

# Non-Linear Models

## *Linear Models as Building Blocks*

Understanding Machine Learning

# Preview

reminder: Most ML algorithms can be described by the general recipe of combining models, costs, and optimization methods.

moreover: Most powerful ML algorithms are compound, with rather simple (often linear) building blocks.

look at some important examples:

- neural networks
- support-vector machines
- decision trees, random forests, gradient-boosted decision trees

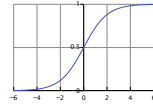
# Neural Networks

# Artificial Neuron

linear model with parameters called weights  $\mathbf{w}$

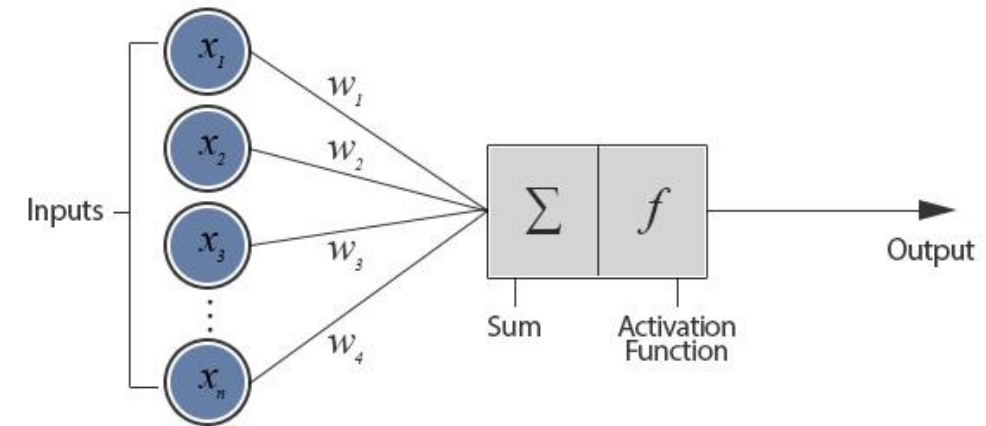
non-linear via (differentiable) activation function on sum of inputs times weights

- sigmoid, tanh, ...
- nowadays, mainly ReLU (Rectified Linear Unit), more on this later ...

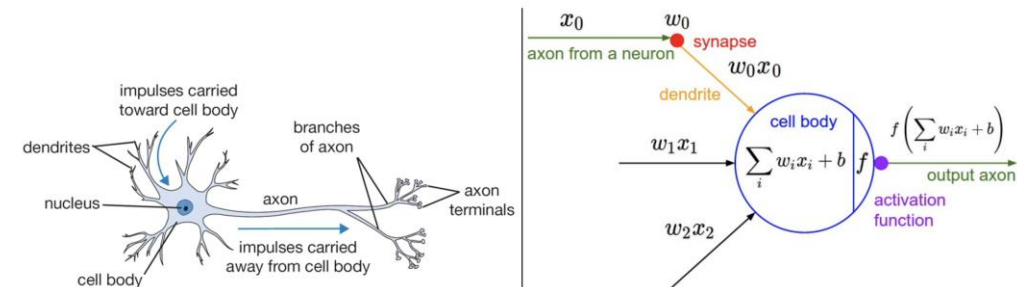


bias node/input to model intercept

artificial neuron (perceptron or node in neural network):



inspired from biological neurons:



# Historical: Perceptron

one neuron, or several ones in single (output) layer

→ linear model (cannot learn XOR)

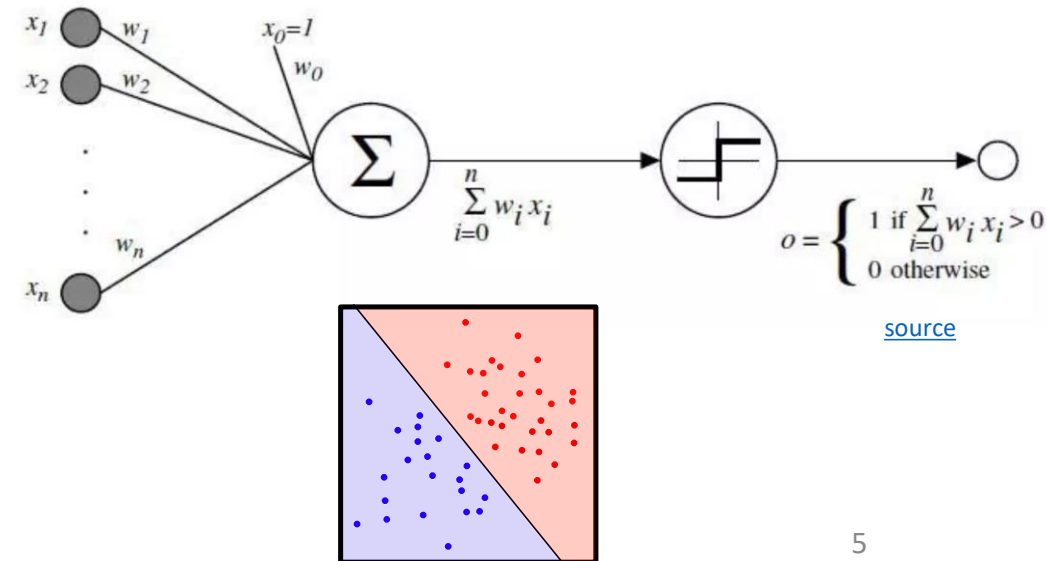
learning, e.g., via delta rule:

$$\Delta w_k = \eta(y - \hat{y})x_k \quad (\text{for simplest, one-node regression case})$$

originally (Rosenblatt), with binary outputs (→ separating hyperplanes), threshold activation function, and perceptron learning rule:

for each training example (repeat until no more errors)

- if output should have been 0 but was 1, decrease the weights that had an input of 1
- if output should have been 1 but was 0, increase the weights that had an input of 1

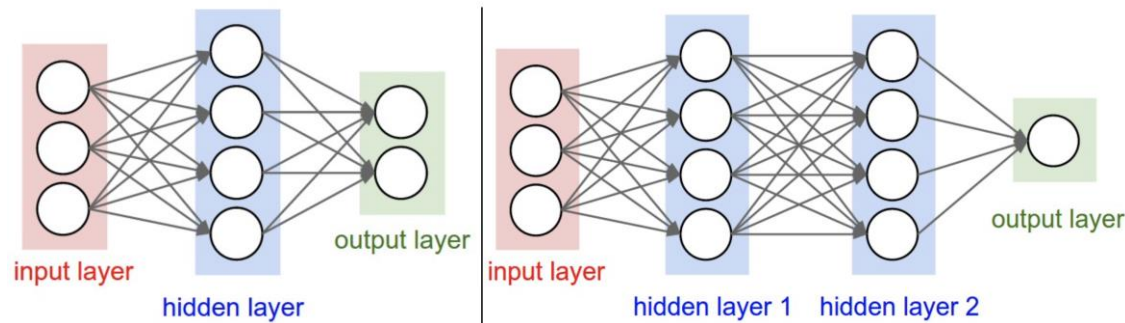


[source](#)

# Multi-Layer Perceptron (MLP)

fully-connected feed-forward network with at least one hidden layer

can learn XOR (in fact, universal function approximator → representation learning)



toward deep learning:  
add more nodes and hidden layers ...

classification:

- logistic regression in hidden nodes
- cross-entropy loss:  $L_i(y_i, \hat{f}(\mathbf{x}_i); \hat{\mathbf{w}}) = -\sum_{k=1}^K y_{ik} \log \hat{f}_k(\mathbf{x}_i; \hat{\mathbf{w}})$
- several output nodes  $k$  for multi-classification
- softmax output function:  $g_k(\mathbf{t}_i) = \frac{e^{t_{ik}}}{\sum_{l=1}^K e^{t_{il}}}$

regression:

