# Boosting

Gradient Boosting, AdaBoost, RankBoost, and LambdaMART — Consolidated Lecture Notes, CS6140 Machine Learning

---

## 1 What Is Boosting?

- A **weak learner** is any model that does just a bit better than chance (e.g. a decision stump, or a shallow tree) — individually mediocre, but each one knows something the others don't.
- **Boosting** combines many weak learners into one strong additive ensemble, built *forward, stage-wise*: never revisit or reweight an earlier learner, just add a new one that fixes what's currently wrong.

$$F(x) = \sum_{t=1}^{T} \rho_t \, h_t(x) \tag{1}$$

  Read this in plain English: $h_t(x)$ is round $t$'s weak learner's opinion, and $\rho_t$ is *how much that opinion counts* — a per-round credit/step-size, exactly like a coefficient in a linear model, except here the "feature" $h_t(x)$ was itself fit to the data, one round at a time, instead of being handed to you in advance.

- Two historical answers to "how do we decide what's currently wrong?":
  - **Reweight the data** (AdaBoost, 1996/97): points the current ensemble gets wrong get *more weight*; the next weak learner is trained to do well on the reweighted data. First successful boosting algorithm, foundational but — as we'll see in §6 — superseded in practice.
  - **Follow the gradient** (Gradient Boosting, Breiman 1997–99, Friedman 2001): treat $F(x_1), \ldots, F(x_n)$ as free parameters, take the gradient of a loss function with respect to them, and fit the next weak learner to (the negative of) that gradient. Strictly more general — works for *any* differentiable loss, so it covers regression, classification, and (with the right loss) ranking, all with the same recursive tree-fitting machinery from the decision-tree notes.

⇒ Both are the same forward stage-wise recipe; they differ only in *what signal* tells the next weak learner what to fix. §6 shows AdaBoost is in fact a special case of the gradient view (exponential loss) — so gradient boosting is the one framework, not two.

**Remark.** Why an ensemble of $T$ small trees instead of one big tree? A single tree of depth $d$ has $O(2^d)$ leaves — capacity grows *exponentially* in depth, and it overfits fast. An ensemble of $T$ shallow (depth 2–5) trees has capacity growing only *linearly*, $O(T)$, in the number of rounds — a much gentler knob, and (as in the decision-tree notes' pruning discussion) one you tune by cross-validation on a held-out set.

**Remark.** **Boosting vs. bagging/Random Forests.** Both are tree ensembles, but they attack opposite ends of the bias-variance tradeoff. **Bagging** (Random Forests) trains many *independent*, deep,

low-bias/high-variance trees on bootstrap-resampled data and *averages* them — averaging cancels out variance, so the trees themselves are grown as large/unstable as possible. **Boosting** trains a *sequence* of shallow, high-bias/low-variance trees, each one correcting the ensemble's current systematic error — sequential correction reduces bias, so the trees are deliberately kept weak. Same building block (decision trees), opposite mechanism, opposite failure mode if misused: an over-deep bagged forest just wastes trees; an over-long or too-fast-shrinking boosted ensemble can genuinely overfit.

**Roadmap:** §2 derives gradient boosting for regression (the version HW1 Problem 3 implements) → §3 derives it for classification (log-loss, softmax, and the AdaBoost-connecting exponential-loss view) → §4 implementation details, real code, worked examples → §5 LambdaMART for learning-to-rank (which reuses *exactly* the Newton-step machinery from §3) → §6 AdaBoost, presented last as history, since §3 will already have shown it's a special case of the gradient view → §7 RankBoost, immediately after — it's AdaBoost's reweighting idea, run on pairs.

# 2 Gradient Boosting for Regression

**The game.** You're given data $(x_1, y_1), \ldots, (x_n, y_n)$ and a model $F$ that's good but imperfect. Rules: you may not touch $F$ or its parameters — you may only *add* a new model $h$, giving prediction $F(x) + h(x)$. How do you pick $h$?

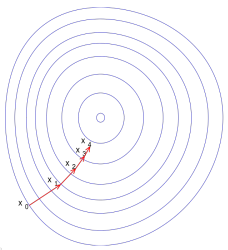
**Naive fix.** You'd like $F(x_i) + h(x_i) = y_i$ for every $i$, i.e. $h(x_i) = y_i - F(x_i)$. No single regression tree hits this exactly, but one can get *close*: fit a regression tree $h$ to the data $(x_i,\ y_i - F(x_i))$. The quantities $y_i - F(x_i)$ are the **residuals** — the part of $y$ that $F$ doesn't yet explain. Add $h$ to $F$; if still unsatisfactory, repeat with another tree.

**Why this is gradient descent.** For square loss $L(y, F) = \frac{1}{2}(y-F)^2$, treat each $F(x_i)$ as a free parameter and minimize $J = \sum_i L(y_i, F(x_i))$:

$$\frac{\partial J}{\partial F(x_i)} = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = F(x_i) - y_i \quad \Rightarrow \quad y_i - F(x_i) = -\frac{\partial J}{\partial F(x_i)} \tag{2}$$

So for square loss, **residual = negative gradient**, and "fit $h$ to the residual, then $F := F + h$" *is* one step of gradient descent in the ($n$-dimensional) space of predictions, with step size $\rho = 1$:

$$F(x_i) := F(x_i) + h(x_i) = F(x_i) + \left(-\frac{\partial J}{\partial F(x_i)}\right) \qquad \text{cf.} \qquad \theta_i := \theta_i - \rho\,\frac{\partial J}{\partial \theta_i} \tag{3}$$



Gradient descent: move against the gradient, one small step at a time. Gradient boosting does the same thing, except the "step" is a fitted regression tree, not a fixed-direction vector.

**General recipe (any differentiable loss).**

**Algorithm: Gradient Boosting for Regression**

1. Initialize $F(x) = \bar{y} = \frac{1}{n}\sum_i y_i$ (or, more generally, $\arg\min_\gamma \sum_i L(y_i, \gamma)$).
2. Iterate $t = 1, \ldots, T$:
   (a) Compute negative gradients $-g(x_i) = -\dfrac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$ at each training point.
   (b) Fit a regression tree $h_t$ to $(x_i,\ -g(x_i))_{i=1}^n$ (*exactly* the regression-tree splitter from the decision-tree notes, targets swapped from $y$ to $-g$).
   (c) Update $F := F + \rho\, h_t$ (take $\rho = 1$, or a shrinkage/learning-rate constant $\eta < 1$ — see §4).

---

**Remark. Why fit a *shallow* tree to the gradient, specifically?** A weak learner's job each round is to spot one coarse, low-order pattern in *where the ensemble is currently wrong* — e.g. "for houses with few bedrooms in cold climates, we're under-predicting" — not to explain the gradient perfectly. A depth-$d$ tree can only represent interactions among at most $d$ features, so depth is a direct dial on *how complex a pattern one round is allowed to grab*: depth 1 (a stump) finds one feature's worth of correction; depth 3 lets three features jointly matter in one round. Deep trees would instead memorize each round's gradient noise (overfitting one round at a time, the tree-ensemble version of what §1's capacity remark warns about); very shallow trees need more rounds $T$ to converge but generalize better — the same depth-vs-rounds tradeoff as choosing $\eta$ in §4.

