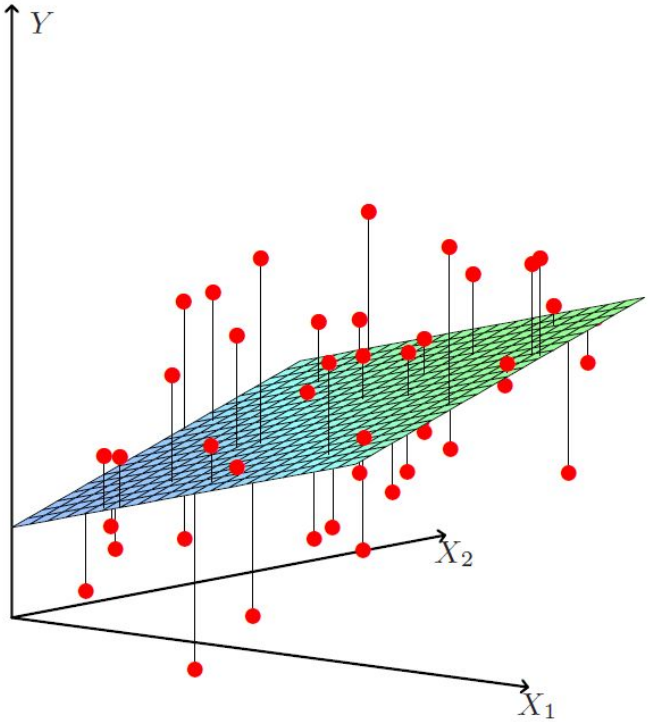
Fitting a Line to the Data

- Let's assume the relationship between x and y to have a linear model : **y = wx**
- Problem boils down to fitting a line to the data
- w is the model parameter (slope of the line here)
- Many w 's (i.e., many lines) can be fit to this data
- Which one is the best?



Fitting a (Hyper)Plane to the Data

- For 2-dim. inputs, we can fit a 2-dim. plane to the data.
- In higher dimensions, we can likewise fit a hyperplane $w > x = 0$
- Defined by a D-dim vector w normal to the plane
- Many planes are possible. Which one is the best?



Linear Regression

Given: Training data with N examples $\{(\mathbf{x}_n , y_n)\}_{n=1}^{N}$, $\mathbf{x}_n \in \mathbb{R}$, $y_n \in \mathbb{R}$

Assume the following linear model with model parameters $\mathbf{w} \in \mathbb{R}^D$

$$y_n \approx \boldsymbol{w}^\top \boldsymbol{x}_n \quad \Rightarrow \quad y_n \approx \sum_{d=1}^{D} w_d x_{nd}$$

- The response y_n is a linear combination of the features of the inputs $\mathbf{x}_n$
- $\mathbf{w} \in \mathbb{R}^D$ is also called the (regression) **weight vector**
  - Can think of w_d as weight/importance of d-th feature in the data

Linear Regression

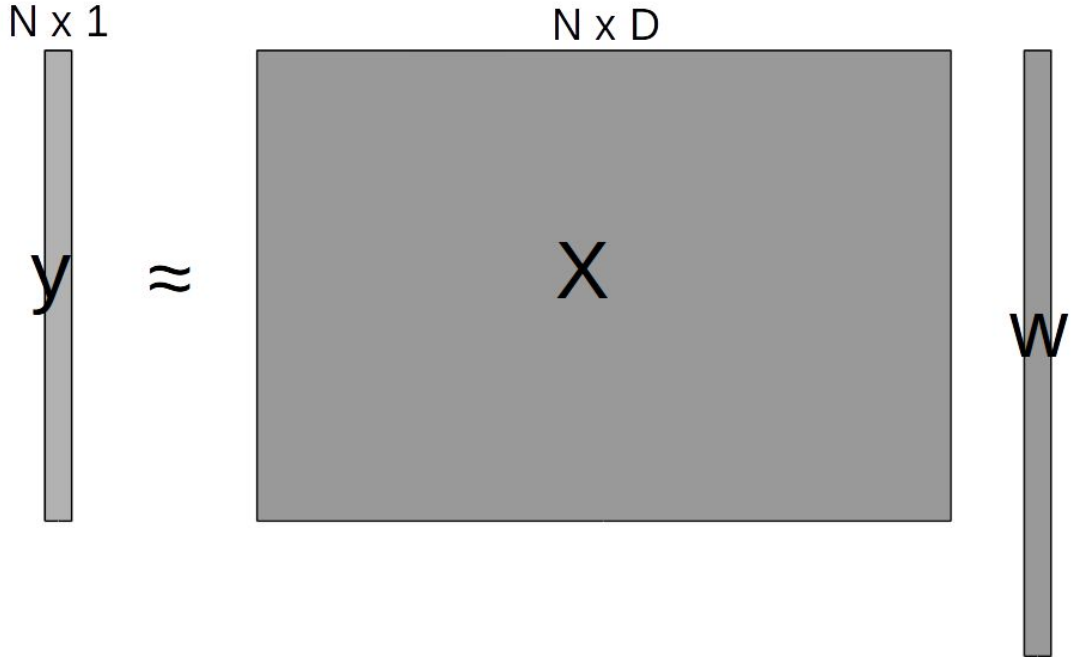
- A simple and interpretable linear model. Can also re-express it compactly for all the N examples

  $\mathbf{y} \approx \mathbf{X}\mathbf{w}$ (akin to a linear system of equations; w being the unknown)

- Notation used here:
  - $\mathbf{w} \in |R^D$ and each $x_n \in |R^D$ are D × 1 column vectors
  - $\mathbf{X} = [\mathbf{x_1}\ \mathbf{x_2} \ldots \mathbf{x_N}]^T$ is an N × D matrix of features
  - $\mathbf{y} = [y_1\ y_2 \ldots y_N]^T$ is an N × 1 column vector of responses

Linear Regression

Linear system of equations with w being the unknown..