- squared error loss
- identity output function
- usually just one output node

# History: AI Winter

AI hype (e.g., around perceptron) in late 1950s and 1960s

Minsky, Papert (1969): Rosenblatt's perceptron not able to learn XOR, no learning rule for multi-layer perceptrons (→ waiting for back-propagation)  
→ (or rather contributed to) AI winter in 1970s

until rise of expert systems in 1980s

its fall end of 1980 marked a second major AI winter (until mid 1990s)

but back-propagation (especially clear formulation of Hinton, Rumelhart, Williams) sparked (some) interest in neural networks again in late 1980s

# Back-Propagation

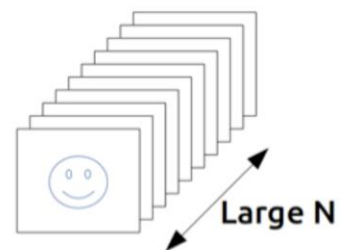


# Find Gradients for (Deep) Neural Networks

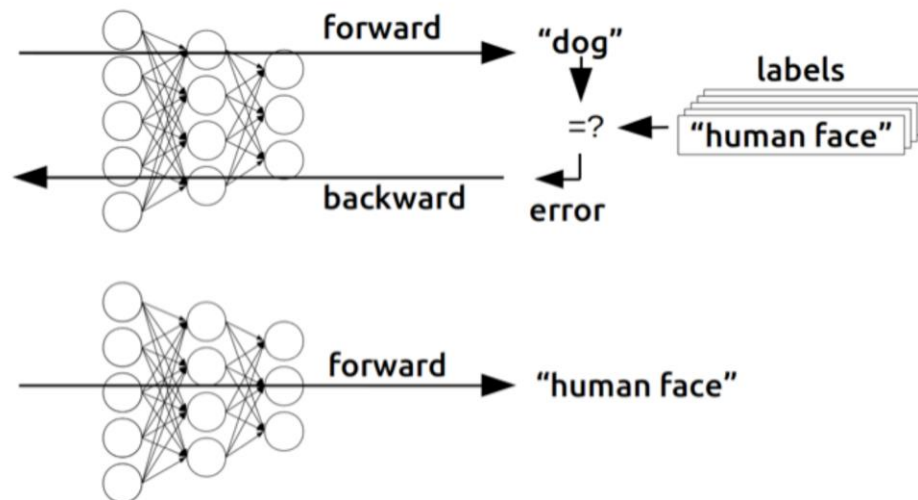
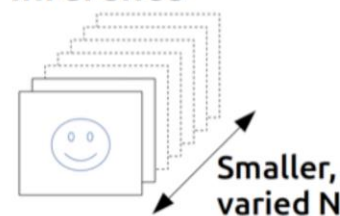
back-propagation of errors through layers via chain rule of calculus (avoiding redundant calculations of intermediate terms) → generalization of single-layer delta rule

each node exchanges information only with directly connected nodes → enables efficient, parallel computation

## Training



## Inference



- forward pass: current weights fixed, predictions computed
- backward pass: errors computed from predictions and back-propagated, weights updated accordingly, e.g., via gradient descent

# Example WOLOG

- regression (squared error loss, identity output function  $g$ )
- with one hidden layer ( $\hat{\mathbf{w}}: \hat{\alpha}, \hat{\beta}$ )

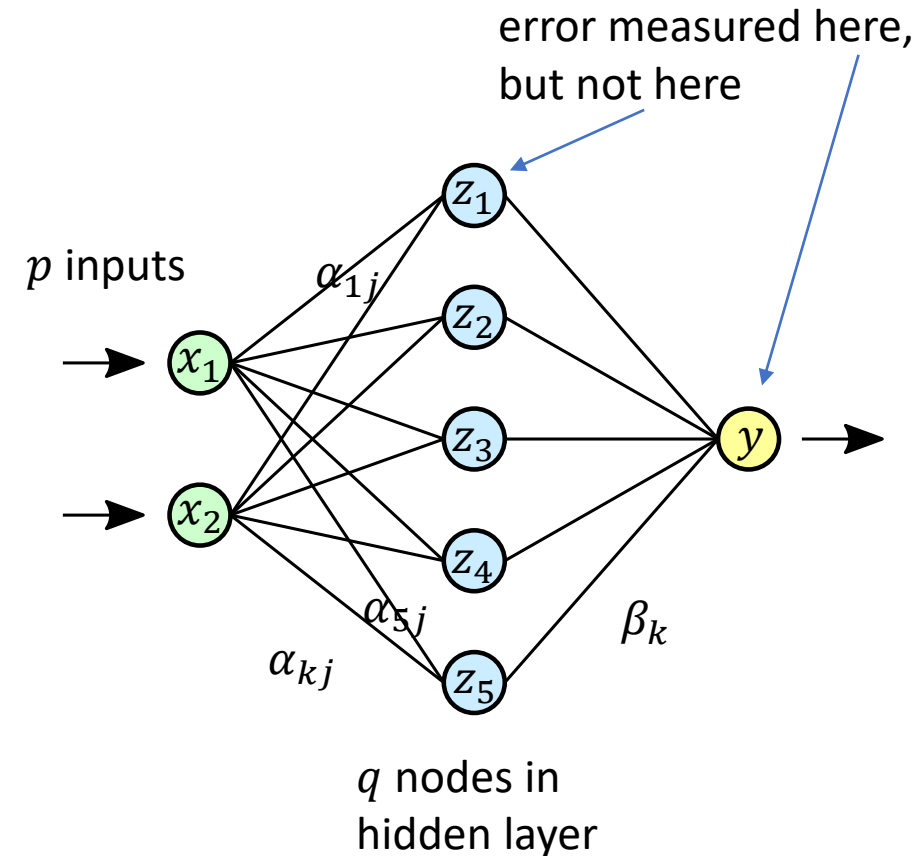
$$\hat{y}_i = \hat{f}(\mathbf{x}_i; \hat{\mathbf{w}}) = g(\mathbf{z}_i; \hat{\beta}) = \sum_{k=0}^q \hat{\beta}_k z_{ik}$$

$$z_{ik} = h(\mathbf{x}_i; \hat{\alpha}_k) = h\left(\sum_{j=0}^p \hat{\alpha}_{kj} x_{ij}\right)$$

activation  
function

cost function:

$$J(\hat{\mathbf{w}}) = \sum_{i=1}^n L_i(y_i, \hat{f}(\mathbf{x}_i); \hat{\mathbf{w}}) = \sum_{i=1}^n \left(y_i - \hat{f}(\mathbf{x}_i; \hat{\alpha}, \hat{\beta})\right)^2$$



# Example WOLOG

gradients:

$$\frac{\partial L_i}{\partial \hat{\beta}_k} = -2 \left( y_i - \hat{f}(\mathbf{x}_i; \hat{\mathbf{w}}) \right) z_{ik} = \delta_i z_{ik}$$
$$\frac{\partial L_i}{\partial \hat{\alpha}_{kj}} = -2 \left( y_i - \hat{f}(\mathbf{x}_i; \hat{\mathbf{w}}) \right) \hat{\beta}_k h'_k \left( \sum_{j=0}^p \hat{\alpha}_{kj} x_{ij} \right) x_{ij} = s_{ik} x_{ij}$$

→ back-propagation equations (use errors of later layers to calculate errors of earlier ones):

$$s_{ik} = h'_k \left( \sum_{j=0}^p \hat{\alpha}_{kj} x_{ij} \right) \hat{\beta}_k \delta_i$$