**Worked example** (predicting car price from 3 features, one full round, square loss, shrinkage $\eta = 0.1$):

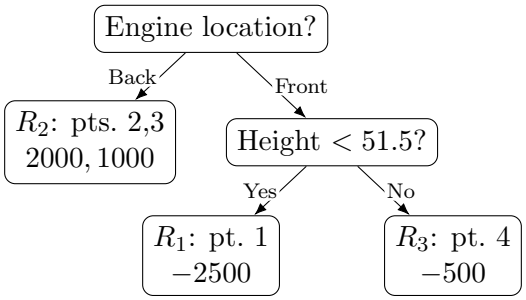
| $i$ | Cylinders | Height | Engine loc. | Price $y_i$ |
|---|---|---|---|---|
| 1 | Four | 48.8 | Front | 12,000 |
| 2 | Six | 48.8 | Back | 16,500 |
| 3 | Five | 52.4 | Back | 15,500 |
| 4 | Four | 54.3 | Front | 14,000 |

**Step 1 (initialize).** $F_0(x) = \arg\min_\gamma \sum_i \frac{1}{2}(y_i - \gamma)^2$; differentiating and setting to 0 gives $\sum_i (y_i - \gamma) = 0$, i.e. $\gamma = \bar{y}$ — confirming §2's claim that the square-loss minimizer is just the mean:

$$F_0(x) = \frac{12{,}000 + 16{,}500 + 15{,}500 + 14{,}000}{4} = 14{,}500 \tag{4}$$

**Step 2 (negative gradient = residual, for square loss).** $-g(x_i) = y_i - F_0(x_i)$: residuals are $-2500,\ 2000,\ 1000,\ -500$.

**Step 3 (fit a tree to the residuals).** Suppose the fitted depth-2 tree partitions the 4 points into three leaves — points 1 alone, points 2 and 3 together, and point 4 alone. Splitting on Engine location first gets points 2,3 (both Back) into one leaf immediately; the remaining two (both Front) still need separating, which a height threshold does:



(Point 1 is FRONT/48.8, point 4 is FRONT/54.3 — the 51.5 threshold, a midpoint between them, is exactly §4's candidate-threshold rule from the decision-tree notes: a split can only ever land between two sorted, distinct training values.) **Step 4 (leaf values $w_k$ = per-leaf mean, for square loss).** $R_1$ and $R_3$ have one point each, so $w_1 = -2500$, $w_3 = -500$ trivially. $R_2$ has two: minimizing $\frac{1}{2}(2000-w)^2 + \frac{1}{2}(1000-w)^2$ over $w$ gives $w_2 = \frac{1}{2}(2000+1000) = 1500$ — the mean, exactly as §2's general recipe says (and, for square loss specifically, exactly what $w^* = -G/H$ from §3 reduces to: $G = \sum g_i = -(2000+1000)$, $H = \sum h_i = 2$ since $\partial^2/\partial F^2 \, \frac{1}{2}(y-F)^2 = 1$ per point, so $w^* = -G/H = 1500$).

**Step 5 (update).** $F_1(x) = F_0(x) + \eta\, h_1(x)$. Point 1 (Four/48.8/Front) lands in $R_1$:

$$F_1(x_1) = 14{,}500 + 0.1 \times (-2500) = 14{,}250 \tag{5}$$

— a small nudge toward point 1's true price (12,000), exactly as much as one shrunken round of one tree should move it; repeat Steps 2–5, refitting a new tree to the new residuals $y_i - F_1(x_i)$ each round, until validation error stops improving.

**Why square loss isn't the whole story.** Square loss is easy to differentiate but *not robust*: because error is squared, one outlier can dominate the total loss.

| $y_i$ | 0.5 | 1.2 | 2 | 5* |
|---|---|---|---|---|
| $F(x_i)$ | 0.6 | 1.4 | 1.5 | 1.7 |
| $L = (y-F)^2/2$ | 0.005 | 0.02 | 0.125 | 5.445 |

If the last point is mislabeled (marked *), it alone contributes $5.445/(0.005+0.02+0.125+5.445) \approx 97\%$ of the total square loss — the algorithm will bend over backwards fitting that one bad point. Two more robust losses and their negative gradients: **Absolute loss:** $L(y,F) = |y-F|$, with negative gradient $-g(x_i) = \text{sign}(y_i - F(x_i))$.
**Huber loss** (threshold $\delta$): quadratic near zero, linear beyond $\delta$ — and its negative gradient, the residual clipped to $\pm\delta$:

$$L(y,F) = \begin{cases} \frac{1}{2}(y-F)^2 & |y-F| \le \delta \\ \delta(|y-F| - \delta/2) & |y-F| > \delta \end{cases} \qquad -g(x_i) = \begin{cases} y_i - F(x_i) & |y_i - F(x_i)| \le \delta \\ \delta\,\text{sign}(y_i - F(x_i)) & |y_i - F(x_i)| > \delta \end{cases} \tag{6}$$

| $y_i$ | 0.5 | 1.2 | 2 | 5* |
|---|---|---|---|---|
| $F(x_i)$ | 0.6 | 1.4 | 1.5 | 1.7 |
| Square loss | 0.005 | 0.02 | 0.125 | 5.445 |
| Absolute loss | 0.1 | 0.2 | 0.5 | 3.3 |
| Huber loss ($\delta = 0.5$) | 0.005 | 0.02 | 0.125 | 1.525 |

**Key point:** In general, *negative gradient ≠ residual* — and you should follow the *gradient*, not the residual. For Huber loss, the residual on the outlier is $5-1.7 = 3.3$, but the negative gradient clips it to $\pm\delta = \pm 0.5$: following the gradient inherits the robust loss's insensitivity to outliers; following the raw residual would not. (Learning-rate/shrinkage choice is deferred to §4; see Friedman 2001 for a detailed treatment.)

This is *exactly* HW1 Problem 3's algorithm (square loss, depth-2 trees, $\rho = 1$, no shrinkage) — the Python is in §4.

# 3 Deriving Boosting for Classification

## Binary classification, log-loss (the clean, modern derivation)

Let $F(x) \in \mathbb{R}$ be a score and $p(x) = \Pr(y = 1 \mid x) = \dfrac{1}{1 + e^{-F(x)}}$ (sigmoid link), $y_i \in \{0, 1\}$. Why go through a score and a link function instead of having the trees predict the probability directly? Because $F(x) = \sum_t \rho_t h_t(x)$ is an unconstrained sum of tree outputs — it can land anywhere on $\mathbb{R}$ — while a probability is stuck in $[0, 1]$; the sigmoid absorbs that constraint only at the very last step, so every intermediate quantity a tree ever has to fit (the score, the gradient, the boosting target) stays an ordinary, unconstrained real number. The per-point log-likelihood is

$$\ell_i(F) = y_i \log p(x_i) + (1 - y_i) \log \big(1 - p(x_i)\big) \tag{7}$$

Differentiate w.r.t. the score $F(x_i)$ (chain rule through the sigmoid; using $1 - p(x_i) = \frac{e^{-F(x_i)}}{1+e^{-F(x_i)}}$ and $\frac{1}{1+e^{F(x_i)}} = \frac{e^{-F(x_i)}}{1+e^{-F(x_i)}}$):

$$\frac{\partial \ell_i}{\partial F(x_i)} = -y_i \cdot \frac{e^{-F(x_i)}}{1 + e^{-F(x_i)}} + (1 - y_i)(-1) + (1 - y_i) \cdot \frac{e^{-F(x_i)}}{1 + e^{-F(x_i)}} \tag{8}$$

$$= \frac{e^{-F(x_i)}}{1 + e^{-F(x_i)}} \big(y_i + (1 - y_i)\big) - (1 - y_i) = \frac{1}{1 + e^{F(x_i)}} - (1 - y_i) \tag{9}$$

$$= \big(1 - p(x_i)\big) - (1 - y_i) = y_i - p(x_i) \tag{10}$$

$$\boxed{\frac{\partial \ell_i}{\partial F(x_i)} = y_i - p(x_i)} \qquad \Rightarrow \qquad \text{boosting target: } -g(x_i) = y_i - p(x_i) \tag{11}$$

Same recipe as §2: fit a regression tree to $\big(x_i,\ y_i - p(x_i)\big)$, update $F := F + \rho h$, recompute $p(x) = \text{sigmoid}(F(x))$, repeat. (This looks like "residual" again — $y_i$ minus a probability instead of $y_i$ minus a real value — but it's the gradient of log-likelihood, not a residual in the regression sense; the resemblance is a coincidence of the sigmoid link, and stops being true for the multiclass case just below.)

