

Models as functions

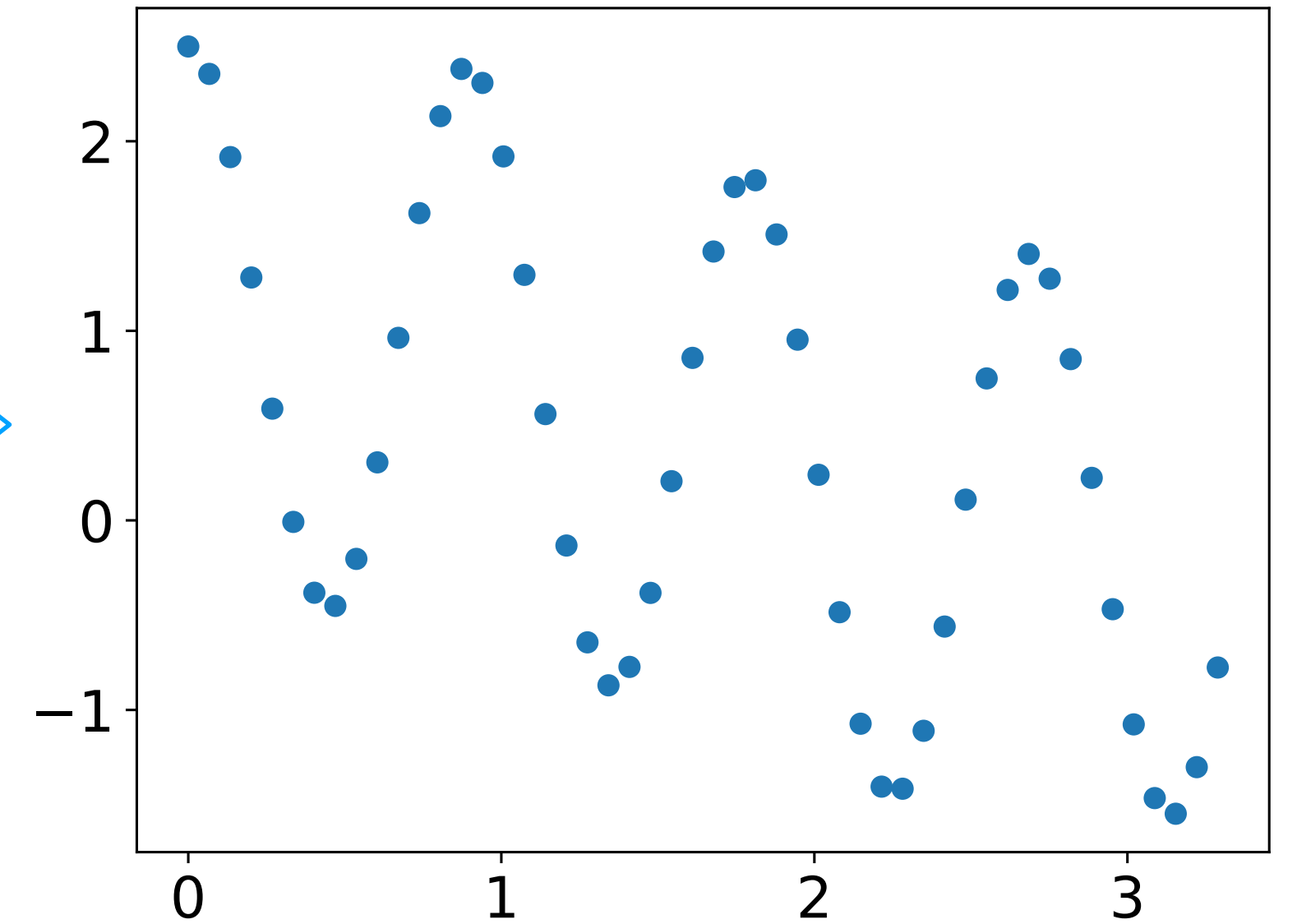
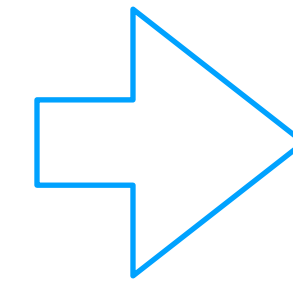
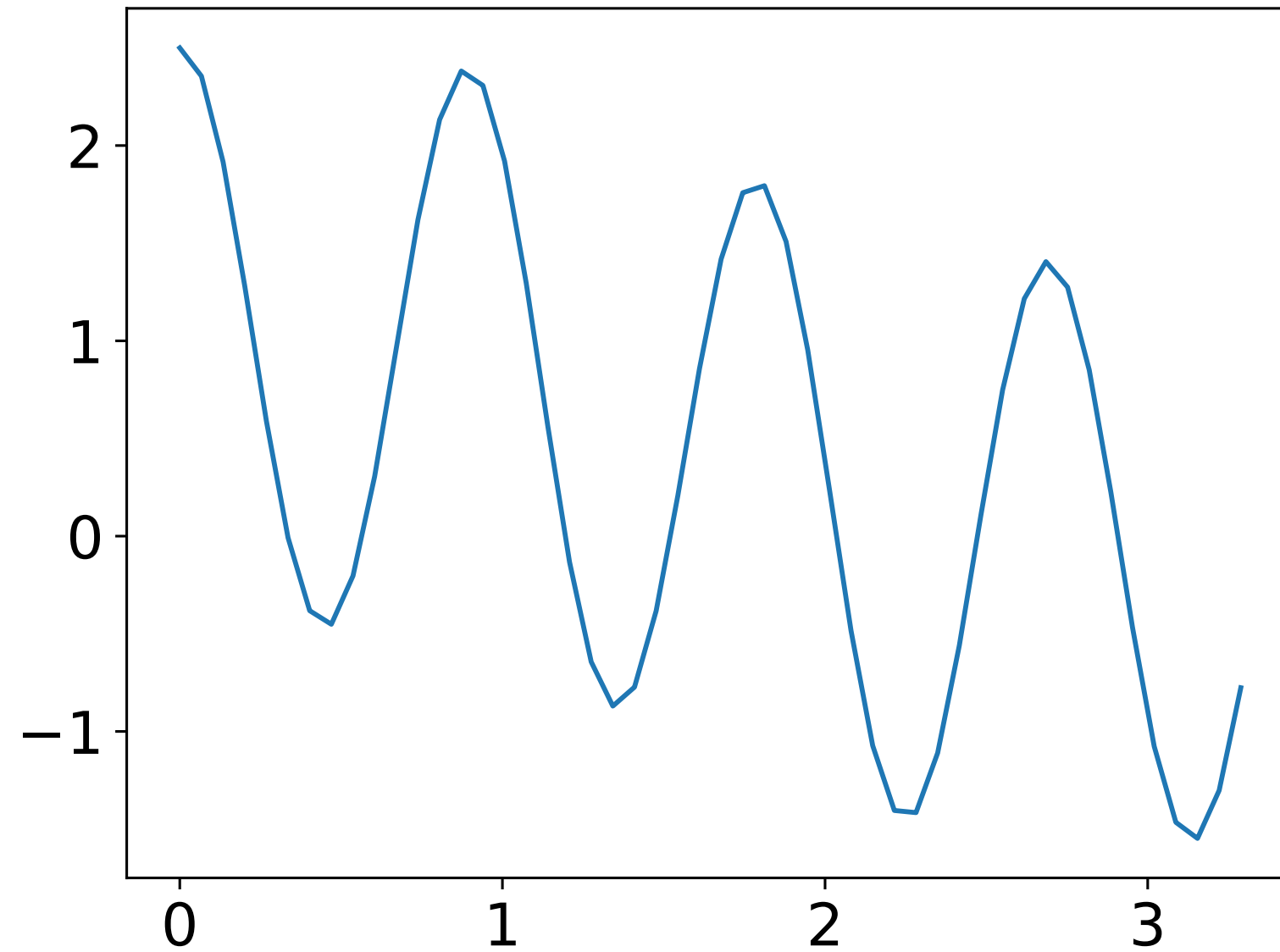