---

computed gradients then used, e.g., in gradient descent ( $r$  denoting iteration), to update weights:

$$\hat{\beta}_k^{(r+1)} = \hat{\beta}_k^{(r)} - \eta_r \sum_{i=1}^n \frac{\partial L_i}{\partial \hat{\beta}_k^{(r)}} \qquad \hat{\alpha}_{kj}^{(r+1)} = \hat{\alpha}_{kj}^{(r)} - \eta_r \sum_{i=1}^n \frac{\partial L_i}{\partial \hat{\alpha}_{kj}^{(r)}}$$

learning rate

# Using Gradients for Iterative Learning

use gradients found via back-propagation for iterative optimization  
usually, by means of gradient descent

- learning rate  $\eta_r$  potentially per iteration adjusted (e.g., via heuristic)
- not necessarily batch learning over full training epoch (one sweep through entire training data set): stochastic gradient descent updates after each training example, mini-batch gradient descent as compromise
- choose small random weights as starting values to break symmetry

→ back-propagation enables learning of deep neural networks (which can encode complex data representations in its hidden layers → feature learning on its own)

But there is so much more to tell about deep neural networks ...

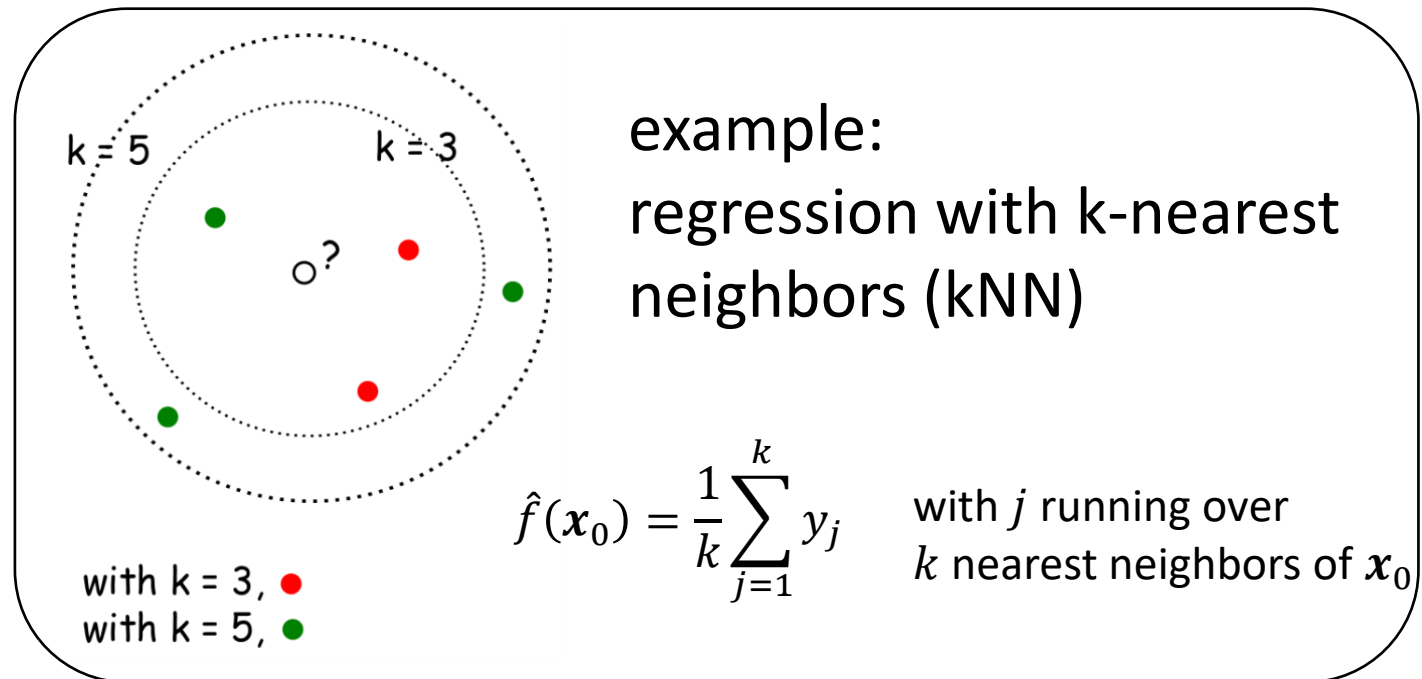
# Kernel Machines

# Instance-Based Learning

predict (e.g., classify) new data point by comparing with similar instances seen in training

→ need for a distance metric to choose nearest neighbors (e.g., Euclidean)

- non-parametric
  - store ( $n$ ) training data points to compare to in memory
  - computation only at inference time (lazy)
- potentially slow ( $O(n)$ )
- hit by curse of dimensionality (many features)



# Similarity Function

kernel machines: dot product  $\langle \mathbf{x}, \mathbf{x}_0 \rangle$ , aka inner or scalar product, between two data vectors  $\mathbf{x}$  (to be compared to) and  $\mathbf{x}_0$  (to be predicted) as similarity measure  $\rightarrow$  geometric interpretation

requirement: need for vectors in dot product space

$\rightarrow$  (potential) transformation of inputs to feature space:  $\mathbf{x} \mapsto \phi(\mathbf{x})$

additional advantage: non-linear mapping to suitable representation

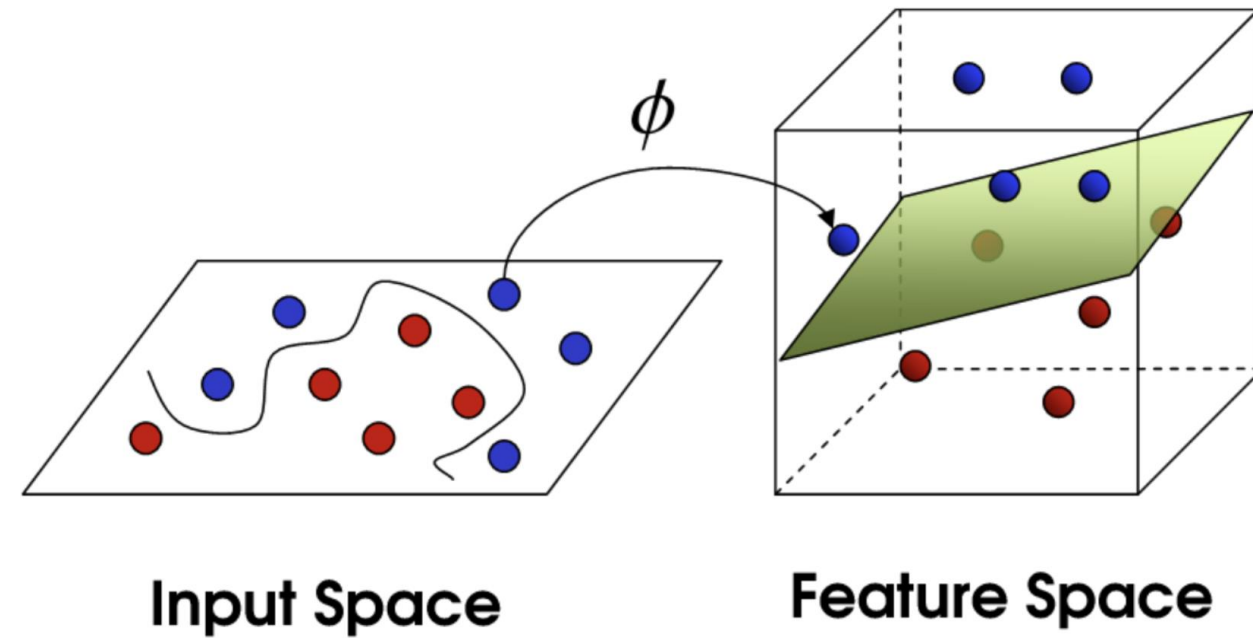
kernel:  $k(\mathbf{x}, \mathbf{x}_0) = \langle \phi(\mathbf{x}), \phi(\mathbf{x}_0) \rangle$