## Multiclass, softmax / cross-entropy

Now $L$ classes, one score function $F_k$ per class, softmax link $p_k(x) = \dfrac{e^{F_k(x)}}{\sum_{j=1}^{L} e^{F_j(x)}}$, and per-point loss the cross-entropy $\ell_i(\mathbf{F}) = -\sum_k \mathbb{1}_{\{y_i = k\}} \log p_k(x_i) = -\log p_{y_i}(x_i)$. Two cases for $\partial \ell_i / \partial F_k(x_i)$:

$$k = y_i: \qquad \ell_i = -F_k(x_i) + \log\Big(\sum_j e^{F_j(x_i)}\Big) \qquad \Rightarrow \qquad \frac{\partial \ell_i}{\partial F_k(x_i)} = -1 + p_k(x_i) \tag{12}$$

$$k \neq y_i: \qquad \text{only the softmax denominator contributes} \qquad \Rightarrow \qquad \frac{\partial \ell_i}{\partial F_k(x_i)} = p_k(x_i) \tag{13}$$

$$\boxed{\frac{\partial \ell_i}{\partial F_k(x_i)} = p_k(x_i) - \mathbb{1}_{\{y_i = k\}}} \qquad \Rightarrow \qquad \text{boosting target for class } k: \ -g_k(x_i) = \mathbb{1}_{\{y_i = k\}} - p_k(x_i) \tag{14}$$

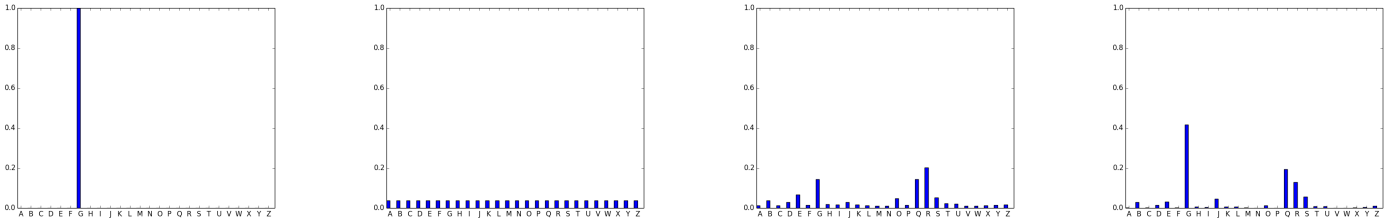
One negative-gradient column per class; fit one tree $h_k$ per class per round to its own column, update every $F_k$ independently:

**Algorithm: Gradient Boosting for Multiclass Classification**

1. Initialize $F_1, \dots, F_L$ (e.g. all zero $\Rightarrow$ uniform $p_k = 1/L$).
2. Iterate $t = 1, \dots, T$: for each class $k = 1, \dots, L$: compute $-g_k(x_i) = \mathbb{1}_{\{y_i = k\}} - p_k(x_i)$; fit tree $h_k$ to it; update $F_k := F_k + \rho_k h_k$.
3. Recompute all $p_k(x) = \mathrm{softmax}_k(F_1(x), \dots, F_L(x))$; predict $\arg\max_k p_k(x)$.

---

**Worked example** (26-class letter recognition, UCI Letter Recognition dataset — one data point, true label G, tracked across boosting rounds):



Left to right: the true distribution $Y$ (all mass on G); the predicted distribution $P$ at round 0 (uniform, $F_k \equiv 0$); after 4 rounds; after 8 rounds. Each round's per-class residual $Y_k - P_k$ shrinks the gap between panel 1 and the current panel.

| round | $P_A$ | $P_E$ | $P_G$ (true) | $P_Q$ | $P_R$ | $P_S$ |
|---|---|---|---|---|---|---|
| 0 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 | 0.04 |
| 4 | 0.01 | 0.06 | 0.15 | 0.14 | 0.20 | $\approx$0 |
| 8 | 0.01 | 0.02 | 0.42 | 0.19 | 0.13 | 0.06 |

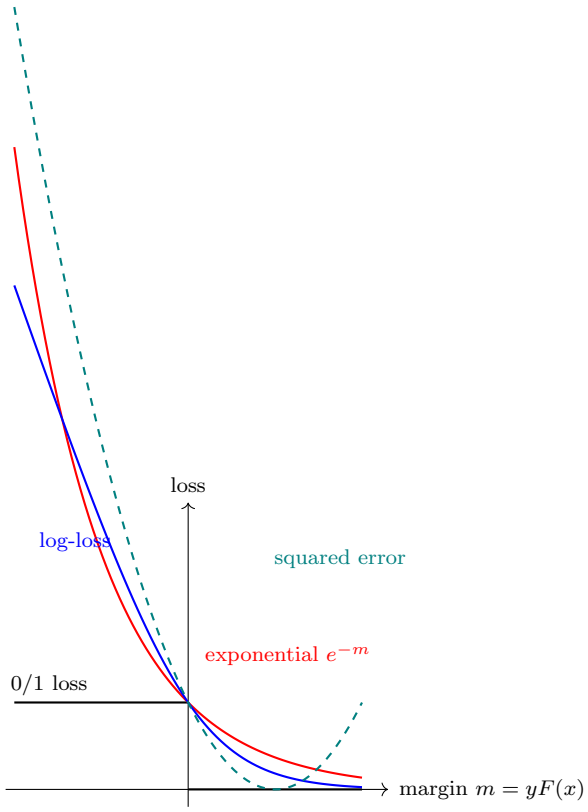
By round 8 the model has (correctly) concentrated most of its mass on G, with Q/R/S as the runners-up (plausible: rounded, all sharing similar 16-feature statistics) — boosting is visibly, monotonically closing the KL-divergence gap one weak learner at a time.

## Aside: the $\{-1, +1\}$ view and the connection to AdaBoost

Everything above used $y \in \{0, 1\}$. The historical treatment (and AdaBoost, §6) uses $y \in \{-1, +1\}$ with a symmetric link $p(x) = \dfrac{e^{F(x)}}{e^{F(x)} + e^{-F(x)}}$, which is algebraically the same model under a change of variables, and gives

$$\Pr(y \mid x) = \frac{1}{1 + e^{-2yF(x)}} \qquad \Rightarrow \qquad -\log \Pr(y \mid x) = \log\left(1 + e^{-2yF(x)}\right) \tag{15}$$

Call $m = yF(x)$ the **margin** (positive $\Leftrightarrow$ correctly, confidently classified). Four losses, all rescaled to pass through $(0, 1)$ for comparison:




- Both exponential loss and log-loss are smooth, convex, monotone-decreasing upper bounds on 0/1 loss, and both are minimized (in population) at the same $F^*(x) = \frac{1}{2}\log\frac{\Pr(y=1|x)}{\Pr(y=-1|x)}$ — **this is the Friedman–Hastie–Tibshirani (2000) result**: AdaBoost's forward stage-wise fitting under exponential loss and gradient boosting's fitting under log-loss are two convex surrogates for the same classification problem, agreeing near the decision boundary ($m \approx 0$) and diverging away from it.

- Squared error, extended to classification via $L = (1-m)^2$, is *not* monotone in the margin — it turns back up for $m > 1$, i.e. it penalizes being *too confidently correct*. A real, known pathology of squared error as a classification loss, visible directly in the plot.