*10-701 Introduction to Machine Learning
Geoff Gordon and Pradeep Ravikumar*

Models as functions

- Consider learning a model $y = f_{\theta}(x)$
- Can think of it two ways:
 - ▶ pick the best $\theta \in \Theta$ — e.g., run SGD on $L(\theta)$
 - ▶ pick the best $f \in \mathcal{F} = \{f_{\theta} \mid \theta \in \Theta\}$
 - ▶ never talk about θ
 - ▶ have to define $L(f)$
- Why?
 - ▶ parameter independence: if we define the same (or similar) \mathcal{F} with different parameterizations, we get the same (or similar) behavior
 - ▶ understanding: θ isn't meaningful, f is what we care about

Functions are vectors



can help to think of f as if it were the sampled version — “components” are values $f(x)$, like components x_i of x , and “indices” are arguments x

- Just like we think of parameter space Θ as a subset of a vector space like \mathbb{R}^d , it helps to think of hypothesis space \mathcal{F} as a subset of a vector space \mathcal{H}
 - ▶ each individual function f is a vector
 - ▶ i.e., satisfies the “vector space API”: we can add $f + g$, scalar multiply $3f$, result is a vector

Vectors everywhere

- Defining a vector by its API is the ***coordinate-free*** or ***abstract*** view
- Lets us use our vector-manipulating skills in new places
- Other examples (that we won't use now):
 - ▶ matrices, tensors
 - ▶ differential operators: $\frac{d}{dx} + 3\frac{d}{dy}$
 - ▶ \mathbb{Q}^d (with scalars from \mathbb{Q}), \mathbb{R}^d
 - ▶ polynomials
 - ▶ ...

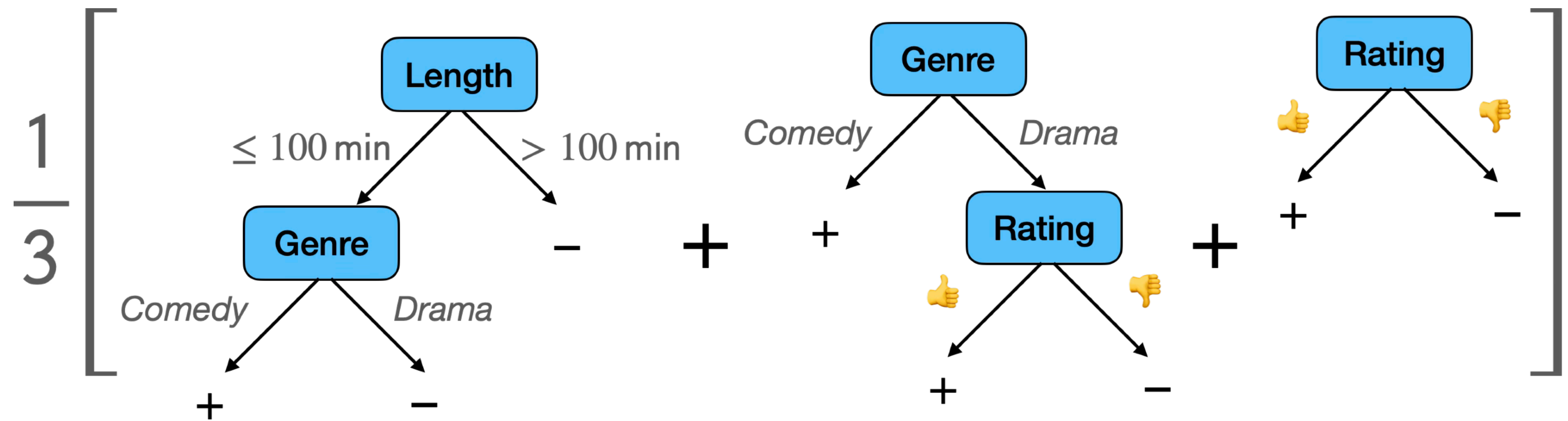
Staying sane: finite- dimensional version