# Non-Linear Transformation

example: data not linearly separable in input space

solution: transformation into higher-dimensional feature space

→ linearly separable by hyperplane in feature space



from [towardsdatascience.com](https://towardsdatascience.com)

options:

- explicit feature engineering (feature map)
- definition of similarity function over pairs of data points → higher abstraction



# Kernel Trick

kernel function:  $k(\mathbf{x}, \mathbf{x}_0) = \langle \phi(\mathbf{x}), \phi(\mathbf{x}_0) \rangle$

- input: vectors in original space
- output: dot product of vectors in feature space

kernel machines only use kernel function to construct decision boundary

→ no need to calculate  $\phi(\mathbf{x})$

dot product usually computationally much cheaper

# Example: Gaussian Kernel

most popular radial basis function (RBF) kernel

$$k_{\lambda}(\mathbf{x}, \mathbf{x}_0) = \exp(-\lambda \underbrace{\|\mathbf{x} - \mathbf{x}_0\|^2}_{\text{Euclidean distance}})$$

hyperparameter

Euclidean distance

remember: kernel methods are non-parametric (instance-based)

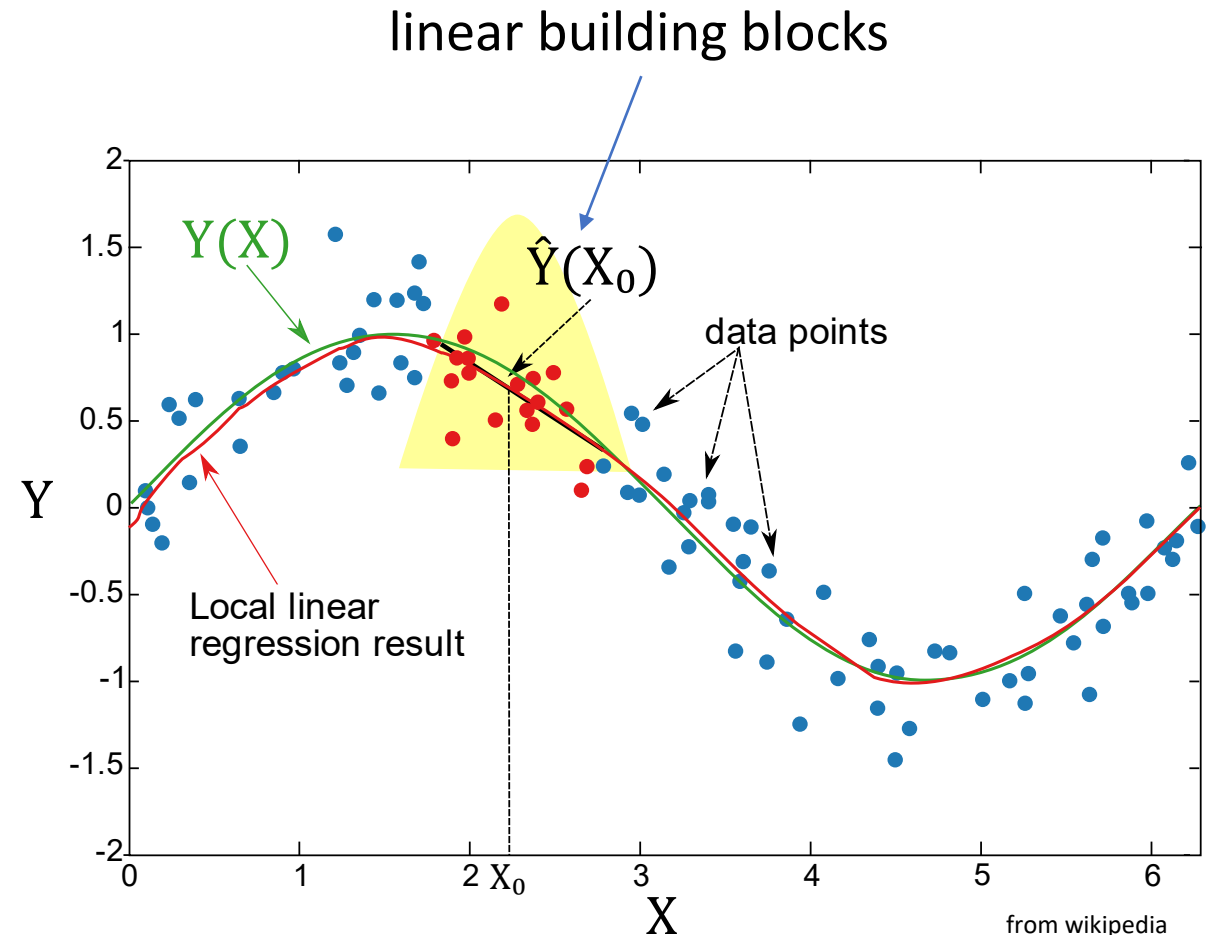
# Kernel Regression

simplest form of kernel regression:  
Nadaraya-Watson weighted average

$$\hat{f}(\mathbf{x}_0) = \frac{\sum_{i=1}^n k(\mathbf{x}_i, \mathbf{x}_0) y_i}{\sum_{i=1}^n k(\mathbf{x}_i, \mathbf{x}_0)}$$

local regression in general (including  
local linear regression): minimize

$$\begin{aligned} J(\hat{\boldsymbol{\theta}}_n, \mathbf{x}_0) \\ = \sum_{i=1}^n k(\mathbf{x}_i, \mathbf{x}_0) \left( y_i - \hat{f}(\mathbf{x}_i; \hat{\boldsymbol{\theta}}) \right)^2 \end{aligned}$$



# Relationship to k-Nearest Neighbors

kNN can be interpreted as kernel method with metric

$$k_k(\mathbf{x}, \mathbf{x}_0) = I(\|\mathbf{x} - \mathbf{x}_0\| \leq \|\mathbf{x}_{(k)} - \mathbf{x}_0\|)$$