$\Rightarrow$ AdaBoost is not a separate algorithm from gradient boosting — it's the special case you get by taking exponential loss inside the same forward-stage-wise-additive-fitting framework (Breiman 1997–99 first made this connection explicit). We revisit this in §6.

## The Newton-step leaf value

The gradient tells a tree *what shape* to fit; it does not, by itself, say what constant value $w$ each leaf should output (we use $w$ for this, not $\gamma$, to match HW1 Problem 5's and XGBoost's own notation — §4 reserves $\gamma$ for something else entirely: a per-leaf complexity penalty). The obvious choice $w$ = mean of that leaf's targets is a first-order (gradient-only) answer — a second-order (Newton) step uses curvature too and is exactly what modern implementations (and HW1 Problem 5, and LambdaMART in §5) actually use.

For a region (leaf) $R$, Taylor-expand the loss in the constant $w$ added to every point in $R$, around $w = 0$:

$$\sum_{i\in R} L\big(y_i, F(x_i) + w\big) \approx \sum_{i\in R}\Big[L(y_i, F(x_i)) + g_i w + \tfrac{1}{2}h_i w^2\Big], \qquad g_i = \frac{\partial L}{\partial F(x_i)}, \quad h_i = \frac{\partial^2 L}{\partial F(x_i)^2} \tag{16}$$

Dropping the constant term and writing $G = \sum_{i \in R} g_i$, $H = \sum_{i \in R} h_i$, this is a quadratic in $w$: $wG + \frac{1}{2}w^2 H$. Minimizing ($\frac{d}{dw} = G + wH = 0$):

$$\boxed{w^* = -\frac{G}{H} = -\frac{\sum_{i \in R} g_i}{\sum_{i \in R} h_i}} \tag{17}$$

For logistic loss specifically, $g_i = p_i - y_i$ (negative of the log-likelihood gradient derived above) and $h_i = \frac{\partial g_i}{\partial F(x_i)} = p_i(1 - p_i)$ (the sigmoid's own derivative), giving

$$w^* = \frac{\sum_{i \in R}(y_i - p_i)}{\sum_{i \in R} p_i(1 - p_i)} \tag{18}$$

In plain English: $G = \sum g_i$ is the *net direction and size* of how wrong the ensemble currently is on this leaf's points — large $|G|$ means a big, one-sided correction is needed here. $H = \sum h_i$ measures how *curved* the loss is on those points; for logistic loss, $h_i = p_i(1 - p_i)$ peaks at $p_i = 0.5$ (points genuinely near the decision boundary, where a small change in $F$ swings the loss a lot) and vanishes as $p_i \to 0$ or 1 (points where the sigmoid has saturated, so the loss barely responds to $F$ there). Dividing by $H$ is therefore standard Newton behavior: a *smaller* step where the loss is sharply curved ($H$ large, near the boundary — easy to overshoot), a *larger* step where it's nearly flat ($H$ small, saturated). This is not the same as "confidence" by itself, though: a saturated leaf the model already has *right* has both $G \approx 0$ and small $H$, so $w^* = -G/H$ stays small there too — it's the *ratio* $G/H$ that sets the step, not $H$ alone. (This is exactly the adaptive, per-region step size a first-order, gradient-only/leaf-mean update doesn't get.)

**Key point:** This $w^* = -G/H$ formula is the single most-reused piece of math in this document: it's HW1 Problem 5's leaf weight $w^* = -G/(H + \lambda)$ (identical, plus an $L_2$ leaf-weight penalty $\lambda$ — the XGBoost regularization, derived in full in §4), and it's LambdaMART's leaf value in §5 ($\gamma_{km} = \sum y_i / \sum w_i$ — Burges' own paper calls it $\gamma$, but it's the identical ratio). Learn it once here.

# 4 Implementation Details and Examples

**General algorithm** (regression, classification, and — with the right per-leaf target — ranking, all share this shape):

**Algorithm: Gradient Boosting (general loss, with shrinkage)**

1. Initialize $F_0(x) = \arg\min_\gamma \sum_i L(y_i, \gamma)$.
2. For $m = 1, \ldots, M$:
   (a) $-g_i = -\partial L(y_i, F_{m-1}(x_i))/\partial F(x_i)$ for every training point (§2, §3).
   (b) Fit a regression tree to $(x_i, -g_i)$, giving regions $R_1, \ldots, R_K$.
   (c) Per leaf, set $w_k$ by the Newton step $w_k = -G_k/H_k$ (§3), or just the leaf mean of $-g_i$ (first-order — what HW1 Problem 3 does).
   (d) $F_m(x) = F_{m-1}(x) + \eta \sum_k w_k \mathbb{1}(x \in R_k)$, shrinkage $0 < \eta \leq 1$.

---

**Shrinkage** $\eta$ (Friedman, 2001) trades rounds for generalization: smaller $\eta$ needs more rounds $M$ but generalizes better (regularization by not fully trusting any one round) — the direct ensemble-boosting

analogue of the decision-tree notes' pre-pruning depth cap. HW1 Problem 3 uses $\eta = 1$ (no shrinkage, simplest case); HW1 Problem 5 exposes it as a tunable `learning_rate`.

**Remark.** $\eta$ is boosting's version of a *learning rate*, in exactly the sense you'll see it again for gradient descent on a neural network's weights: both are "how far to move in the direction the gradient just pointed," and in both settings too large a value causes overshoot/divergence while too small a value just needs more iterations to get there. The difference is only *where* the step is taken — in weight space for a neural net's backprop update, in function space (one new tree, added to the running sum $F$) here.

**HW1 Problem 3 — first-order (residual) boosting, real code** (reuses the `DecisionTree` class from the decision-tree notes' §13 unchanged, as the weak learner):

```
class BoostedRegressionTrees:
    def fit(self, X, y):
        residual = y.copy()
        for _ in range(self.n_rounds):
            tree = DecisionTree(mode="regression", max_depth=self.tree_depth)
            tree.fit(X, residual)
            residual = residual - tree.predict(X)     # what's left for the next tree to explain
            self.trees.append(tree)

    def predict(self, X):
        return sum(tree.predict(X) for tree in self.trees)   # sum of all rounds' outputs
```

No re-weighting of examples (that's AdaBoost, §6 — a different mechanism entirely), no learning rate (Problem 5 below adds one). Just: fit the negative gradient (here, literally the residual, since it's square loss), add it in, repeat.

**HW1 Problem 5 (optional) — second-order (Newton) boosting, from the XGBoost paper.** Problem 3 above is *first-order*: it only uses the gradient. XGBoost (Chen & Guestrin, 2016, §§2.1–2.3, Problem 5's assigned reading) generalizes this to *second-order*, redoing Problem 3's boosting as log-loss classification, where — unlike square loss, whose Hessian is trivially 1 — the Hessian actually carries information.

*(a) Where the split-scoring gain formula comes from (the regularized objective).* §3's $w^* = -G/H$ Taylor-expanded the *loss* alone. XGBoost adds an explicit model-complexity penalty $\Omega$ on top, and re-derives everything from the *penalized* objective:

$$\text{Obj} = \sum_i L\big(y_i, F(x_i)\big) + \Omega(f), \qquad \Omega(f) = \gamma T + \tfrac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{19}$$

where $f$ is the tree being added this round, $T$ its number of leaves, $w_j$ leaf $j$'s output. Two knobs, two different jobs: $\gamma$ charges a flat cost *per leaf* — exactly $\alpha$'s role in the decision-tree notes' cost-complexity pruning ($R_\alpha(T) = R(T) + \alpha|\widetilde{T}|$, §7 there): a split must earn back at least $\gamma$ to be worth making. $\lambda$ penalizes *large* leaf values ($L_2$/ridge-style — the same idea as HW1 Problem 1's ridge regression, just shrinking leaf outputs instead of linear-model coefficients), independent of how many leaves there are.