Linear Regression with Squared Loss

- Our linear regression model: $y_n \approx \mathbf{w}^T \mathbf{x}_n$. The goal is to learn $\mathbf{w} \in |R^D$
- Let's use the squared loss to define our loss function.

$$\ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n) = (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

- Using the squared loss, the total (empirical) error on the training data

$$L_{emp}(\boldsymbol{w}) = \sum_{n=1}^{N} \ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n) = \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

- We'll estimate $\mathbf{w}$ by minimizing $L_{emp}(\mathbf{w})$ w.r.t. $\mathbf{w}$ (an optimization problem)

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

Least Squares Linear Regression

- Recall our objective function: $L_{emp} = \sum_{n=1}^{N} (y_n - \mathbf{w}^\top \mathbf{x}_n)^2$
- Taking derivative of $L_{emp}(\mathbf{w})$ w.r.t. $\mathbf{w}$ and setting to zero

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

- Simplifying further, we get a nice, closed form solution for $\mathbf{w}$

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

Least Squares Linear Regression

- Analytic, closed form solution, but has some issues
  - We didn't impose any regularization on w (thus prone to overfitting)
  - Have to invert a D × D matrix; prohibitive especially when D (and N) is large
  - The matrix $\mathbf{X}^T \mathbf{X}$ may not even be invertible (e.g., when D > N). Unique solution not guaranteed

Ridge Regression: Regularized Least Squares

- Least Squares objective: $L_{emp} = \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$
- No constraints/regularization on w . Components [w_1 , w_2 , . . . , w_D] of **w** may become arbitrarily large. Why is this a bad thing to have?
- Let's add squared l_2 norm of **w** as a regularizer: R(f) = R(w) = $||\mathbf{w}||^2$
- This results in the so-called "Ridge Regression" model.

$$L_{reg} = \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda ||\boldsymbol{w}||^2$$

- Note that $||\boldsymbol{w}||^2 = \boldsymbol{w}^\top \boldsymbol{w} = \sum_{d=1}^{D} w_d^2$

Ridge Regression: Regularized Least Squares

- Minimizing L_{reg} will prevent components of **w** from becoming very large. Why is this nice?
- Taking derivative of L reg w.r.t. w and setting to zero gives (verify yourself)

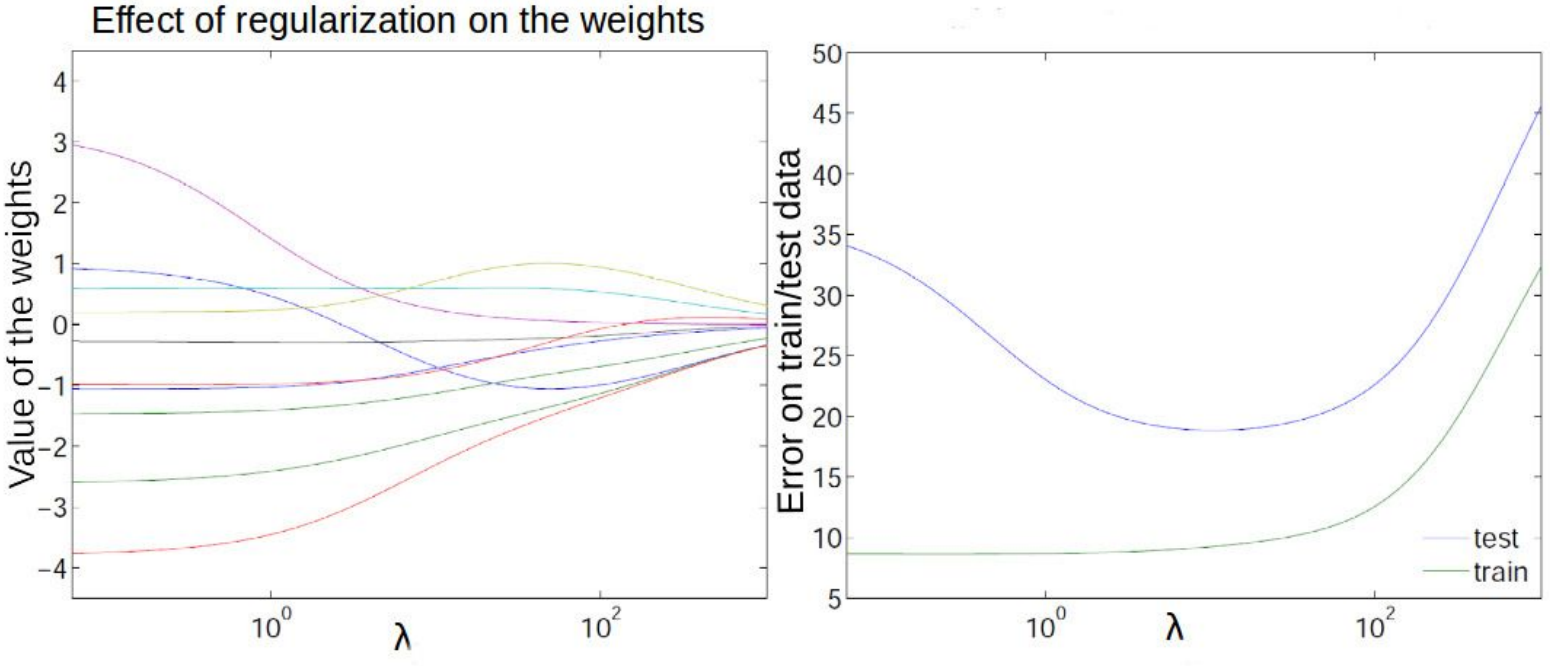
$$\boldsymbol{w} = (\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda \mathbf{I}_D)^{-1} \sum_{n=1}^{N} y_n \boldsymbol{x}_n = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

Intuitively, Why Small Weights are Good?

- Small weights ensure that the function $y = f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ is **smooth** (i.e., we expect similar x's to have similar y 's). Below is an informal justification:
- Consider two points $\mathbf{x_n} \in R^D$ and $\mathbf{x_m} \in R^D$ 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 (x), y n and y m should also be close
- However, as per the model y = f (x) = w > x, y n and y m will differ by ϵw_d
- Unless we constrain w_d to have a small value, the difference w 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

Ridge Regression: Effect of Regularization

- Consider ridge regression on some data with 10 features (thus the weight vector w has 10 components)



Solution via Gradient-based Methods

- Both least squares and ridge regression require matrix inversion.

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

$$\text{Ridge} \quad \boldsymbol{w} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \boldsymbol{y}$$

- This can be computationally very expensive when D is very large.
- We can instead solve for w more efficiently using generic/specialized optimization methods on the respective loss functions (L_{emp} or L_{reg}).