calculating the Nadaraya-Watson weighted average

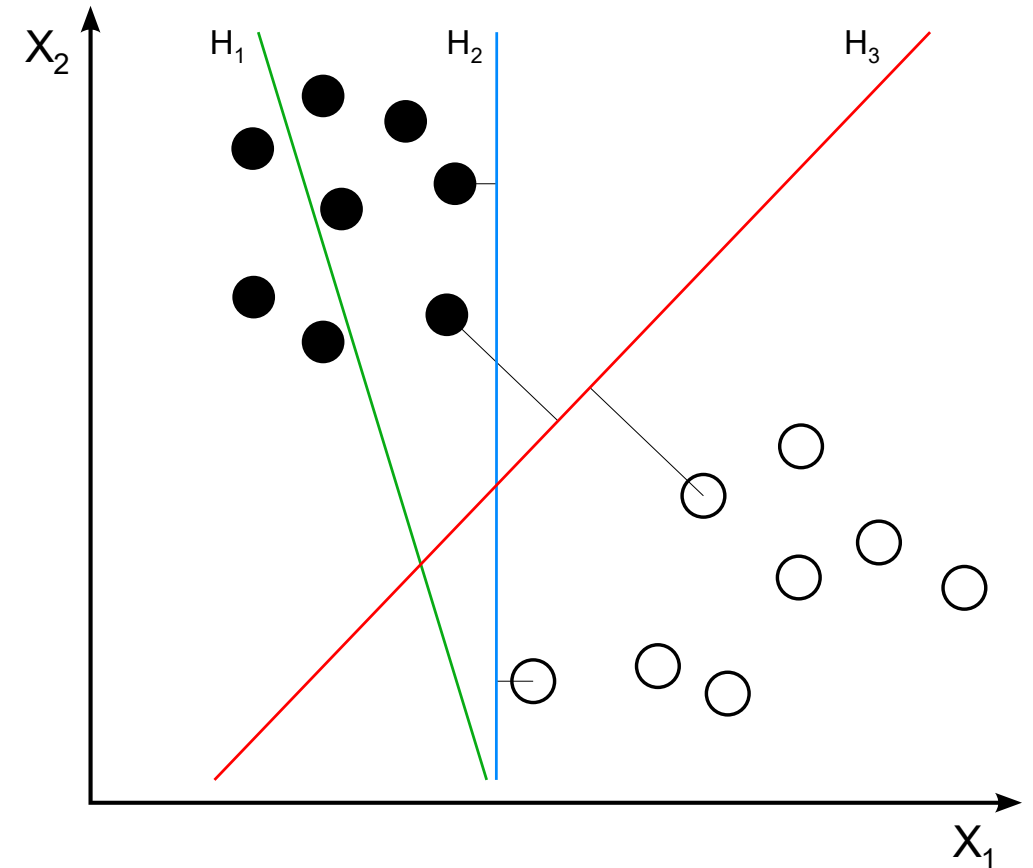
# Optimal Separating Hyperplanes

(toward support-vector machines)

linear classifier using  $p$  features  $\rightarrow$   
separation of classes by  $(p - 1)$ -  
dimensional hyperplane as decision  
boundary

searching for maximum-margin  
hyperplane: maximize distance from  
closest points on both sides of  
hyperplane

(in contrast to Rosenblatt's perceptron  
minimizing the distance of misclassified  
points to separating hyperplane)



from wikipedia

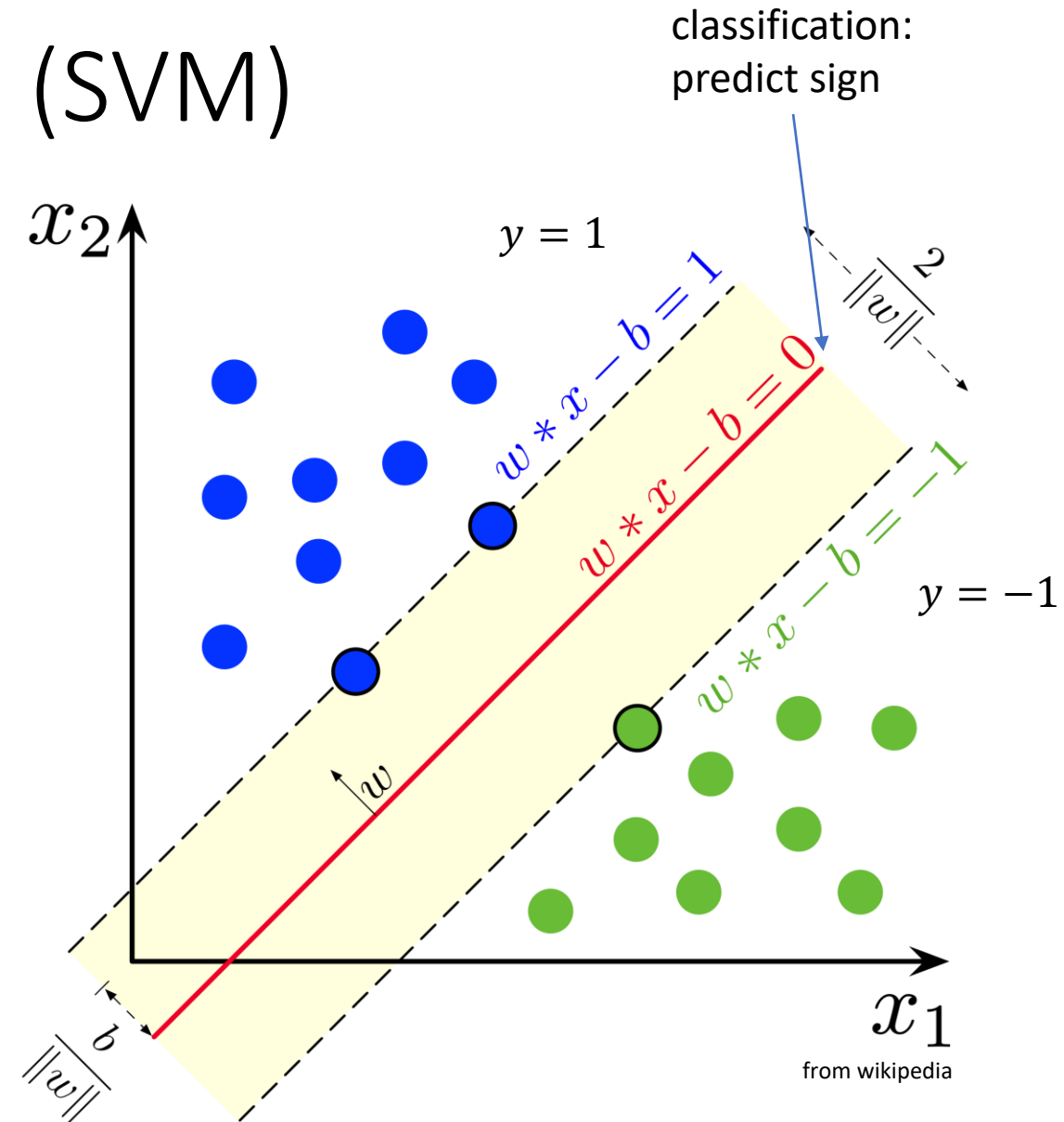
# Support-Vector Machines (SVM)

maximum-margin hyperplane completely determined by (linear combination of) data points closest to it: **support vectors**

(identification of support vectors requires all data though)

hard margin:

holds only if classes are linearly separable  
(for training, not necessarily for test data)



$\|w\|$  to be minimized

# Soft-Margin SVM

for most practical applications: need to deal with overlapping classes (soft margin)

→ minimize regularized hinge loss:

hinge loss:

- 0 for prediction with same sign as target and outside margin
  - linear increase for opposite sign or inside margin
- prefers larger  $\| \mathbf{w} \|$

$$\lambda \| \mathbf{w} \|^2 + \left( \frac{1}{n} \sum_{i=1}^n \max \left( 0, 1 - y_i (\mathbf{w}^T \mathbf{x}_i - b) \right) \right)$$

tradeoff between  
increasing margin  
(smaller  $\| \mathbf{w} \|$ ) and  
correct classification