Second-order Taylor-expand $L$ (identical step to §3), then group points by leaf $j$ (writing $G_j = \sum_{i \in j} g_i$, $H_j = \sum_{i \in j} h_i$, $f(x_i) = w_j$ for $x_i$ in leaf $j$): this collapses to $T$ independent quadratics in $w_j$,

$$\text{Obj} \approx \sum_{j=1}^{T} \Big[ G_j w_j + \tfrac{1}{2}(H_j + \lambda) w_j^2 \Big] + \gamma T, \tag{20}$$

minimized leaf-by-leaf exactly as in §3 (now with $\lambda$ folded into the curvature term):

$$w_j^* = -\frac{G_j}{H_j + \lambda} \qquad \text{(Problem 5's leaf weight, no longer just asserted — derived)} \tag{21}$$

Plugging $w_j^*$ back in scores *any fixed partition into leaves* — lower is better:

$$\text{Obj}^* = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda} + \gamma T \tag{22}$$

A candidate split turns one leaf $(G, H)$ into two ($G_L, H_L$ and $G_R, H_R$, with $G{=}G_L{+}G_R$, $H{=}H_L{+}H_R$); its **gain** is how much this score *improves* — one fewer leaf's $-G^2/(H+\lambda)$ term, one more $\gamma$ charged:

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}\right] - \gamma \tag{23}$$

— the textbook (Chen & Guestrin's own) version. Reject every candidate split with $Gain \le 0$ (make a leaf instead): at $\gamma = 0$ this is §2's ordinary "no split helps" rule; at $\gamma > 0$ it's a per-leaf price, baked directly into the split search rather than pruned away afterward.

**Remark.** HW1's own assignment text and the code below *drop the leading* $\frac{1}{2}$:

```
gain = G_L^2/(H_L+lambda) + G_R^2/(H_R+lambda) - G^2/(H+lambda) - gamma
```

This is a harmless rescaling, not an error — a split search only cares which candidate has the *largest* gain, and multiplying every candidate's score by the same positive constant (2) never changes that ranking. The one place it matters is the accept/reject threshold: HW1's $\gamma$ is effectively *twice* the textbook $\gamma$ above for the same accept/reject behavior. Either convention is fine as long as you're consistent within one run — but if you're ever comparing your fitted $\gamma$ against a value quoted from the XGBoost paper directly, remember the factor of 2.

**Key point:** $\Omega(f) = \gamma T + \frac{1}{2}\lambda\sum w_j^2$ is the decision-tree notes' cost-complexity pruning *plus* ridge-style leaf shrinkage, fused into the per-round objective instead of run as a separate post-hoc pass: $\gamma \leftrightarrow \alpha$ (price per leaf); $\lambda$ has no pruning analogue — it's the $L_2$ idea from HW1 Problem 1, now regularizing leaf outputs instead of regression coefficients.

*(b) Real code* (each tree's *splits* are scored by the structure gain built from $g, h$ — HW1's un-halved convention, matching `_grow` below — and each *leaf* gets the $w^* = -G/H$ formula from §3, with an added ridge term $\lambda$):

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

class XGBoostStyleTree:                       # splits scored by structure gain, not entropy/variance
    def _leaf_weight(self, g, h):
        return -g.sum() / (h.sum() + self.lmbda)             # w* = -G/(H+lambda), Sec. "Newton step"
    # ... _grow() picks the split maximizing
    #     G_L^2/(H_L+lambda) + G_R^2/(H_R+lambda) - G^2/(H+lambda) - gamma