Solution via Gradient-based Methods

- A simple scheme can be the following iterative gradient-descent procedure
  - Start with an initial value of $\mathbf{w} = \mathbf{w}^{(0)}$
  - Update $\mathbf{w}$ by moving along the gradient of the loss function L (L_{emp} or L_{reg}), where η is the learning rate
  - Repeat until converge

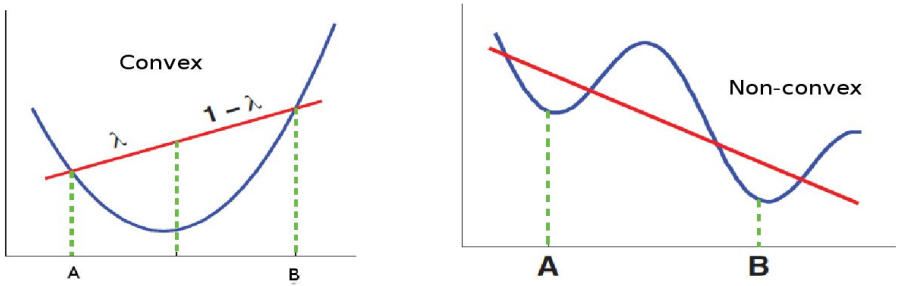
$$\boldsymbol{w}^{(t)} = \boldsymbol{w}^{(t-1)} - \eta \frac{\partial L}{\partial \boldsymbol{w}}\bigg|_{\boldsymbol{w}=\boldsymbol{w}^{(t-1)}}$$

- For unreg. least squares, the gradient is

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

Gradient-based Methods: Some Notes

- Guaranteed to converge to a local minima
- Converge to global minima if the function is **convex**



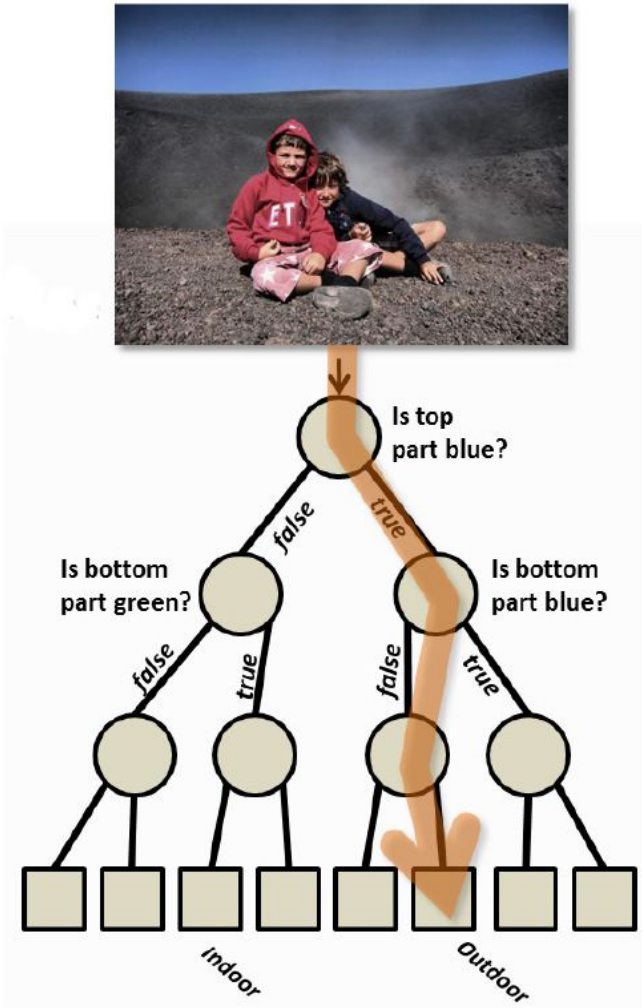
- Note: The squared loss function in linear regression is convex
  - With l_2 regularizer, it becomes strictly convex (single global minima).
- Learning rate is important (should not be too large or too small)
- Can also use stochastic/online gradient descent for more speed-ups. Require computing the gradients using only one or a small number of examples

DECISION TREES

A Classification Problem

Indoor or Outdoor ?

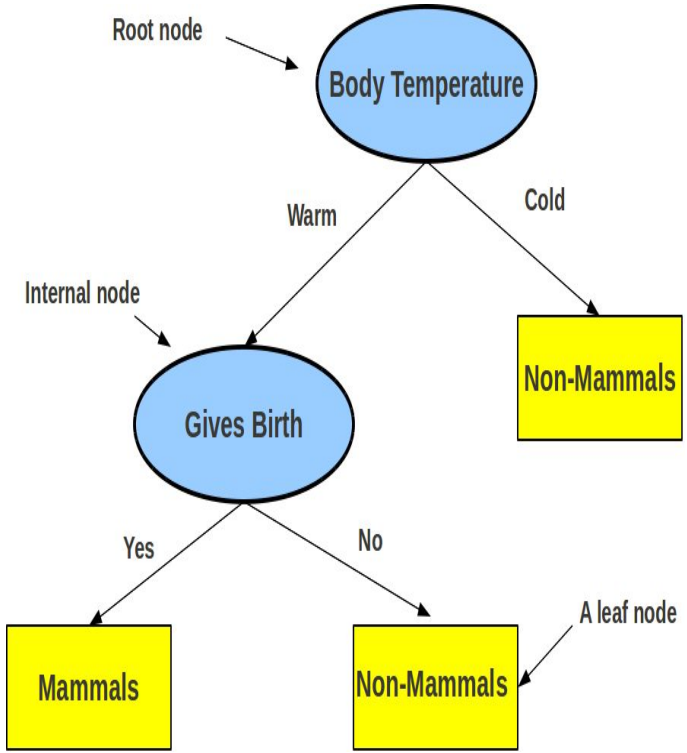
Predicting by Asking Questions



How can we learn this tree using labeled training data?

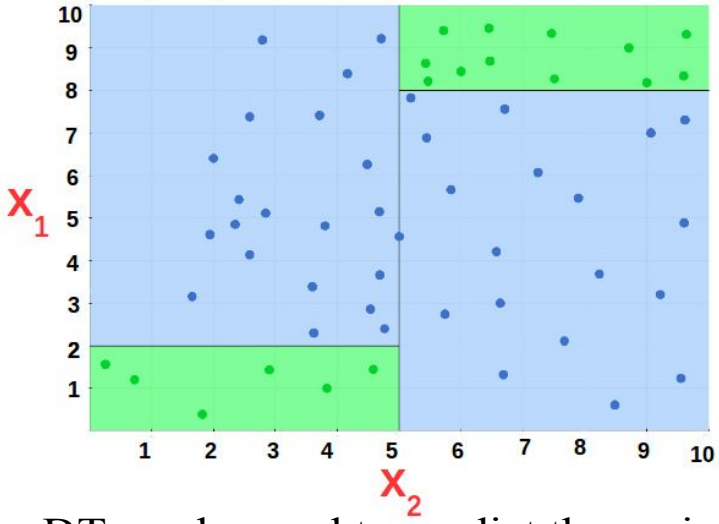
Decision Tree

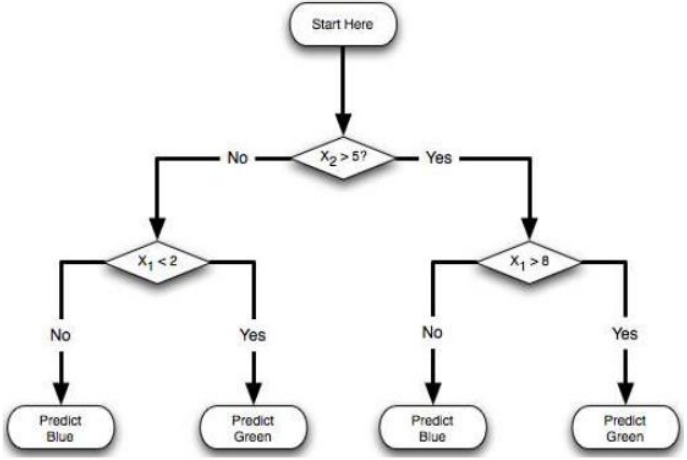
- Defined by a hierarchy of rules (in form of a tree).
- Rules form the internal nodes of the tree (topmost internal node = root).
- Each internal node tests the value of some feature and "splits" data across the outgoing branches.
- Note: The tree need not be a binary tree
- (Labeled) Training data is used to construct the Decision Tree (DT)
- The DT can then be used to predict label **y** of a test example **x**