$L^2$  regularization

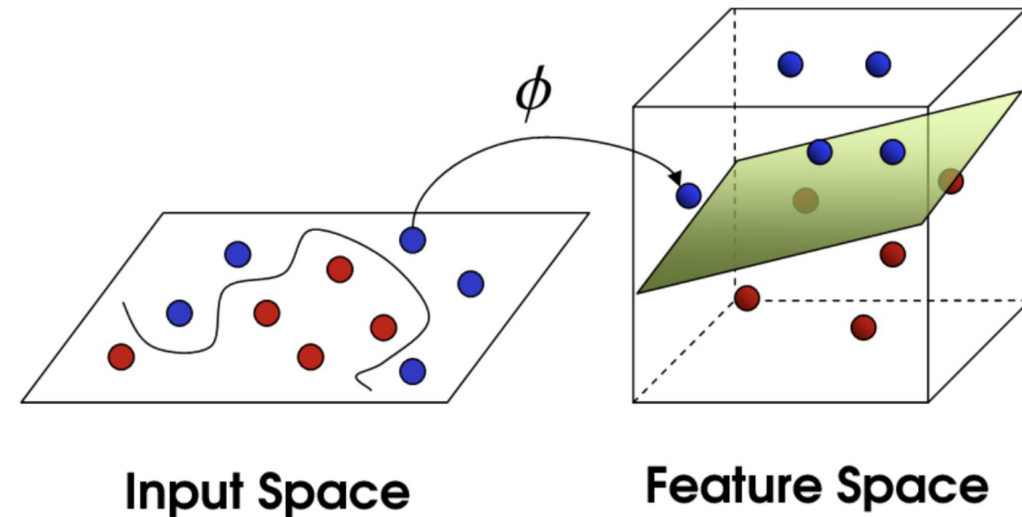
# Go Non-Linear via Kernels

SVM: finding maximum-margin hyperplane  
→ linear model

non-linear by means of kernel trick:  
replacing dot products by kernel functions  
 $k(\mathbf{x}, \mathbf{x}_0) = \langle \phi(\mathbf{x}), \phi(\mathbf{x}_0) \rangle$

( $\mathbf{w}$  expressed by  $\mathbf{x}$  and  $\mathbf{y}$ , all feature vectors only occur in dot products)

→ linear model in feature space, but  
non-linear model in input space



from [towardsdatascience.com](https://towardsdatascience.com)

- kernel method not restricted to SVMs
- non-linear generalization to any algorithm expressible in dot products
- e.g., principal component analysis



# History: Another AI (Neural Network) Winter

in mid 1990s, neural networks with several layers (nowadays called deep learning) did not work well yet for practical applications (except for CNNs to some degree)

- issues with vanishing/exploding gradients in backpropagation
- hardware not yet fast enough, not enough data

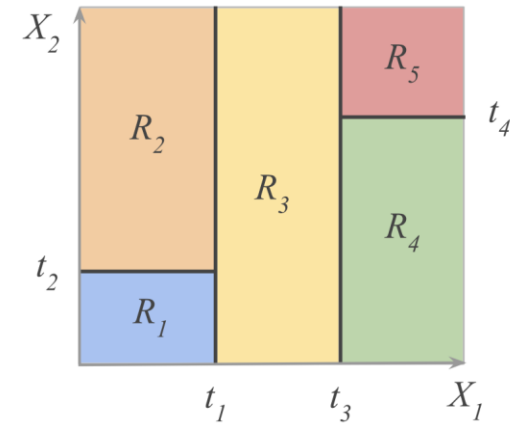
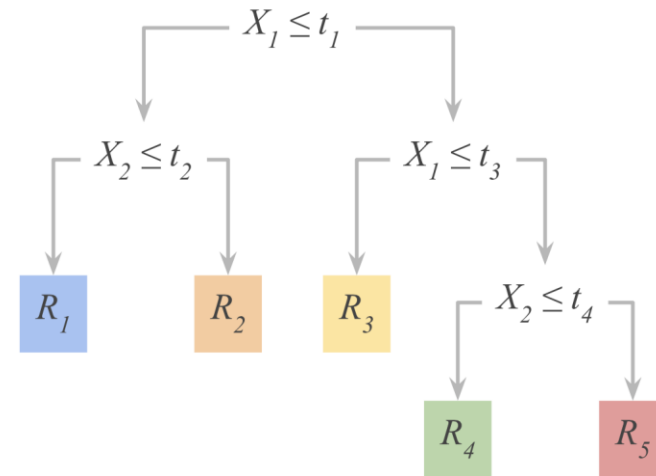
→ advent of SVMs (and other methods like random forests) marked another small AI winter (at least for neural networks)

- solidly founded in statistical learning theory
- kind of equivalent to two-layer neural networks
- easier to handle than difficult to work with neural networks
- kernel trick (feature space transformation) reduces need for a priori knowledge

# Decision Trees

# White-Box Model

- non-parametric learning of simple decision rules
- usually, binary trees
- axis-parallel decision boundaries (box-shaped regions in feature space)
- fit constant  $\hat{c}$  in each box (note similarity to kNN)
  - classification: majority class of target
  - regression: average of target
- fully explainable models



from Hastie


# Decision Tree Learning

finding optimal partitions  $R$  by overall loss minimization computationally infeasible  $\rightarrow$  recursive partitioning of data (greedy algorithm)

for each binary partition into regions  $R_1$  and  $R_2$ , decisions on splitting variable  $j$  and split point  $s$  by minimizing impurity functions  $I$  (weighted by number of observations  $N$  in child nodes):

$$R_1(j, s) = \{\mathbf{x} | x_j \leq s\} \quad \text{and} \quad R_2(j, s) = \{\mathbf{x} | x_j > s\}$$

$$\operatorname{argmin}_{j,s} \left[ \frac{N_{R_1}}{N} \operatorname{argmin}_{\hat{c}_1} (I_{R_1}) + \frac{N_{R_2}}{N} \operatorname{argmin}_{\hat{c}_2} (I_{R_2}) \right]$$

 recursively for  
each node in tree

typical strategy:

- grow large tree down to minimum node size (hyperparameter)  $\rightarrow$  low bias
- to avoid overfitting: subsequent pruning by controlling the number of terminal nodes

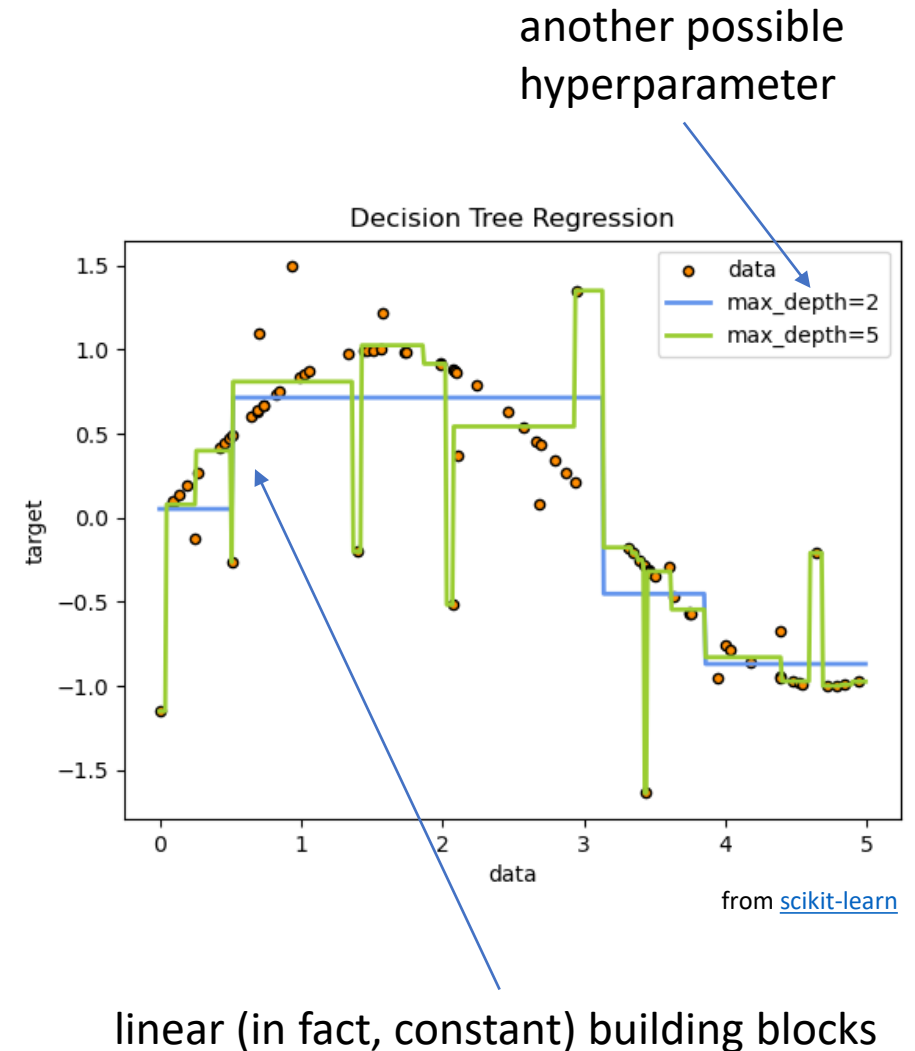
# Regression Trees

usually, mean squared error as impurity function (analogous for  $R_2$ ):