class SecondOrderBoostedClassifier:
    def fit(self, X, y):
        self.trees = []
        F = np.zeros(X.shape[0])
        for _ in range(self.n_rounds):
            p = sigmoid(F)
            g = p - y                          # log-loss gradient  (Sec. "classification", boxed result)
            h = p * (1 - p)                    # log-loss Hessian   (sigmoid's own derivative)
            tree = XGBoostStyleTree(max_depth=self.tree_depth, lmbda=self.lmbda, gamma=self.gamma)
            tree.fit(X, g, h)
            self.trees.append(tree)
            F = F + self.learning_rate * tree.predict(X)      # Newton step, scaled by shrinkage
        return self
```

Line-by-line: `g = p - y` is §3's boxed gradient with a sign flip ($g = -$(boosting target), since `XGBoostStyleTree` takes the raw loss gradient, not the negative); `h = p*(1-p)` is exactly the Hessian used to derive $w^*$ in §3; `_leaf_weight` *is* that boxed formula, verbatim, with $\lambda$ added.

**Common bugs** (all silent):

- **Refitting instead of accumulating**: each round's tree must be fit to the *current* residual/gradient (after all previous rounds' updates), and `predict` must *sum every round*, not just return the last tree's output — an easy copy-paste error when adapting single-tree `fit`/`predict`.
- **Learning rate too large**: with $\eta$ close to 1 and many rounds, both regression and classification boosting can *diverge* (overshoot, oscillate) rather than converge — if train error goes *up* between rounds, suspect $\eta$ first.
- **Sign errors between "gradient" and "negative gradient"**: §3's boxed results are for the *log-likelihood* (something to *maximize*); the *loss* is its negative, so the *gradient of the loss* used in `XGBoostStyleTree` above is $g = p - y$, while the *boosting target* (negative gradient of the loss = gradient of the log-likelihood) is $y - p$ — both correct, opposite signs, easy to mix up mid-implementation.
- **Sanity check**: fit `sklearn.ensemble.GradientBoostingRegressor/Classifier` with matching `max_depth`, `n_estimators`, `learning_rate` on the same data (HW1 already asks for this library baseline) — train curves should track closely; a large, systematic gap points at a bug above, not just "different implementation." For Problem 5 specifically, HW1 asks you to compare against the real `xgboost.XGBClassifier` (matching `max_depth`, `reg_lambda`= $\lambda$, `gamma`, `learning_rate`) — since it's optimizing the exact objective just derived above, train/test log-loss curves should track closely, round for round; if yours converges much slower with identical hyperparameters, suspect a sign error in $g, h$ (previous bullet) before suspecting the library.

**Connection to $k$-NN (HW1 Problem 8).** Trees and $k$-NN (HW1 Problem 4) are both *local, nonparametric* predictors — neither fits one global functional form; both predict from a *neighborhood* of similar training points. The difference is entirely in how that neighborhood is defined:

- $k$-NN's neighborhood is the $k$ literal nearest points under a *fixed, chosen* distance or kernel (Euclidean, cosine, RBF, polynomial — Problem 4's exact menu), computed the same way at every query point, with no reference to the labels at all.
- A tree's "neighborhood" for a point $x$ is the leaf $x$ lands in — an axis-aligned region *learned* by the greedy, impurity-driven split search (entropy/variance, decision-tree notes §3), i.e. shaped by where the training labels actually live, not by a distance formula fixed in advance.

Problem 8's hypothesis connects these two very different mechanisms: a test point sitting in a region where its literal nearest neighbors overwhelmingly agree (label-pure, or low target-spread) should *also* tend to fall on the "easy" side of most of the ensemble's $T$ trees — their individually-shallow votes reinforce instead of partially cancelling, giving a large-magnitude, confident $F(x)$. In a genuinely mixed/sparse neighborhood, different rounds' trees split $x$ inconsistently, their contributions partially cancel, and $F(x)$ stays close to the "unsure" region ($p(x) \approx 0.5$, or the top two softmax probabilities close together). Concretely, using this document's own notation:

- **Boosting confidence** (Spambase, binary): $|p(x) - 0.5| \times 2 \in [0, 1]$ — the same sigmoid $p(x)$ from §3, just rescaled so 0 = maximally unsure, 1 = maximally confident.
- **Boosting confidence** (MNIST, multiclass): $p_{(1)}(x) - p_{(2)}(x)$, the gap between the top two softmax probabilities from §3's multiclass derivation — this is *exactly* the binary margin above when there are only 2 classes, not merely an analogous quantity.
- **Boosting uncertainty** (Housing, regression): no decision boundary to be far from here, so use the std of predictions across several bootstrap refits of the same `BoostedRegressionTrees` pipeline instead — small std = confident, large std = unsure.
- **$k$-NN neighborhood consistency**: `knn_purity` (majority-class fraction among the $k$ nearest neighbors, classification) or `knn_target_spread` (std of $y$ among the $k$ nearest neighbors, regression) — computed with the *same* `pairwise_distances` call as Problem 4, and completely independent of the boosted model: a purely data-side measure with no reference to $F$ at all.

Problem 8 asks you to correlate (Pearson, Spearman) these two independently-computed quantities across a few choices of $k$. If the hypothesis holds, a signal computed with *zero reference to the fitted model* (raw-feature-space proximity) should predict where a completely different mechanism (a greedily-partitioned, gradient-fit tree ensemble) happens to be confident or unsure — evidence that both are, underneath their very different machinery, picking up on the same underlying local structure of the data.

**Remark.** A deeper, practical version of the same idea: since a leaf *is* a data-driven neighborhood, **leaf co-membership across many trees can itself be used as a learned similarity function** — two points are "similar" if they land in the same leaf in most of the ensemble's $T$ trees, in exactly the way $k$-NN would call two points similar if they're close in Euclidean distance. This isn't hypothetical: it's a standard technique (e.g. Facebook's GBDT+LR pipeline, and `sklearn`'s tree `.apply()` method exists precisely for this) to turn a fitted boosted-tree ensemble into a learned embedding/kernel, usable for exactly the same nearest-neighbor-style tasks HW1 Problem 4 does with a fixed metric — except now the "distance" has been shaped by the labels, not chosen by hand in advance.

# 5 LambdaMART and Learning to Rank

**The ranking problem is different from classification/regression**: given a query and a set of candidate documents with relevance labels, we don't need to predict the labels accurately — we need to *order* the documents so the most relevant ones come first. The natural evaluation metric, **NDCG@k** (Normalized Discounted Cumulative Gain), rewards relevant documents more the higher they're ranked:

$$\text{NDCG@}k = \frac{1}{N} \sum_{j=1}^{k} g(r_j)\, d(j), \qquad g(r) = 2^r - 1 \text{ (gain)}, \quad d(j) = \frac{1}{\log_2(1+j)} \text{ (position discount)} \tag{24}$$