Decision Tree: An Example

- Identifying the region - blue or green - a point lies in (binary classification).
  - Each point has 2 features: its coordinates $\{x_1, x_2\}$ on the 2D plane
  - Left: Training data, Right: A DT constructed using this data



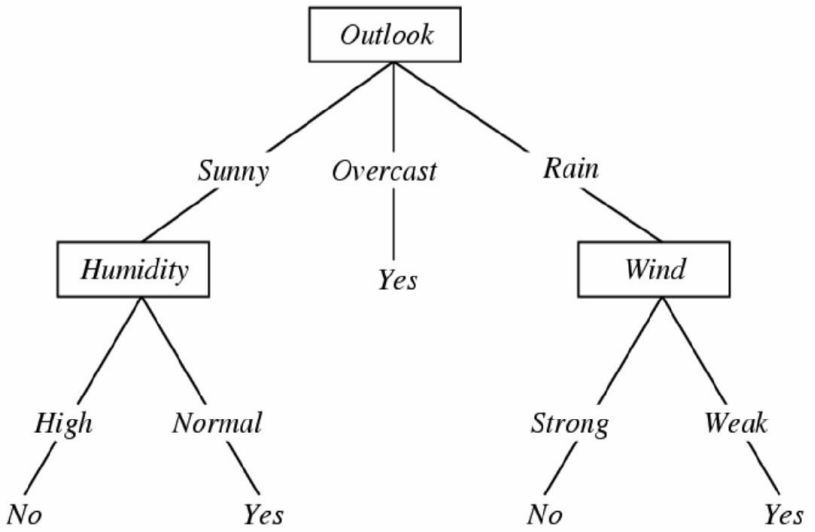


- The DT can be used to predict the region (blue/green) of a new test point
  - By testing the features of the test point
  - In the order defined by the DT (first x_2 and then x_1)

Decision Tree: Another Example

- Deciding whether to play or not to play Tennis on a Saturday
  - A binary classification problem (play vs no-play)
  - Each input (a Saturday) has 4 features: Outlook, Temp., Humidity, Wind
  - 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

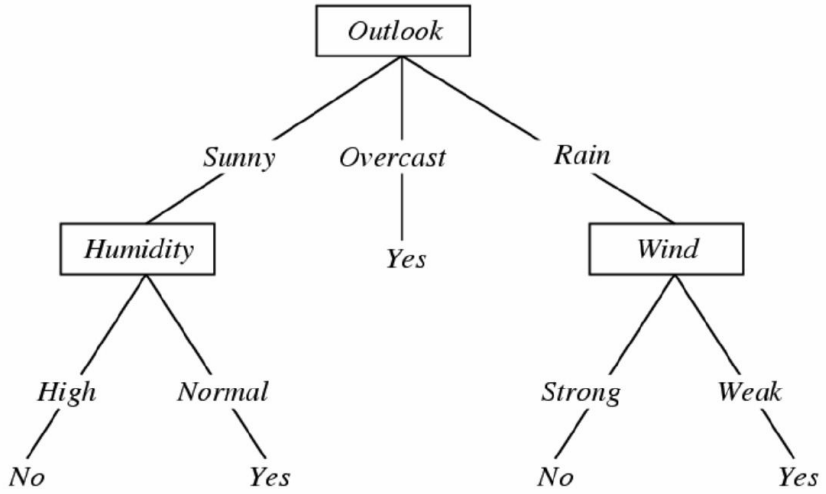


The DT can be used to predict play vs no-play for a new Saturday

Decision Tree Construction

- Now let's look at the playing Tennis example

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
- We will see shortly how to quantity the informativeness

Entropy

- Entropy is a measure of randomness/uncertainty of a set
- Assume our data is a set S of examples with C many classes
- p_c is the probability that a random element of S belongs to class c
- .. basically, the fraction of elements of S belonging to class c
- Probability vector $p = [p_1, p_2, \ldots, p_C]$ is the class distribution of the set S
- Entropy of the set S

$$H(S) = -\sum_{c \in C} p_c \log_2 p_c$$

- If a set S of examples (or any subset of it) has..
  - Some dominant classes ⟹ small entropy of the class distribution
  - Equiprobable classes ⟹ high entropy of the class distribution
- We can assess informativeness of each feature by looking at how much it reduces the entropy of the class distribution

Information Gain

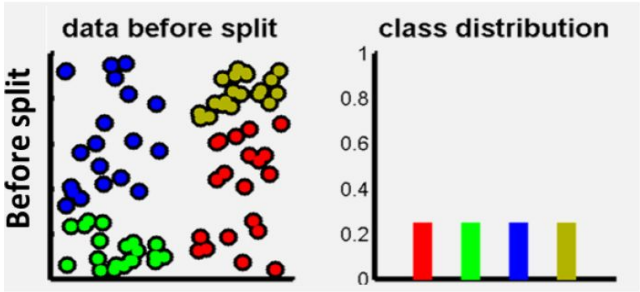
- Let’s assume each element of S has a set of features
- Information Gain (IG) on knowing the value of some feature ’F ’

$$IG(S, F) = H(S) - \sum_{f \in F} \frac{|S_f|}{|S|} H(S_f)$$

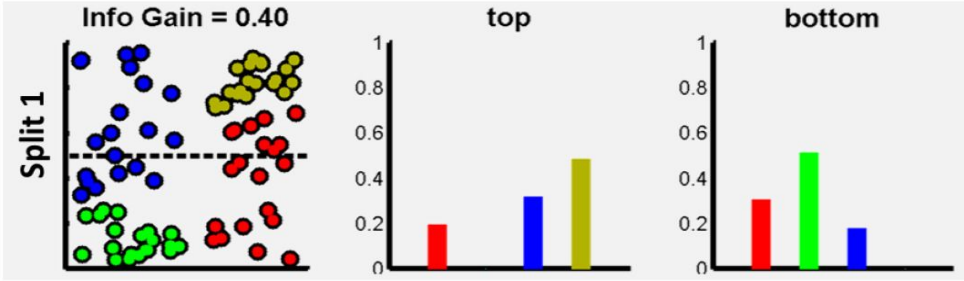
- S_f denotes the subset of elements of S for which feature F has value f
- IG (S, F) = entropy of S minus the weighted sum of entropy of its children
- IG (S, F): I**ncrease in our certainty** about S once we know the value of F