- Useful to connect back to old-fashioned linear-in-parameters functions $\mathcal{F} = \{f_\theta(x)\} = \{\theta \cdot \phi(x)\}$
 - ▶ parameter vector $\theta \in \mathbb{R}^d$
 - ▶ feature vector $\phi(x) \in \mathbb{R}^d$
 - ▶ feature function ϕ is fixed, may be nonlinear
 - ▶ but f_θ is linear in θ
- Won't get the full power of functional ML in this case, but it helps to understand what's going on

Storing a function

- Problem:
 - ▶ when we store a vector x , we do so by storing each component x_i
 - ▶ we can't afford to do this for the infinitely many components of f — its values $f(x)$
- Solution:
 - ▶ start from named functions like $\sin(x)$, e^x , or $T_i^d(x)$ — can even use $f_\theta(x)$, i.e., functions named by their parameters
 - ▶ construct other functions w/ vector space API:
$$\sin(x) + 3e^x - 2T_1^5(x) \quad \leftarrow \text{this is the } \mathbf{span} \text{ and original set is } \mathbf{basis}$$
 - ▶ all such representations are finite: store as a list of coefficients \times named fns
 - ▶ although long lists can get slow to work with!

Example of storing a complex function: bagging



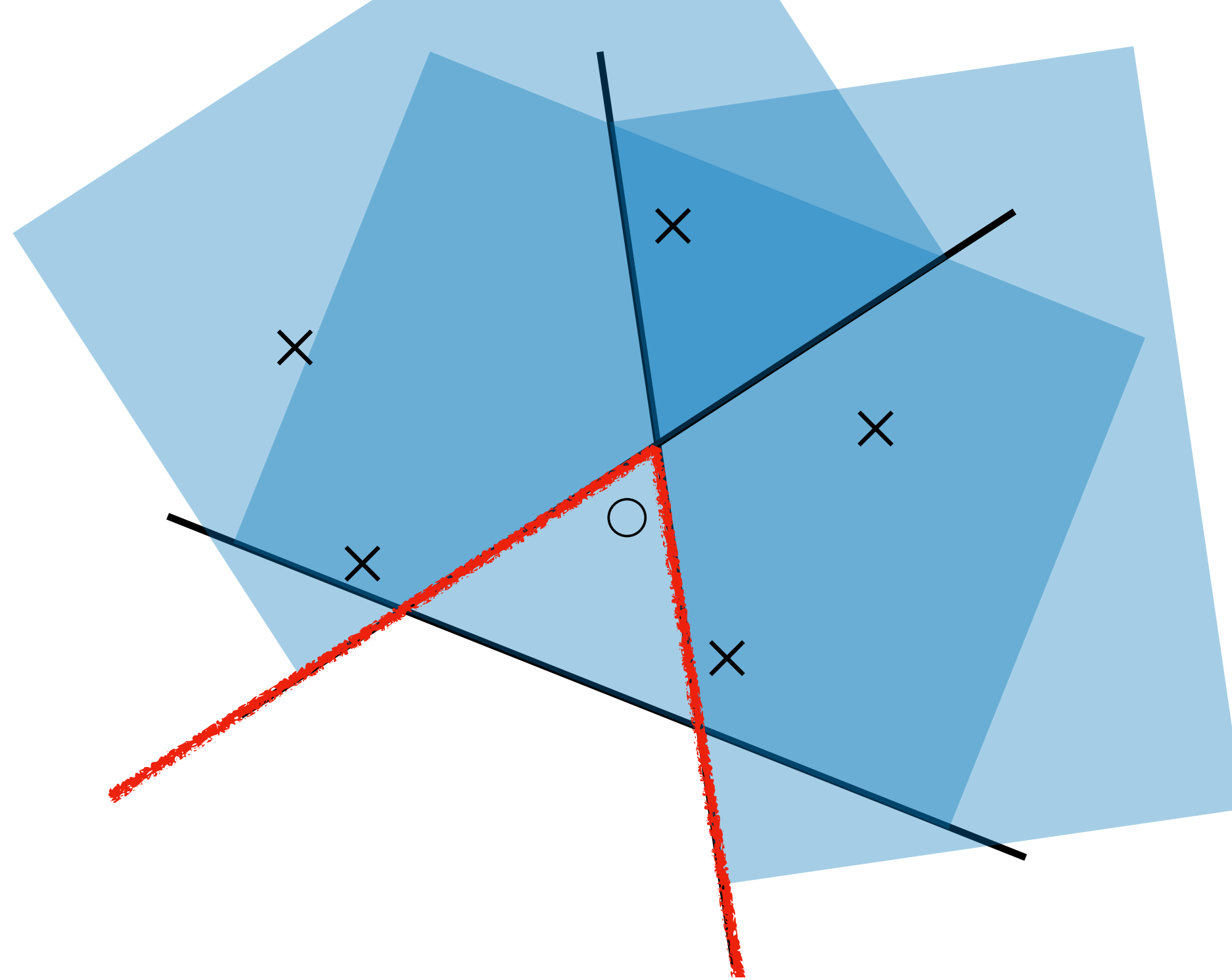
- We just saw an example: bagged classifier is stored as a sequence of vector-space operations on base learners like decision trees

Functional gradient

- What is the functions-as-vectors view good for?
- Example application: learning by **functional gradient**
 - ▶ instead of gradient of loss wrt θ , take gradient wrt f — then step in function space r.t. parameter space
- Advantages:
 - ▶ convexity: with the extra flexibility of moving in function space, we avoid getting stuck in local optima
 - ▶ black-box access: someone has a fancy algorithm for fitting a function, but will only run it on their own servers
 - ▶ parameter independence: the answer doesn't change if we reparameterize our set of functions
- Todo: setup (choose $L(f)$), gradients (define $\nabla_f L$)
 - ▶ we'll use the example problem of learning a voted classifier for higher accuracy, called **boosting**

Boosting

a wrapper we can put around any classification algorithm to try to make it stronger



ex: vote of 3 linear classifiers

- Problem: learn a *voted classifier*

- ▶ each base classifier $f_t : \mathbb{R}^d \rightarrow \{-1, 1\}$

can also do $\mathbb{R}^d \rightarrow [-1, 1]$
or even $\mathbb{R}^d \rightarrow \mathbb{R}$