$$I_{R_1} = \sum_{x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2$$

→  $\operatorname{argmin}_{\hat{c}_1} (I_{R_1})$  solved by  $\hat{c}_1 = \bar{y}_{R_1}$

for  $\operatorname{argmin}_{j,s}$ , choose boxes to make target averages in each box as different as possible



# Impurity Functions for Classification Trees

classification in given node according to majority class

several possibilities for impurity function used for classification, preferred ones:

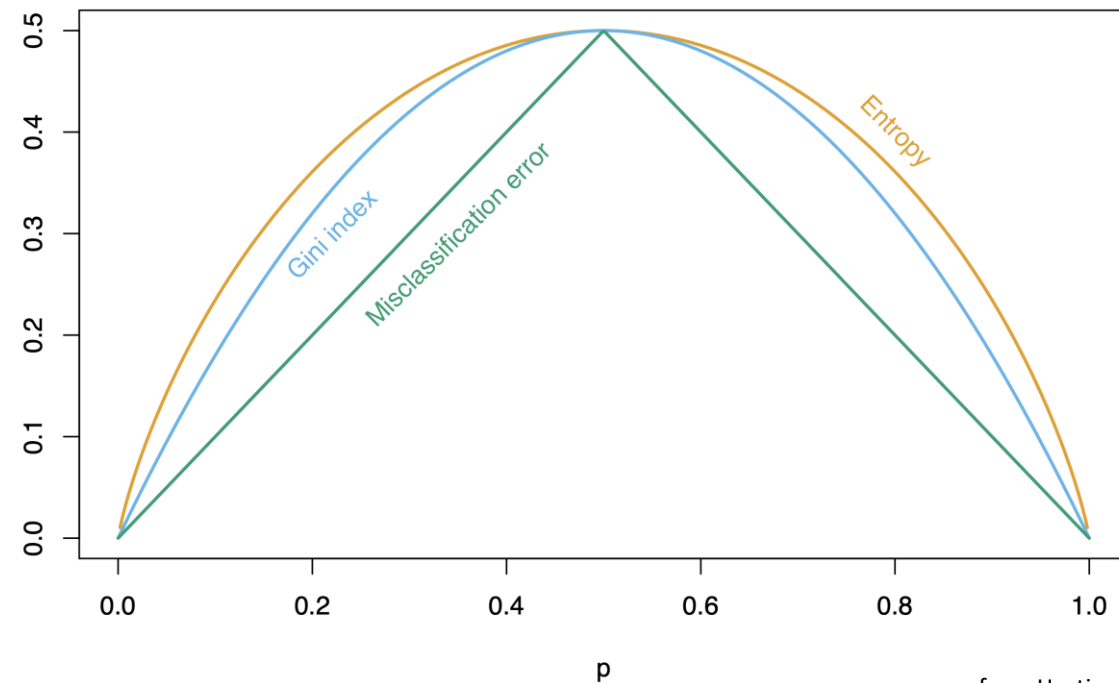
- Gini index (used in popular CART algorithm):

$$2p(1 - p)$$

- difference in entropy, aka information gain (used in popular C4.5 algorithm):

$$-p \log p - (1 - p) \log(1 - p)$$

node impurity as function of class proportion  $p$  (two-class classification):



from Hastie

# Ensemble Methods

# Ensemble Learning

idea:

combine several individual models (often of the same type) to form an ensemble model with better predictive performance than each of its constituents

learning in several different ways from the training data

- the trainings of the individual constituent models are (kind of) independent
- outputs of individual models are combined (e.g., averaged)



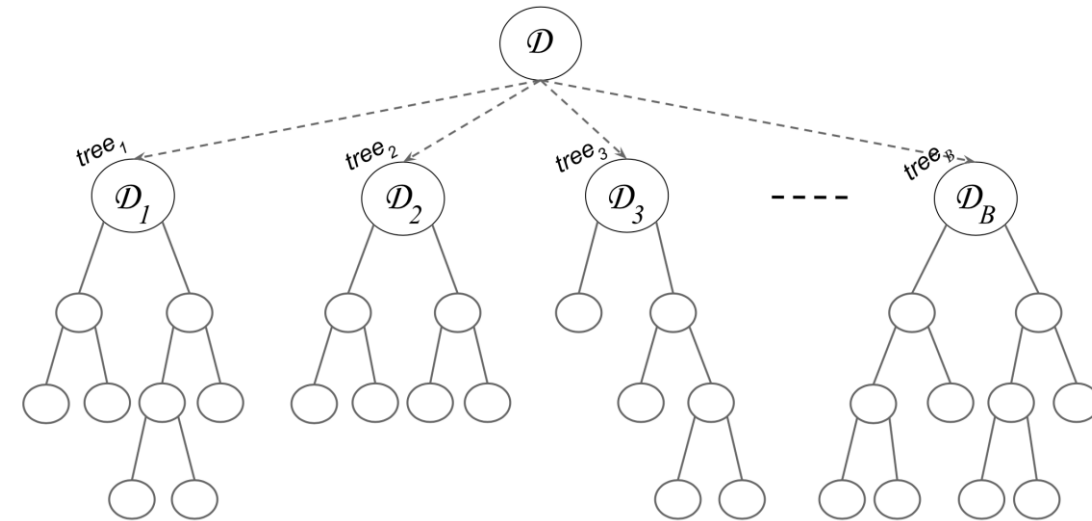
# Bagging (Bootstrap Aggregating)

idea: train individual models of ensemble method on random (sub)sets of the training data (random sample with replacement)

- reduces variance of ensemble model compared to individual models
- but not bias  $\rightarrow$  use low-bias base models

individual models combined by averaging (regression) or majority voting (classification)  $\rightarrow$  committee of models

example with decision trees:



[source](#)

# Random Subspace Method

aka feature bagging

idea: train different members (models) of an ensemble method on random subsets of all features (random sample with replacement)

→ reduce correlation between ensemble members

individual models combined by averaging (regression) or majority voting (classification) → committee of models

# Random Forest

ensemble method using

- decision trees as individual models (usually, rather large, low-bias decision trees)
- combination of bagging and random subspace method (features sampled at each node in the trees)

compared to individual decision trees, random forests

+ reduce variance → improve accuracy

- lose explainability

one of the most popular off-the-shelf ML algorithms (often, good performance with little hyperparameter tuning and data preparation)

# Boosting

idea: sequentially learn and combine several “weak” learners (such as small decision trees, but in principle any ML algorithm) to construct a “strong” one

→ gradually (in a greedy fashion) reducing bias of ensemble model

in simple terms:

building a model from the training data, then creating a second model that attempts to correct the errors from the first model, ...