Entropy and Information Gain: Pictorially

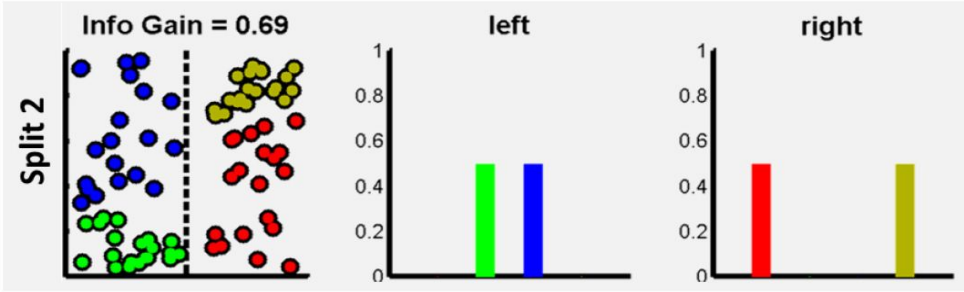
- Assume we have a 4-class problem. Each point has 2 features
- Which feature should we test (i.e., split on) first?



(a)



(b)



(c)

Computing Information Gain

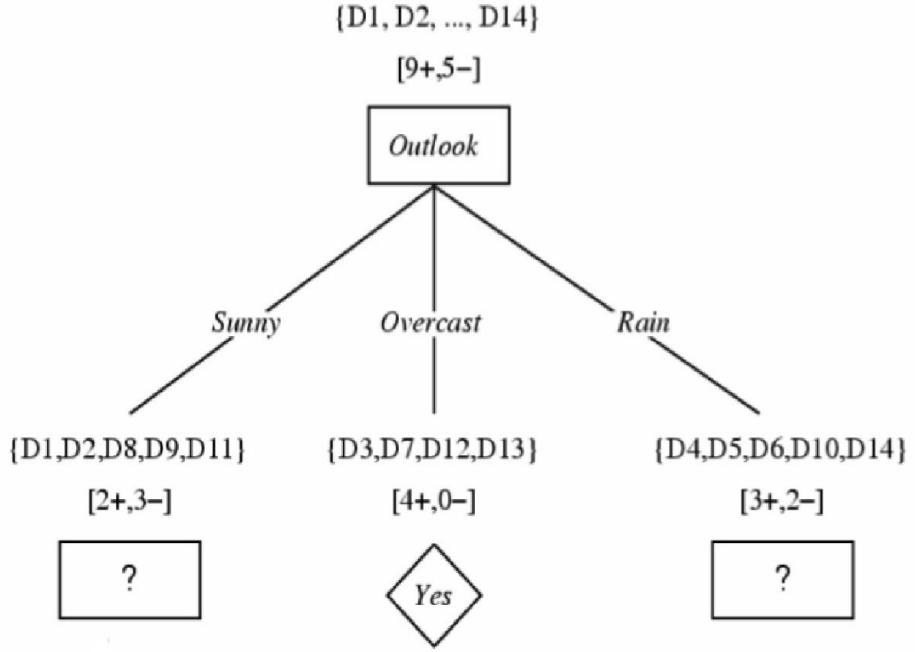
- Coming back to playing tennis..
- Let's begin with the root node of the DT and compute IG of each feature
- Consider feature "wind" $\in$ {weak,strong} and its IG w.r.t. the root node
- Root node: S = [9+, 5−] (all training data: 9 play, 5 no-play)
- Entropy: H(S) = −(9/14) log 2 (9/14) − (5/14) log 2 (5/14) = 0.94
- S weak = [6+, 2−] $\Longrightarrow$ H(S weak) = 0.811
- S strong = [3+, 3−] $\Longrightarrow$ H(S strong) = 1

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

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

Choosing the most informative feature

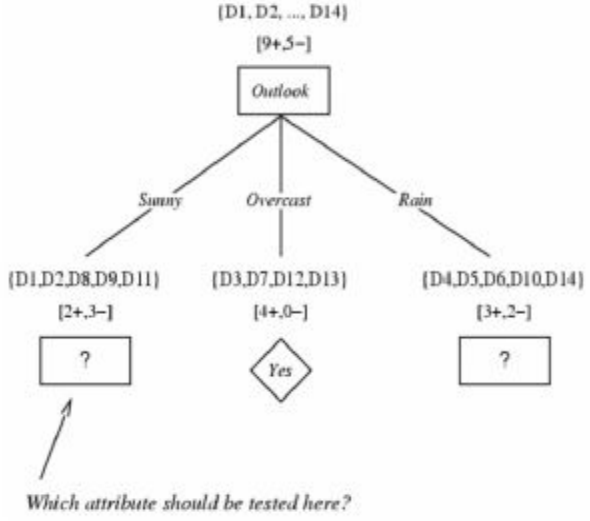
- At the root node, the information gains are:
  - IG (S, wind) = 0.048 (we already saw)
  - IG (S, outlook) = 0.246
  - IG (S, humidity) = 0.151
  - IG (S, temperature) = 0.029
- "outlook" has the maximum IG $\Longrightarrow$ chosen as the root node



Growing The Tree

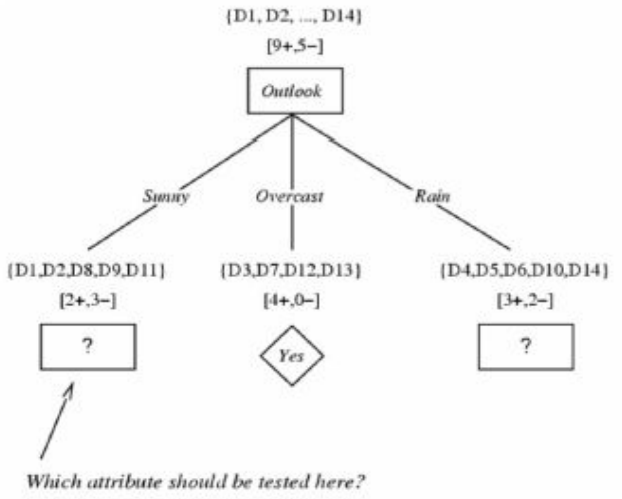
- How to decide which feature to test next ?
- **Rule:** Iterate - for each child node, select the feature with the highest IG