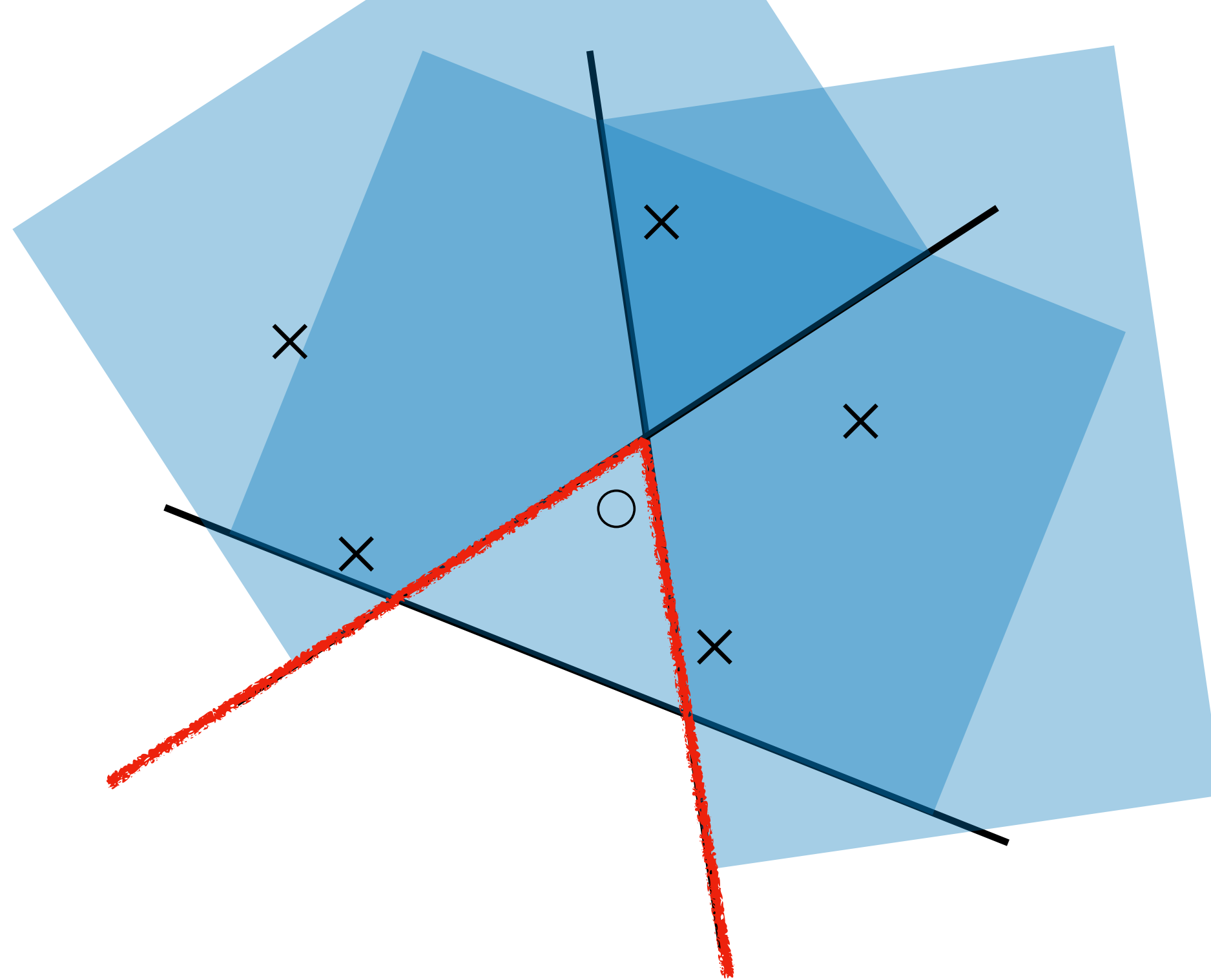
- ▶ vote: is $\sum_t f_t(x) \geq 0$?

- ▶ or weighted vote: is $\sum_t \eta_t f_t(x) \geq 0$? (weights η_t)

- ▶ wlog $\eta_t > 0$ for all t (assume \mathcal{F} closed under negation)

- ▶ normalize η_t if desired: divide by sum (doesn't affect vote)

Build a better classifier



ex: vote of 3
linear classifiers

- Voted classifier can be *strictly more expressive* than base classifiers: wedge-shaped red region $\notin \mathcal{F}$
- Take advantage of higher expressiveness to get higher accuracy (“boost” the base learner)
 - ▶ **weak** learner (base classifier, in \mathcal{F}) vs. **strong** learner (voted classifier, in \mathcal{H})