where $r_j$ is the relevance of the document at rank $j$, and $N$ (the ideal DCG) normalizes so a perfect ranking scores 1. (A legacy discount variant used by some eval scripts is piecewise: $d(j) = 1$ for $j = 1, 2$, else $1/\log_2 j$ — scores under the two discounts aren't comparable.)

**MART** (Multiple Additive Regression Trees) is just §4's general gradient-boosting-with-trees algorithm, under its name in the ranking literature — same loop (negative gradient → fit tree → Newton leaf value → shrink-and-add), no change.

**RankNet** turns pairs of documents into a classification problem: for a pair $(i, j)$ where $i$ should rank above $j$, model $P_{ij} = \Pr(i \succ j) = \dfrac{1}{1 + e^{-\sigma(s_i - s_j)}}$, $s_i = F(x_i)$ the current score, and minimize cross-entropy $C = -\bar{P}_{ij} \log P_{ij} - (1 - \bar{P}_{ij}) \log(1 - P_{ij})$ against the true preference $\bar{P}_{ij}$. The gradient w.r.t. the scores factors through a single pairwise quantity $\lambda_{ij} = \partial C / \partial s_i$, and each document's total pull is the sum over all pairs it appears in:

$$\lambda_i = \sum_{j:\{i,j\}\in\text{pairs}} \lambda_{ij} \; - \sum_{k:\{k,i\}\in\text{pairs}} \lambda_{ki} \tag{25}$$

**Problem**: this $\lambda_{ij}$ only cares about pairwise order, not *where* in the ranking the pair sits — but NDCG cares a lot more about the top of the list than the bottom.

**LambdaRank** fixes this by directly weighting $\lambda_{ij}$ by how much swapping the pair would change NDCG:

$$\lambda_{ij} = \frac{-\sigma}{1 + e^{\sigma(s_i - s_j)}} \cdot |\Delta\text{NDCG}_{ij}|, \qquad |\Delta\text{NDCG}_{ij}| = \big|\text{NDCG after swapping ranks of } i, j \; - \; \text{NDCG before}\big| \tag{26}$$

This is exactly the missing ingredient: a pairwise gradient that's *aware of position* in the final ranking, without needing NDCG itself to be differentiable (it isn't — it's a step function of rank order; $\lambda$ sidesteps this by using swap-*deltas* directly as if they were gradients).

**Remark.** Burges's own metaphor for $\lambda_{ij}$ (worth keeping): picture a **spring** connecting the two documents' scores, pulling the more-relevant one up and the less-relevant one down. $|\Delta NDCG_{ij}|$ is the spring's *stiffness* — a misordered pair near the top of the ranking (where users actually look) gets a stiff spring and a strong correction; a misordered pair buried at rank 40 gets a slack spring and is barely tugged at all. The worked example below computes one such spring's pull, by hand, on a concrete 4-document toy ranking.

**LambdaMART** = MART, with the per-example target and Newton weight swapped for $\lambda$-gradients (Burges, 2010; algorithm as in Schamoni's slides):

**Algorithm: LambdaMART**

---

1. Initialize $F_0(x_i) = 0$ (or a base model's score).
2. For $m = 1, \ldots, M$: for each document $i$: compute $y_i = \lambda_i$ (pseudo-target) and $w_i = \partial\lambda_i / \partial F_{m-1}(x_i)$ (Newton weight, the $\lambda$ analogue of a Hessian).
3. Fit a $K$-leaf regression tree on $\{(x_i, y_i)\}$, giving regions $R_{1m}, \ldots, R_{Km}$.
4. Leaf value (*exactly* §3's $w^* = -G/H$ ratio, sign folded into the definition of $y_i, w_i$ above — note Burges' paper swaps the letters: *leaf output* is $\gamma_{km}$ here, and $w_i$ is the *per-point* Newton weight): $\gamma_{km} = \dfrac{\sum_{x_i \in R_{km}} y_i}{\sum_{x_i \in R_{km}} w_i}$.
5. $F_m(x_i) = F_{m-1}(x_i) + \eta \sum_k \gamma_{km} \mathbb{1}(x_i \in R_{km})$.

---

**Worked example** (one query, 4 documents, 2 relevance grades $r \in \{0, 1\}$, current scores giving the wrong order):

| document | relevance $r$ | current score $s$ | current rank |
|---|---|---|---|
| $d_2$ | 0 | 0.9 | 1 |
| $d_1$ | 1 | 0.6 | 2 |
| $d_4$ | 0 | 0.3 | 3 |
| $d_3$ | 1 | $-0.2$ | 4 |

Gains $g(1) = 2^1-1 = 1$, $g(0) = 0$; discounts $d(1) = 1,\ d(2) = \frac{1}{\log_2 3}$=0.631, $\ d(3) = \frac{1}{2},\ d(4) = \frac{1}{\log_2 5}$=0.431.

$$DCG_{\text{current}} = 0(1) + 1(0.631) + 0(0.5) + 1(0.431) = 1.062 \tag{27}$$

$$IDCG \text{ (ideal order } d_1, d_3, d_2, d_4) = 1(1) + 1(0.631) = 1.631 \ \Rightarrow\ NDCG_{\text{current}} = 1.062/1.631 = 0.651 \tag{28}$$

The crucial misordered pair is $(d_1, d_2)$: $d_1$ is relevant but outranked by irrelevant $d_2$. Swapping just that pair (new order $d_1, d_2, d_4, d_3$):

$$DCG_{\text{swap}} = 1(1) + 0(0.631) + 0(0.5) + 1(0.431) = 1.431 \ \Rightarrow\ NDCG_{\text{swap}} = 1.431/1.631 = 0.877 \tag{29}$$

$$|\Delta NDCG_{12}| = |0.877 - 0.651| = 0.226 \tag{30}$$

With $\sigma = 1$: $\lambda_{d_1,d_2} = \dfrac{-1}{1 + e^{(0.6-0.9)}} \cdot 0.226 = \dfrac{-1}{1 + 0.741} \cdot 0.226 = -0.574 \cdot 0.226 \approx \mathbf{-0.130}$. Negative $\lambda_{d_1,d_2}$ pulls $d_1$'s score *up*; by antisymmetry $\lambda_{d_2,d_1} = +0.130$ pulls $d_2$'s score *down* — the tree-fitting machinery will nudge exactly this misordered pair toward the correct order, weighted by how much it actually matters to NDCG (a swap deep in the list, with small $|\Delta NDCG|$, would get a much smaller push).

**Practical extensions** (JForests, Ganjisaffar et al. 2011): plain gradient boosting already has **shrinkage** $\eta$ (§4); LambdaMART implementations commonly add two more stochastic regularizers, both 1.0 = off: **sub-sampling** (fraction of training queries used per round — Friedman's stochastic gradient boosting) and **feature sampling** (best split from a random feature subset — the random-forest idea, grafted on). **Bagged LambdaMART**: train several LambdaMART models independently on different subsamples (embarrassingly parallel) and combine by **Borda count** (rank-based voting) rather than averaging raw scores, since different sub-models' score scales aren't comparable. On MQ2007, bagging raised NDCG@1 from 0.4137 to 0.4200 — beating the best previously-reported RankBoost result of 0.4134 (§7) by a small but real margin.

# 6 AdaBoost (Historical Perspective — Superseded by Gradient Boosting)

AdaBoost (Freund & Schapire, 1996/97) is the **first successful boosting algorithm**, and the one that started the field — foundational, but (per §3's aside and the key point below) a special case of the more general gradient-boosting framework, and rarely the right tool to reach for today.

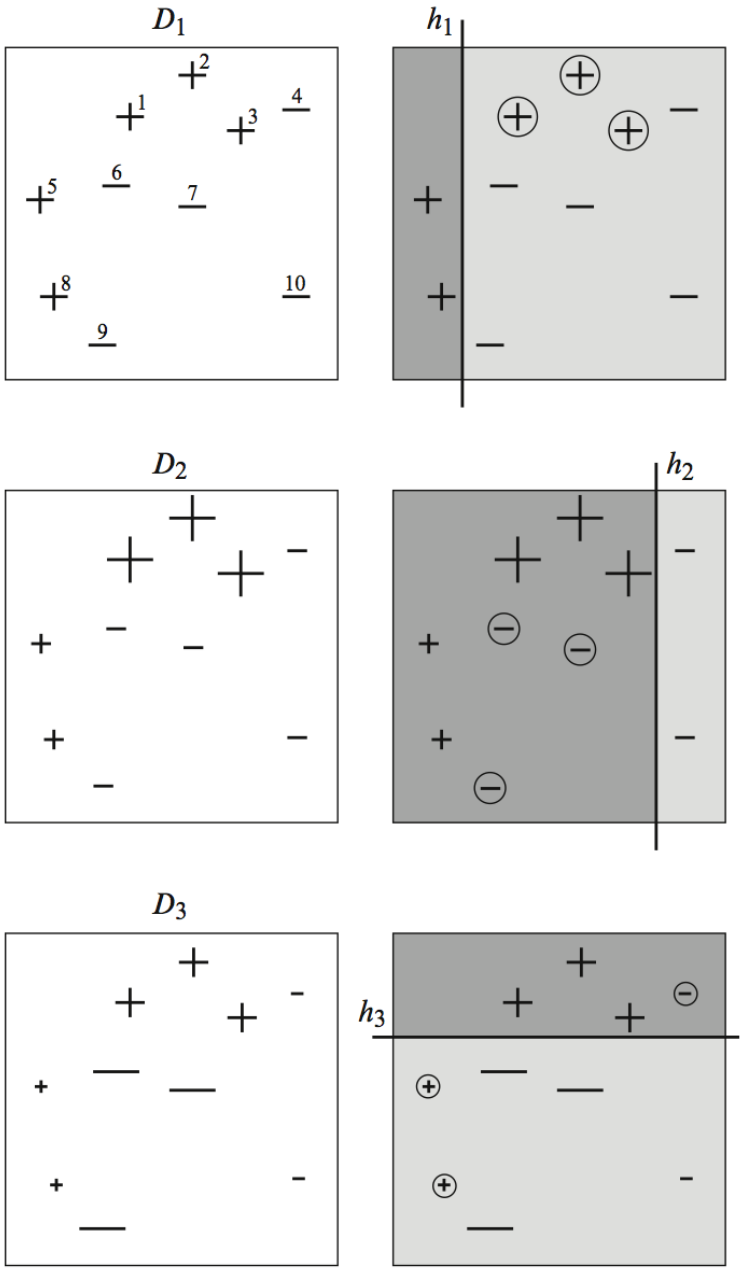