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



- For level-2, left node: S = [2+; 3-] (days 1,2,8,9,11)
- Compute the Information Gain for each feature (except outlook)
- The feature with the highest Information Gain should be chosen for this node

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



For this node ($S = [2+, 3-]$), the IG for the feature **temperature**:

$$IG(S, \text{temperature}) = H(S) - \sum_{v \in \{hot, mild, cool\}} \frac{|S_v|}{|S|} H(S_v)$$

$S = [2+, 3-] \Longrightarrow H(S) = -(2/5) * \log_2(2/5) - (3/5) * \log_2(3/5) = 0.971$

$S_{hot} = [0+, 2-] \Longrightarrow H(S_{hot}) = -0 * \log_2(0) - (2/2) * \log_2(2/2) = 0$

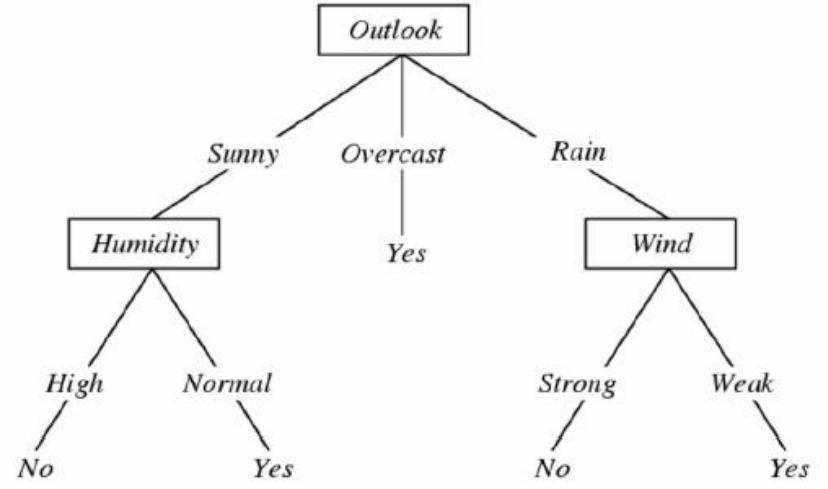
$S_{mild} = [1+, 1-] \Longrightarrow H(S_{mild}) = -(1/2) * \log_2(1/2) - (1/2) * \log_2(1/2) = 1$

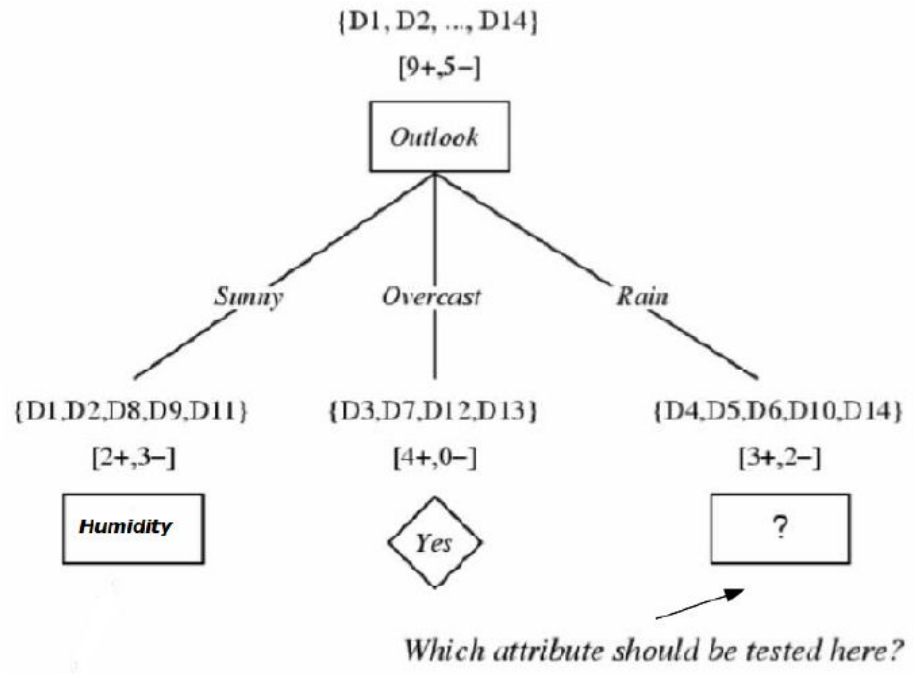
$S_{cool} = [1+, 0-] \Longrightarrow H(S_{cool}) = -(1/1) * \log_2(1/1) - (0/1) * \log_2(0/1) = 0$

$IG(S, \text{temperature}) = 0.971 - 2/5 * 0 - 2/5 * 1 - 1/5 * 0 = 0.570$

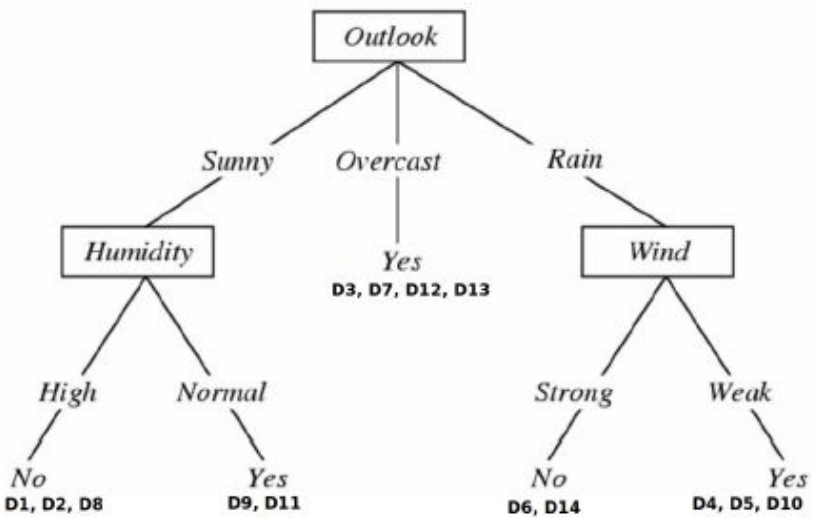
Likewise we can compute: $IG(S, \text{humidity}) = 0.970$, $IG(S, \text{wind}) = 0.019$

- Level-2, middle node: no need to grow (already a leaf)
- Level-2, right node: repeat the same exercise!
  - Compute IG for each feature (except outlook)
- Level-2 expansion gives us the following tree:





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"**)
  - Or we run out of features to test!