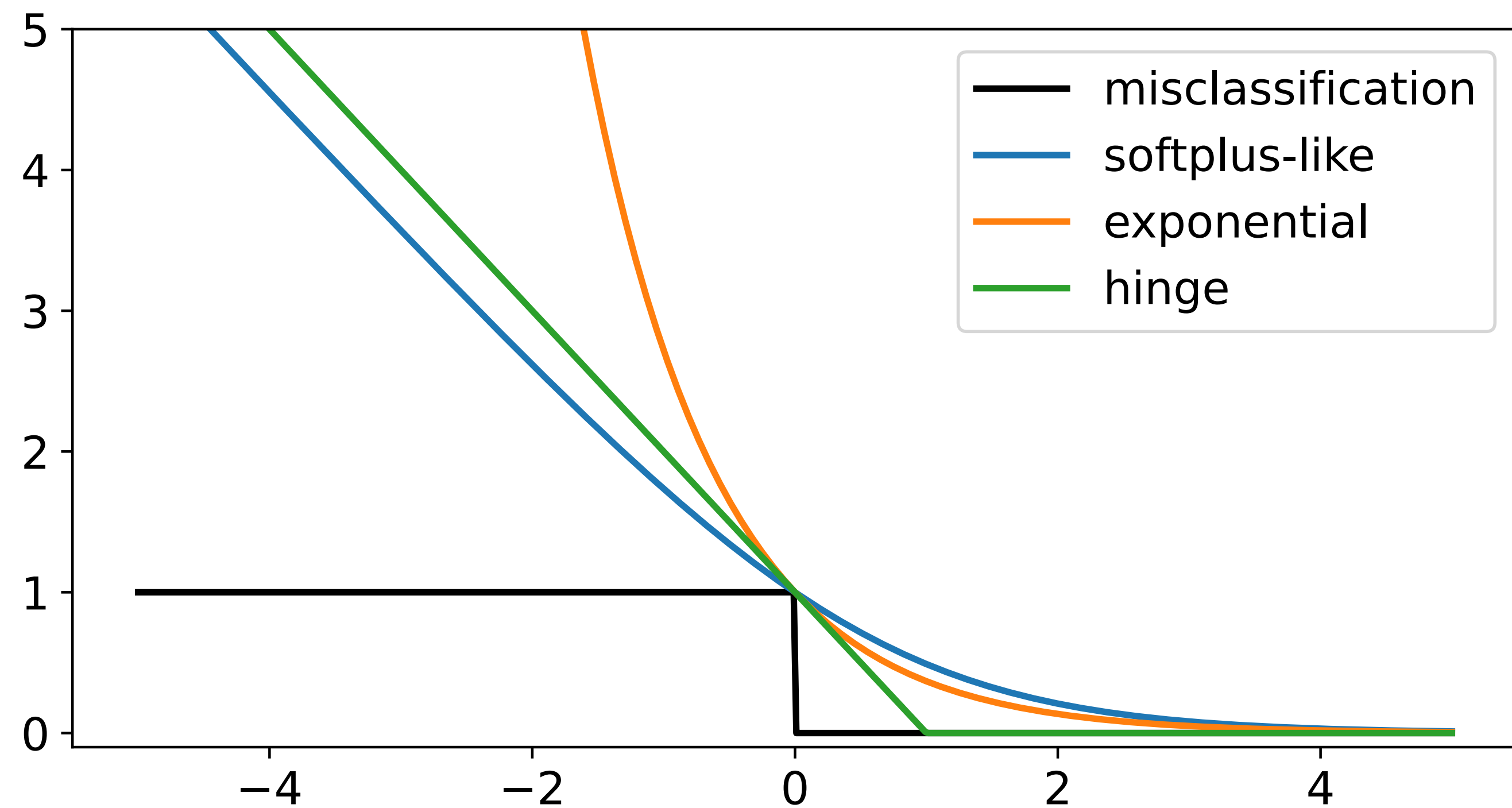
Comparison: boosting vs. bagging

- Both boosting and bagging are ensembles: vote over list of hypotheses from base hypothesis class
- But their purpose is different:
 - ▶ bagging tries to capture uncertainty (posterior over classifiers, predictive distribution) without necessarily beating the base hypothesis class
 - ▶ boosting tries to beat the base hypothesis class (weak learners), doesn't help much with estimating uncertainty
 - ▶ common boosting methods are focused on binary classification (though generalizations exist); bagging is easy to use for multi-class and regression

Comparison: boosting vs. bagging

- And their behavior is different
 - ▶ many small trees / weak hypotheses (boosting) vs. fewer large trees / stronger hypotheses (bagging)
 - ▶ fit different parts of target concept (boosting) vs. all of it in different ways (bagging)
 - ▶ limits overfitting by max margin (boosting) vs. by randomization and averaging (bagging)

Boosting setup



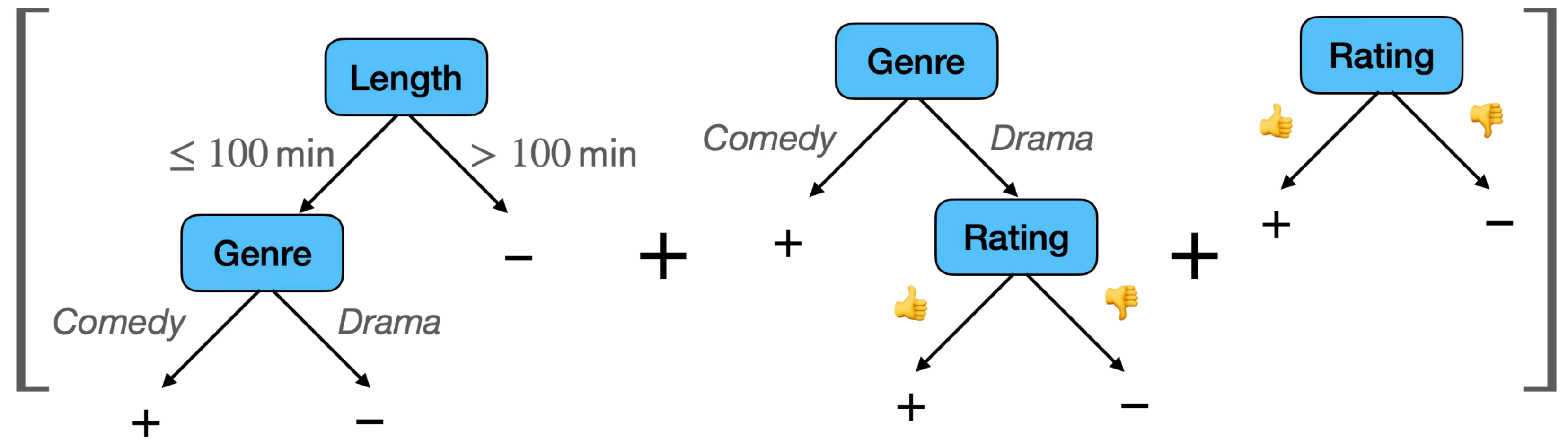
- Given a training set $\{(x^{(i)}, y^{(i)})\}_{i=1}^N$ where
 - ▶ $x^{(i)} \in \mathcal{X}$ (any feature set)
 - ▶ $y^{(i)} \in \{\pm 1\}$ (binary classification, classes +1 and -1)and a base hypothesis set $\mathcal{F} \subset \mathcal{X} \rightarrow \{\pm 1\}$
- Pick a convex per-example loss function ℓ that upper-bounds misclassification loss