→ each subsequent weak learner is forced to concentrate on the examples that are missed by the previous ones in the sequence

not a committee of models

typically, use simple, high-bias methods as individual models

# Forward Stagewise Additive Modeling

boosting can be understood as fitting an additive expansion in a set of basis functions  $b(\mathbf{x}; \hat{\gamma}_m)$  (e.g., decision trees, with  $\hat{\gamma}_m$  parametrizing splits):  $\hat{f}(\mathbf{x}) = \sum_{m=1}^M \hat{\beta}_m b(\mathbf{x}; \hat{\gamma}_m)$

loss minimization of  $\hat{f}(\mathbf{x})$  computationally expensive  $\rightarrow$  boosting: sequentially add and fit individual basis functions without changing already fitted ones

boosting algorithm for sequential optimization using basis functions:

1. initialize  $\hat{f}_0(\mathbf{x}) = 0$
2. for  $m = 1$  to  $M$ 
  - a) compute 
$$\argmin_{\hat{\beta}_m, \hat{\gamma}_m} \sum_{i=1}^N L\left(y_i, \hat{f}_{m-1}(\mathbf{x}_i) + \hat{\beta}_m b(\mathbf{x}_i; \hat{\gamma}_m)\right)$$
  - b) set 
$$\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + \hat{\beta}_m b(\mathbf{x}; \hat{\gamma}_m)$$

# AdaBoost (Adaptive Boosting)

in its simplest form: binary classification

forward stagewise additive modeling with individual classifiers

$\hat{G}_m$  and loss function  $L(y, \hat{f}(x)) = e^{-y \hat{f}(x)}$

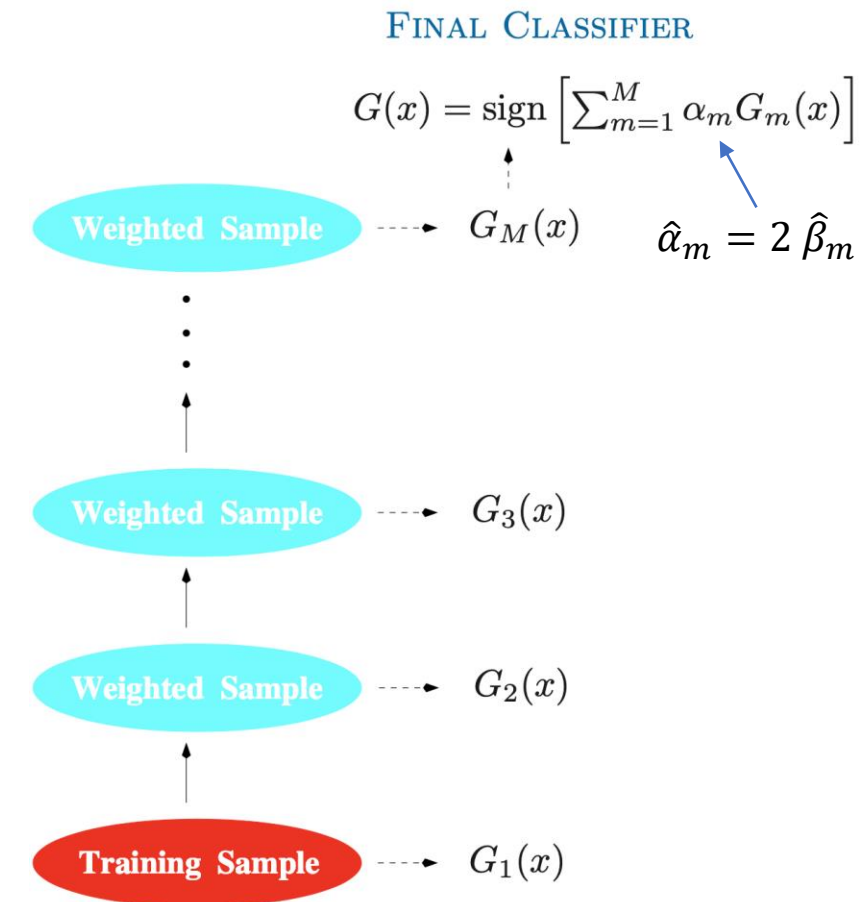
$$(\hat{\beta}_m, \hat{G}_m) = \underset{\hat{\beta}, \hat{G}}{\operatorname{argmin}} \sum_{i=1}^N \underbrace{e^{-y_i \hat{f}_{m-1}(x_i)}}_{w_i((x_i, y_i))} e^{-\hat{\beta} y_i \hat{G}(x_i)}$$

two steps:

(1): weighted error rate minimized by  $\hat{G}_m$ :

$$\operatorname{err}_m = \frac{\sum_{i=1}^N w_i^{(m)} I(y_i \neq \hat{G}_m(x_i))}{\sum_{i=1}^N w_i^{(m)}} \quad \rightarrow (2): \hat{\beta}_m = \frac{1}{2} \log \frac{1 - \operatorname{err}_m}{\operatorname{err}_m}$$

corresponds to sample weights  $w_i$  (in the trainings for  $\hat{G}_m$ ), reflecting accuracy of current ensemble at each successive iteration (weights increased for incorrect predictions, decreased for correct ones)



# Gradient Boosting

approximation of forward stagewise additive modeling (solution can be hard for general  $L$ ):

$$\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + \hat{h}_m(\mathbf{x}) \quad \text{by successively solving} \quad \underset{\hat{h}_m}{\operatorname{argmin}} \sum_{i=1}^N L(y_i, \hat{f}_m(\mathbf{x}_i))$$

$$\tilde{L}(y_i, \hat{f}_m(\mathbf{x}_i)) \approx L(y_i, \hat{f}_{m-1}(\mathbf{x}_i)) + \hat{h}_m(\mathbf{x}_i) \underbrace{\left[ \frac{\partial L(y_i, \hat{f}(\mathbf{x}_i))}{\partial \hat{f}(\mathbf{x}_i)} \right]_{\hat{f}=\hat{f}_{m-1}}}_{g_{im}}$$

$$\rightarrow \hat{h}_m(\mathbf{x}) \approx \underset{\hat{h}_m}{\operatorname{argmin}} \sum_{i=1}^N \hat{h}_m(\mathbf{x}_i) g_{im}$$

$\hat{h}_m$  fitted to predict negative gradients of samples at each iteration (regression model), corresponding to gradient descent in functional space

- $\hat{h}_m$ : usually, decision tree regressors of fixed size (gradient-boosted decision trees)
- works like this for both regression and classification (just different loss functions)

# Decision Tree Sizes for Boosting

consider degree to which features interact with each other in given data set  
(see ANOVA expansion)

- decision trees with single split (aka decision stumps): covering no interaction effects, just main effects of individual features
  - decision trees with two splits: covering second-order interactions
  - decision trees with three splits: covering third-order interactions
- (usually interaction order not much higher than  $\sim 5$ )

why boosting often works better than single large, low-bias model:  
uncorrelated learners, each focusing on a specific aspect of the data



# Literature

papers:

- back-propagation: one of the founding moments of deep learning
- Gradient Boosting
- XGBoost: uses second-order Taylor approximation in loss function (Newton-Raphson instead of gradient descent)
- LightGBM: uses improved histogram-based algorithm (gains in computational speed and memory consumption)
- CatBoost: uses kind of leave-one-out encoding for categorical features

# Support Scientific Progress

AI/ML as tool to supercharge the scientific method

prime example medicine:

drug discovery, medical imaging, gene classification, protein structure prediction ([AlphaFold](#) [[1](#), [2](#)])

but also many other areas, e.g., material sciences ([M3GNet](#))