Decision Tree Learning Algorithm

A recursive algorithm:

DT(Examples, Labels, Features):

If all examples are positive, return a single node tree *Root* with label $= +$

If all examples are negative, return a single node tree *Root* with label $=$ -

If all features exhausted, return a single node tree *Root* with majority label

Otherwise, let F be the feature having the highest information gain

$Root \leftarrow F$

For each possible value f of F

- Add a tree branch below *Root* corresponding to the test $F = f$
- Let $Examples_f$ be the set of examples with feature F having value f
- Let $Labels_f$ be the corresponding labels
- If $Examples_f$ is empty, add a leaf node below this branch with label $=$ most common label in *Examples*
- Otherwise, add the following subtree below this branch:

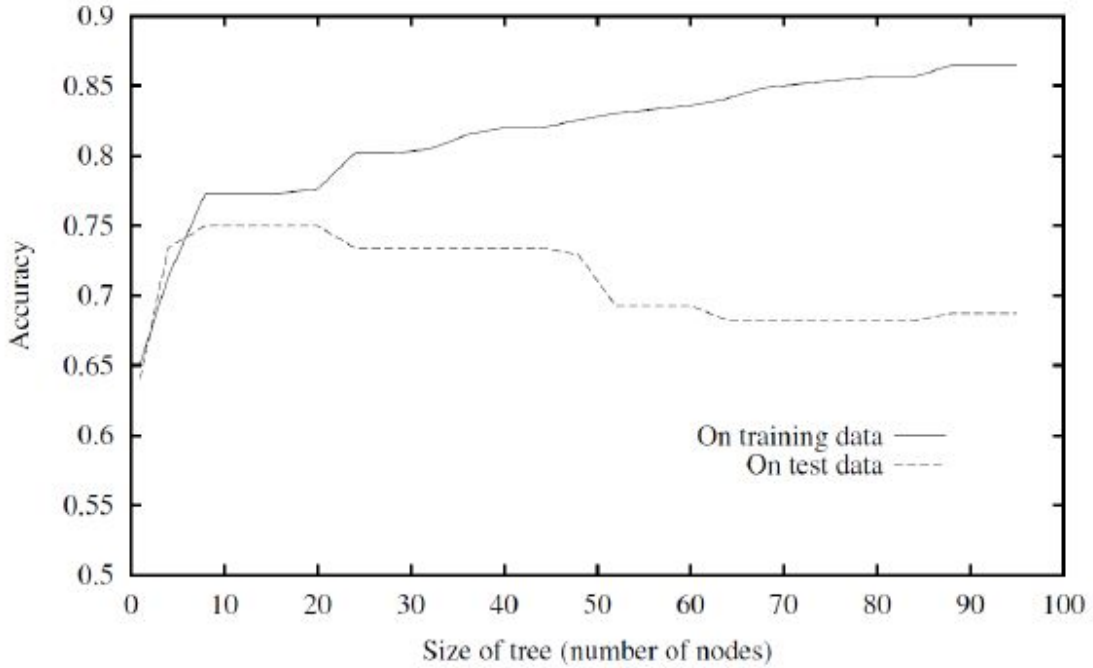
$$\mathrm{DT}(Examples_f,\ Labels_f,\ Features - \{F\})$$

- Note: $Features - \{F\}$ removes feature F from the feature set *Features*

OVERFITTING IN DECISION TREES

Overtting Illustration

- High training accuracy doesn't necessarily imply high test accuracy



Avoiding Overfitting: Decision Tree Pruning

- Desired: a DT that is not too big in size, yet ts the training data reasonably
- Mainly two approaches
  - 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
- Minimum Description Length (MDL): more details when we cover Model Selection

Decision Tree Extensions

- Real-valued features can be dealt with using thresholding
- Real-valued labels (Regression Trees) by re-defining entropy or using other criteria (how similar to each other are the **y**'s at any node)
- Other criteria for judging feature informativeness
  - Gini-index, misclassification rate
- More sophisticated decision rules at the internal nodes (anything that splits the data into homogeneous groups; e.g., a machine learning classifier)
- Handling features with differing costs

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 Aspects about Decision Trees

Some key weaknesses:

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

Logistic Regression

Not really a regression....

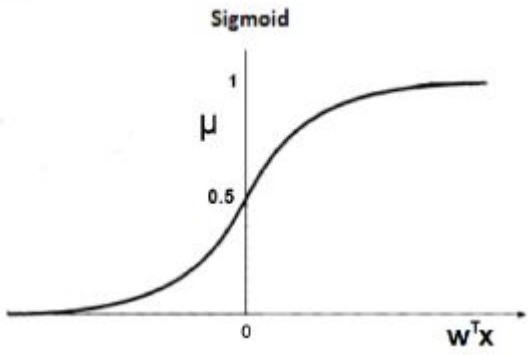
Logistic Regression: The Model

- A model for doing *probabilistic* binary classification
- Predicts label probabilities rather than a hard value of the label

$$p(y_n = 1|\boldsymbol{x}_n, \boldsymbol{w}) = \mu_n$$
$$p(y_n = 0|\boldsymbol{x}_n, \boldsymbol{w}) = 1 - \mu_n$$

- The model's prediction is a probability defined using the sigmoid function

$$f(\boldsymbol{x}_n) = \mu_n = \sigma(\mathbf{w}^\top \mathbf{x}_n) = \frac{1}{1 + \exp(-\mathbf{w}^\top \mathbf{x}_n)} = \frac{\exp(\mathbf{w}^\top \mathbf{x}_n)}{1 + \exp(\mathbf{w}^\top \mathbf{x}_n)}$$



- The sigmoid first computes a real-valued "score" $\mathbf{w}^\mathbf{T}\mathbf{x} = \sum w_d x_d$ and "squashes" it between (0,1) to turn this score into a probability score.
- Model parameter is the unknown **w**. Need to learn it from training data.

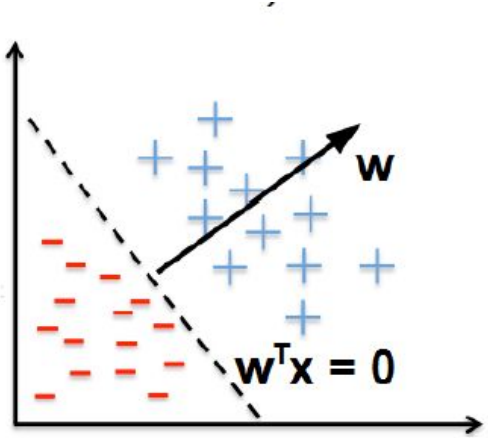
Logistic Regression: An Interpretation

- Recall that the logistic regression model defines

$$p(y=1|\boldsymbol{x},\boldsymbol{w}) = \mu = \sigma(\boldsymbol{w}^\top \boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}^\top \boldsymbol{x})} = \frac{\exp(\boldsymbol{w}^\top \boldsymbol{x})}{1+\exp(\boldsymbol{w}^\top \boldsymbol{x})}$$

$$p(y=0|\boldsymbol{x},\boldsymbol{w}) = 1-\mu = 1-\sigma(\boldsymbol{w}^\top \boldsymbol{x}) = \frac{1}{1+\exp(\boldsymbol{w}^\top \boldsymbol{x})}$$