**Intuition.** A committee of weak learners can be accurate even if each member is only slightly better than a coin flip, *provided* members disagree usefully. AdaBoost engineers that disagreement by maintaining a weight distribution $D_t$ over training points — think of $D_t(i)$ as literally how much *attention* round $t$'s weak learner is asked to pay to point $i$: points the current ensemble gets wrong get *more* weight/attention, so the next weak learner is pushed to focus on exactly the current ensemble's blind spots. Each weak learner also earns a confidence vote $\alpha_t$ based on its own weighted accuracy — a committee member who's right more often (on the task it was actually given) gets listened to more.

**Algorithm: AdaBoost**

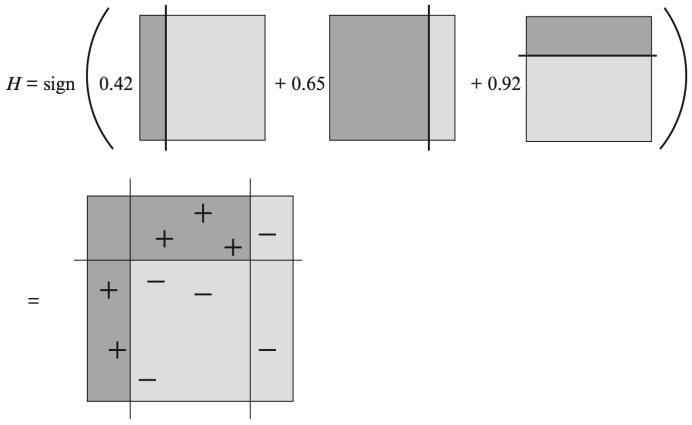
1. Given $(x_1, y_1), \ldots, (x_m, y_m)$, $y_i \in \{-1, +1\}$. Initialize $D_1(i) = 1/m$.
2. For $t = 1, \ldots, T$: train weak learner on $D_t$, get $h_t : \mathcal{X} \to \{-1, +1\}$ minimizing weighted error $\epsilon_t = \Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$.
3. Set $\alpha_t = \frac{1}{2} \ln \dfrac{1 - \epsilon_t}{\epsilon_t}$ (large when $\epsilon_t$ small — confident, accurate learners get a louder vote).
4. Update $D_{t+1}(i) = \dfrac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$ ($Z_t$ = normalizer): weight *shrinks* on points $h_t$ got right, *grows* on points it got wrong.
5. Output $H(x) = \operatorname{sign}\Big( \sum_t \alpha_t h_t(x) \Big)$.

---

**Worked example** (10 points, weak learners = vertical/horizontal lines; no single line separates all 10, but three rounds do):




Round 1: $h_1$ (a vertical split) misclassifies 3 points (circled) $\Rightarrow$ their weight grows for round 2. Round 2: $h_2$ fixes those but misses 3 others (lower weight). Round 3: $h_3$ misses only 3 very-low-weight points.



The weighted vote $H = \text{sign}(0.42h_1 + 0.65h_2 + 0.92h_3)$ classifies all 10 points correctly — three individually-imperfect linear splits combine into a perfect (for this toy set), genuinely nonlinear decision boundary.

**Training-error bound.** Let $\gamma_t = \frac{1}{2} - \epsilon_t$ (the "edge" over random guessing). Then

$$\Pr_{i \sim D_1}[H(x_i) \neq y_i] \;\leq\; \prod_{t=1}^{T} \sqrt{1 - 4\gamma_t^2} \;\leq\; \exp\left(-2\sum_{t=1}^{T} \gamma_t^2\right) \tag{31}$$

*Proof.* Let $F(x) = \sum_t \alpha_t h_t(x)$. Unrolling the update rule,

$$D_{T+1}(i) = D_1(i) \prod_{t=1}^{T} \frac{e^{-y_i \alpha_t h_t(x_i)}}{Z_t} = \frac{D_1(i) \exp(-y_i F(x_i))}{\prod_t Z_t} \tag{32}$$

Since $H(x_i) \neq y_i \Rightarrow y_i F(x_i) \leq 0 \Rightarrow e^{-y_i F(x_i)} \geq 1$, the indicator is bounded by the exponential, so summing over $i$:

$$\Pr_{i \sim D_1}[H(x_i) \neq y_i] = \sum_i D_1(i) \mathbb{1}[H(x_i) \neq y_i] \leq \sum_i D_1(i) e^{-y_i F(x_i)} = \prod_{t=1}^{T} Z_t \tag{33}$$

And each $Z_t = e^{-\alpha_t}(1-\epsilon_t) + e^{\alpha_t}\epsilon_t$, minimized exactly at the chosen $\alpha_t = \frac{1}{2}\ln\frac{1-\epsilon_t}{\epsilon_t}$, giving $Z_t = \sqrt{1-4\gamma_t^2} \leq e^{-2\gamma_t^2}$. Chaining over $t$ gives the bound. ■

⇒ As long as every weak learner beats random guessing by *any* fixed margin ($\gamma_t \geq \gamma > 0$), training error $\to 0$ exponentially fast in $T$. (This margin — how confidently, not just how often, points are classified correctly — is also the intuition behind why AdaBoost tends to keep generalizing even after training error hits zero, and is the same word, with a kindred idea, as the margin you'll maximize directly and explicitly when we cover SVMs.)

**Key point:** Why "obsolete now," precisely: §3's aside showed AdaBoost is forward-stagewise fitting under *exponential* loss — one special case inside the general gradient-boosting framework (Breiman 1997–99; Friedman, Hastie & Tibshirani 2000). Exponential loss grows *unboundedly* on confidently-wrong points, making AdaBoost noticeably more sensitive to label noise and outliers than log-loss-based boosting (FHT00's own finding, motivating their "LogitBoost"). Modern gradient boosting keeps AdaBoost's forward-stagewise idea but generalizes it to any differentiable loss, adds Newton (second-order) leaf values (§3) and shrinkage (§4), and is what HW1, XGBoost, LightGBM, and LambdaMART all actually build on. AdaBoost's proof technique, though, is not obsolete at all — RankBoost's Theorem 1 (§7, next) is the identical telescoping-$Z_t$ argument, one abstraction level up.

# 7 RankBoost

RankBoost (Freund, Iyer, Schapire & Singer, 2003) belongs here, right after AdaBoost, because that is *exactly* what it is: **AdaBoost run on pairs of documents instead of single points.** Same reweighting idea (§6), same update-rule shape, same $\alpha$ formula — the only change is that "training example" now means an ordered pair $(x_0, x_1)$ instead of a single $x$, and "correct" means "ranked in the right relative order" instead of "labeled correctly."

A feedback function $\Phi(x_0, x_1) > 0$ means "$x_1$ should be ranked above $x_0$"; build a distribution over crucial pairs, $D(x_0, x_1) \propto \max\{0, \Phi(x_0, x_1)\}$ — the pairwise analogue of AdaBoost's $D_t(i)$: how much attention the next weak ranker should pay to *getting this particular pair's relative order right*. The **ranking loss** being minimized:

$$rloss_D(H) = \sum_{x_0, x_1} D(x_0, x_1) \mathbb{1}[H(x_1) \le H(x_0)] = \Pr_{(x_0, x_1) \sim D}[H(x_1) \le H(x_0)] \tag{34}$$

**Algorithm: RankBoost**

1. $D_1 = D$ (given initial pairwise distribution).
2. For $t = 1, \ldots, T$: train a weak ranker $h_t : \mathcal{X} \to \mathbb{R}$ on $D_t$; choose $\alpha_t \in \mathbb{R}$; update

$$D_{t+1}(x_0, x_1) = \frac{D_t(x_0, x_1) \exp\big(\alpha_t (h_t(x_0) - h_t(x_1))\big)}{Z_t} \tag{35}$$

3. Output $H(x) = \sum_t \alpha_t h_t(x)$.

---

**Theorem 1** (Freund et al.): $rloss_D(H) \le \prod_t Z_t$. *Proof idea* (structurally identical to AdaBoost's training-error bound, §6): unroll $D_{T+1}$ as a telescoping product of the per-round update, then bound the ranking-loss indicator $\mathbb{1}[H(x_1) \le H(x_0)]$ by the exponential $\exp(H(x_0) - H(x_1))$ that the $D_{T+1}$ expression already contains — same proof skeleton, pairs instead of points.

**Choosing** $\alpha_t$ (for weak rankers with range $[0, 1]$): let $r = \sum_{x_0, x_1} D(x_0, x_1)(h(x_1) - h(x_0))$; then $\alpha = \frac{1}{2} \ln \frac{1 + r}{1 - r}$ gives $Z \le \sqrt{1 - r^2}$ — the direct pairwise analogue of AdaBoost's $\alpha_t = \frac{1}{2} \ln \frac{1 - \epsilon_t}{\epsilon_t}$: maximize $|r|$ (equivalently, minimize the weighted pairwise error) when choosing $h_t$, then set $\alpha_t$ by this closed form.

**Key point:** RankBoost predates LambdaMART by several years and was, for a while, a leading ranking algorithm (the 0.4134 NDCG@1 result cited in §5 is RankBoost's). It has largely been displaced in practice by LambdaMART/GBDT-based rankers, for the same reason AdaBoost has been displaced by gradient boosting generally (§6): reweighting is a special, less flexible case of the gradient view, which generalizes more easily to new metrics (LambdaRank's $|\Delta NDCG|$ trick has no clean RankBoost analogue) and to the same tree-based, Newton-stepped machinery used everywhere else in this document.