*minimizing ℓ means
we want its argument
to be positive*

*we'll use ℓ to
construct total loss L*

Boosting loss

$$\mathcal{H} = \left\{ \sum_t \eta_t f_t \mid f_t \in \mathcal{F} \right\}$$



$y^{(i)} f(x^{(i)})$: want $f(x^{(i)}) > 0$ when $y^{(i)} = +1$,
 $f(x^{(i)}) < 0$ when $y^{(i)} = -1$

- Our goal:

$$\min_{f \in \mathcal{H}} L(f) = \sum_{i=1}^N \ell(y^{(i)} f(x^{(i)}))$$

- ▶ where \mathcal{H} = weighted votes over elements of \mathcal{F} (before thresholding, unnormalized weights)
- Minimize a convex loss function over a convex set


Finite-d analogy

$$\min_{f \in \mathcal{H}} L(f) = \sum_{i=1}^N \ell(y^{(i)} f(x^{(i)}))$$

- If our base learners are linear-in-parameters functions, then $\mathcal{F} = \mathcal{H} = \mathbb{R}^d$: $\sum_t \eta_t (\phi(x) \cdot \theta_t) = \phi(x) \cdot \sum_t \eta_t \theta_t$
 - ▶ i.e., we were already convex, so boosting doesn't increase expressiveness
- So the optimization becomes
 - note: outputs are in \mathbb{R} not $\{\pm 1\}$ but this is OK*
- $$\min_{\theta \in \mathbb{R}^d} \sum_{i=1}^N \ell(y^{(i)} \phi(x^{(i)}) \cdot \theta)$$
- This setup is exactly what we used for linear classifiers:
 - ▶ $\ell(z) = \ln(1 + \exp(-z))$ yields logistic regression
 - ▶ $\ell(z) = [-z]_+$ yields perceptron
- So, novelty of boosting is handling ∞ -dim nonconvex \mathcal{F}

Poll



- Which of the following statements are true about boosting and bagging?
 - A. Boosting and bagging end up with the same form of classifier: a weighted vote over base classifiers
 - B. Boosting can work well with simple base classifiers, while bagging generally needs more expressive ones
 - C. Boosting won't help if our base learners are linear-in-parameters, but bagging can still be useful
 - D. All of the above 
 - E. None of the above

Gradient of loss

$$\min_{f \in \mathcal{H}} L(f) = \sum_{i=1}^N \ell(y^{(i)} f(x^{(i)}))$$

- Functional gradient descent update: subtract a multiple of $\nabla_f L(f)$ from current hypothesis f
- Still need to define ∇_f but let's see what we can do given that it will act like other derivative operators:

$$\begin{aligned} \nabla_f L(f) &= \sum_{i=1}^N \nabla_f \ell(y^{(i)} f(x^{(i)})) \\ &= \sum_{i=1}^N \ell'(y^{(i)} f(x^{(i)})) y^{(i)} \nabla_f f(x^{(i)}) \end{aligned}$$

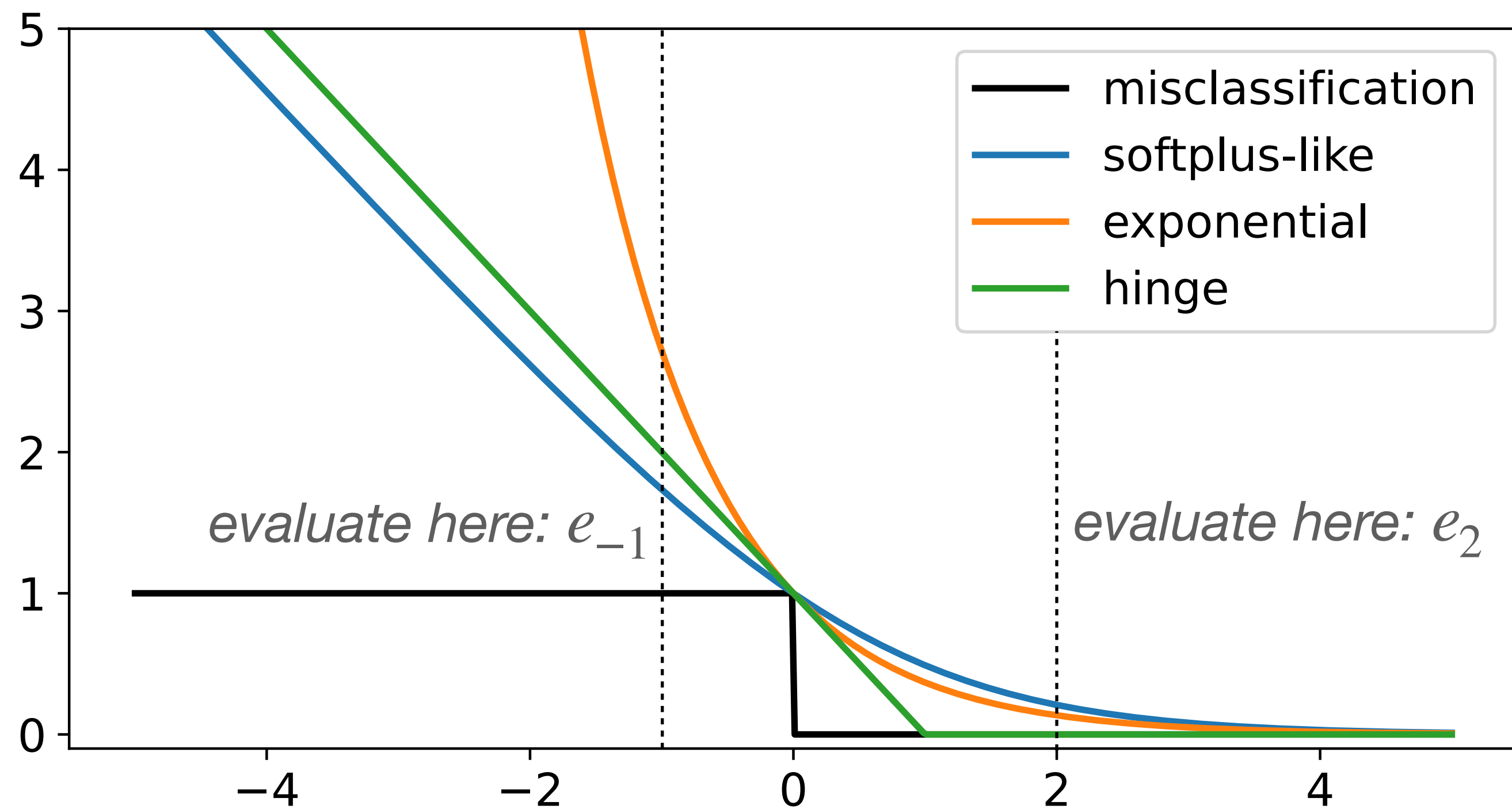
- So, just need to figure out what to do with $\nabla_f f(x^{(i)})$