- Thus if $\mathbf{w}^T\mathbf{x} > 0$ then the positive class is more probable.
- A linear classification model. Separates the two classes via a hyperplane (similar to other linear classification models such as Perceptron and SVM)



Logistic Regression: The Loss Function

- What loss function to use? One option is to use the squared loss

$$\ell(y_n, f(\boldsymbol{x}_n)) = (y_n - f(\boldsymbol{x}_n))^2 = (y_n - \mu_n)^2 = (y_n - \sigma(\boldsymbol{w}^\top \boldsymbol{x}_n))^2$$

- This is non-convex and not easy to optimize.
- Consider the following loss function

$$\ell(y_n, f(\boldsymbol{x}_n)) = \begin{cases} -\log(\mu_n) & y_n = 1 \\ -\log(1 - \mu_n) & y_n = 0 \end{cases}$$

- This loss function makes intuitive sense
  - If $y_n = 1$ but n is close to 0 (model makes error) then loss will be high
  - If $y_n = 0$ but n is close to 1 (model makes error) then loss will be high

- The above loss function can be combined and written more compactly as

$$\ell(y_n, f(\boldsymbol{x}_n)) = -y_n \log(\mu_n) - (1 - y_n) \log(1 - \mu_n)$$

- This is a function of the unknown parameter w since $\mu_n = (\mathbf{w}^T x_n)$

The loss function over the entire training data

$$L(\boldsymbol{w}) = \sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n)) = \sum_{n=1}^{N} [-y_n \log(\mu_n) - (1 - y_n) \log(1 - \mu_n)]$$

This is also known as the cross-entropy loss

- Sum of the cross-entropies b/w true label y_n and predicted label prob. μ_n

Plugging in $\mu_n = \frac{\exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}{1+\exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}$ and chugging, we get (verify yourself)

$$\boxed{L(\boldsymbol{w}) = -\sum_{n=1}^{N} (y_n \boldsymbol{w}^\top \boldsymbol{x}_n - \log(1 + \exp(\boldsymbol{w}^\top \boldsymbol{x}_n)))}$$

We can add a regularizer (e.g., squared ℓ_2 norm of $\boldsymbol{w}$) to prevent overfitting

$$\boxed{L(\boldsymbol{w}) = -\sum_{n=1}^{N} (y_n \boldsymbol{w}^\top \boldsymbol{x}_n - \log(1 + \exp(\boldsymbol{w}^\top \boldsymbol{x}_n))) + \lambda ||\boldsymbol{w}||^2}$$

Estimating the Weight Vector w

Loss function/NLL for logistic regression (ignoring the regularizer term)

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

The loss function is convex in $\boldsymbol{w}$ (thus has a unique minimum)

The gradient/derivative of $L(\boldsymbol{w})$ w.r.t. $\boldsymbol{w}$ (let's ignore the regularizer)

$$\begin{aligned} \mathrm{g} = \frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} &= \frac{\partial}{\partial \boldsymbol{w}}[-\sum_{n=1}^{N}(y_n \boldsymbol{w}^\top \boldsymbol{x}_n - \log(1 + \exp(\boldsymbol{w}^\top \boldsymbol{x}_n)))] \\ &= -\sum_{n=1}^{N}\left(y_n \boldsymbol{x}_n - \frac{\exp(\boldsymbol{w}^\top \boldsymbol{x}_n)}{(1 + \exp(\boldsymbol{w}^\top \boldsymbol{x}_n))}\boldsymbol{x}_n\right) \\ &= -\sum_{n=1}^{N}(y_n - \mu_n)\boldsymbol{x}_n = \mathbf{X}^\top(\boldsymbol{\mu} - \boldsymbol{y}) \end{aligned}$$

Can't get a closed form solution for $\boldsymbol{w}$ by setting the derivative to zero

- Need to use iterative methods (e.g., gradient descent) to solve for $\boldsymbol{w}$

Gradient Descent for Logistic Regression

We can use gradient descent (GD) to solve for $\boldsymbol{w}$ as follows:

- Initialize $\boldsymbol{w}^{(1)} \in \mathbb{R}^D$ randomly.
- Iterate the following until convergence

$$\underbrace{\boldsymbol{w}^{(t+1)}}_{\text{new value}} = \underbrace{\boldsymbol{w}^{(t)}}_{\text{previous value}} - \eta \underbrace{\sum_{n=1}^{N} (\mu_n^{(t)} - y_n)\boldsymbol{x}_n}_{\text{gradient at previous value}}$$

where η is the learning rate and $\mu^{(t)} = \sigma(\boldsymbol{w}^{(t)\top} \boldsymbol{x}_n)$ is the predicted label probability for $\boldsymbol{x}_n$ using $\boldsymbol{w} = \boldsymbol{w}^{(t)}$ from the previous iteration

Note that the updates give larger weights to those examples on which the current model makes larger mistakes, as measured by $(\mu_n^{(t)} - y_n)$

More on Gradient Descent..

GD can converge slowly and is also sensitive to the step size

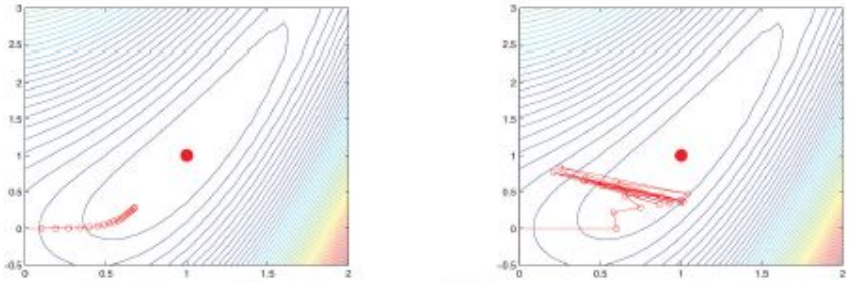


Figure: Left: small step sizes. Right: large step sizes

Several ways to remedy this[1]. E.g.,

- Choose the optimal step size η_t (different in each iteration) by line-search
- Add a momentum term to the updates

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \mathbf{g}^{(t)} + \alpha_t(\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)})$$

- Use second-order methods (e.g., **Newton's method**) to exploit the curvature of the loss function $L(\boldsymbol{w})$: Requires computing the Hessian matrix

That's all Folks!

Checkout the Python Notebook again now.