Functionals

- What is the data type of $\nabla_f f(x)$?
- Recall: derivatives represent linear functions
 - ▶ e.g., $d\|x\|^2 = 2x^\top dx$: the change in $\|x\|^2$ is locally approximated by a linear function of the change in x
- So we want a function that maps $df \in \mathcal{H}$ (the change in f) to $d(f(x)) \in \mathbb{R}$ (the change in $f(x)$)
- This is called a **functional**:
 - ▶ a function that takes another function as its argument and returns a scalar
 - ▶ in our case a linear functional of type $\mathcal{H} \rightarrow \mathbb{R}$
- Other examples: $L(f)$ is a (nonlinear) functional, and...

Ex: point evaluation functional

some functions to evaluate



- **Point evaluation** functional e_x is defined as $e_x(f) = f(x)$
- E.g., $e_2(\text{hinge}) = 0$ or $e_{-1}(\text{exponential}) = 2.718$
- e_x is a linear functional:
$$e_2(3 \sin x + 1) = 3e_2(\sin x) + e_2(1) = 3 \sin 2 + 1$$

Linear functionals as inner products

- We've encountered some useful examples of linear functionals: $\nabla_f f(x)$ and e_x
- In \mathbb{R}^d we can always write a linear functional $a(x)$ as $\tilde{a} \cdot x$ for some vector \tilde{a} (the ***representer*** of $a(x)$)
 - ▶ in fact we often don't bother distinguishing: use a for both and let context decide whether we mean a vector $a \in \mathbb{R}^d$ or a linear functional $a \in \mathbb{R}^d \rightarrow \mathbb{R}$
- This property is essential for gradient descent (so essential we rarely even think about the type cast):
 - ▶ each step adds a gradient (a linear functional) to the parameter we're trying to optimize (a vector)
 - ▶ if they were incompatible types, addition would break
- So for functional gradient, we need to define an inner product and interpret $\nabla_f f(x)$ as $\langle g, df \rangle$ for some g

Inner product

- A valid inner product $\langle f, g \rangle$ satisfies:
 - ▶ **linear** in each argument: $\langle 3f + g, h \rangle = 3\langle f, h \rangle + \langle g, h \rangle$
 - ▶ **symmetric**: $\langle f, g \rangle = \langle g, f \rangle$
 - ▶ **positive definite**: $\langle f, f \rangle = 0$ iff f is the all-zero function
- It tells us which pairs of unit vectors are similar (high inner product) or different (low or negative inner product)
- Typically \exists many ways to define an inner product on a set of functions, leading to spaces that behave differently
 - ▶ e.g., $\langle f, g \rangle = \int_x f(x)g(x) dx$ (like dot, leads to L_2)
 - ▶ or $\langle f, g \rangle = \frac{1}{2} \int_x f(x)g(x) dx + \frac{1}{2} \int_x f'(x)g'(x) dx$
(leads to a Sobolev space) — a good example to keep in mind, since L_2 fns are too non-smooth to be good for ML
 - ▶ or (many) other examples we'll define later

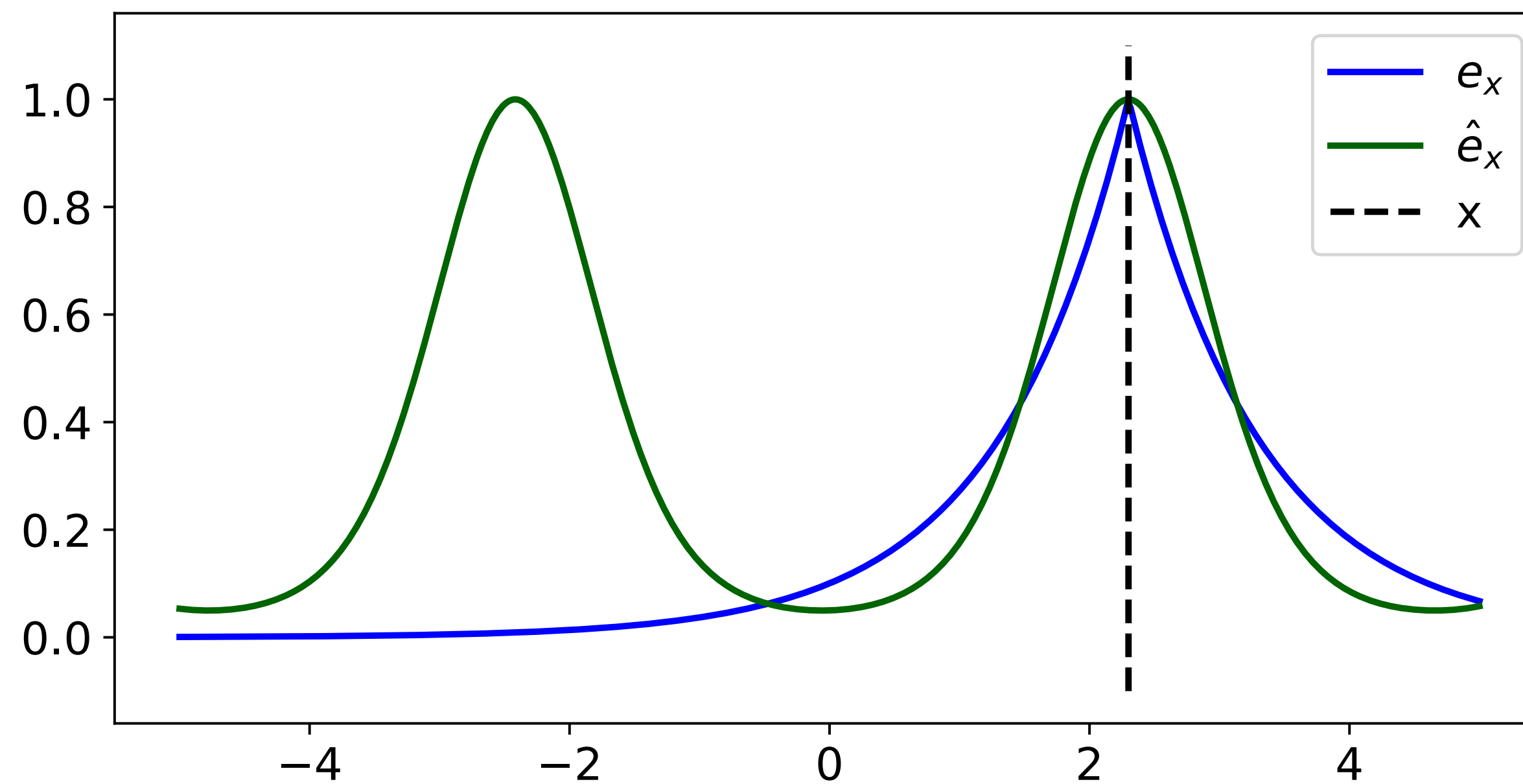
Finite-d analogy

- For linear-in-parameters functions,
 - ▶ the inner product is the usual dot product on \mathbb{R}^d
 - ▶ the representer of $f_\theta(x) = \theta \cdot \phi(x)$ is the vector θ
 - ▶ the representer of a point-evaluation functional is $e_x = \phi(x)$
- So, to evaluate a function at a point, we take the dot product between f_θ 's vector θ and e_x 's vector $\phi(x)$:
 - ▶ $e_x(f_\theta) = \langle e_x, f_\theta \rangle = \phi(x) \cdot \theta$
- Surprising conclusion: point-evaluation functionals are the generalization of **feature vectors** of points

Point-eval functionals as feature vectors

point-evaluation functionals are related to kernels — more about this soon

two different point evaluation functionals for the same x , different inner products



blue is for Sobolev space defined above

- We'll shortly assume the representer e_x for a point evaluation functional exists and is a function in \mathcal{H} — i.e., we can evaluate it at other values x'
- The shape of e_x depends on how we define inner product (which pairs of unit vectors in \mathcal{H} we think are similar)
- The function values ($e_x(3.27)$, $e_x(-1.05)$, ...) act like features of the point x : high for x' we consider similar to x

Extending the API

- We've just extended our API
 - ▶ from vector space (supports addition and scalar product)
 - ▶ to *inner product space* (also supports $\langle f, g \rangle$)
- Turns out we need a couple more operations before we can define gradients:
 - ▶ taking limit of a sequence $\lim_{t \rightarrow \infty} f_t = f$: adding limits to our API yields a *Hilbert space*
 - ▶ representer for point evaluation, $x \mapsto e_x$ (type $\mathcal{X} \rightarrow \mathcal{H}$): yields a *reproducing kernel Hilbert space* or *RKHS*
- All these ops work fine on \mathbb{R}^d , so adding them to API lets us treat functions even more like ordinary vectors

Gradients

- New API lets us define and compute derivative $\nabla_f f(x)$
- We added three things: inner product, limits, point eval
 - ▶ inner product lets us express derivative as a vector $g \in \mathcal{H}$ representing the linear functional $\langle g, df \rangle$
 - ▶ limits let us define which g we want:
if $df \rightarrow 0$ then $\frac{\|[(f + df)(x) - f(x)] - \langle g, df \rangle\|}{\|df\|} \rightarrow 0$
 - ▶ we'll shortly see that $x \mapsto e_x$ ensures gradient exists and lets us compute it

Gradients

- In fact it turns out ***the derivative*** $\nabla_f f(x)$ ***is exactly*** e_x
 - ▶ to see why: $(f + df)(x) = f(x) + (df)(x)$
 - ▶ so the change in $f(x)$ is df evaluated at x , or $\langle e_x, df \rangle$
- Assuming the API operation $x \mapsto e_x$ means assuming that e_x must exist in \mathcal{H}
 - ▶ i.e., the assumption restricts how we choose an inner product in order to guarantee $f(x)$ is differentiable wrt f

Good- enough step direction

- Substituting in, the gradient we want is

$$\nabla_f L(f) = g = \sum_{i=1}^N \ell'(y^{(i)} f(x^{(i)})) y^{(i)} e_{x^{(i)}}$$

- Problem: this g probably isn't in \mathcal{F}
- So we have to work with it as a list (slow) instead of as a base classifier (fast)
- Solution: instead of using gradient g directly, pick an element of \mathcal{F} that has *positive inner product* with desired step direction $-g$
 - ▶ recall that gradient descent still works if we use an approximate step, as long as on average the inner product between step and $-g$ is enough bigger than 0

Solving for the step direction

to ensure inner product is big enough, we can (approximately) maximize

$$f_t = \max_{f \in \mathcal{F}} \langle f, -g \rangle$$

$$= \max_{f \in \mathcal{F}} \langle f, -\sum_{i=1}^N \ell'(y^{(i)} f(x^{(i)})) y^{(i)} e_{x^{(i)}} \rangle$$

$$= \max_{f \in \mathcal{F}} \sum_{i=1}^N -\ell'(y^{(i)} f(x^{(i)})) y^{(i)} \langle f, e_{x^{(i)}} \rangle$$

$$= \max_{f \in \mathcal{F}} \sum_{i=1}^N \underbrace{\alpha^{(i)}}_{\text{weight}} \underbrace{y^{(i)} f(x^{(i)})}_{\substack{+1 \text{ if correct} \\ -1 \text{ if error}}}$$

ℓ' is negative, so $\alpha^{(i)} > 0$

- Maximize weighted number of correct predictions: a classification problem on weighted copy of our dataset!
- More weight for points whose current signed prediction $z^{(i)} = y^{(i)} f(x^{(i)})$ has big effect on loss (big derivative ℓ')

Boosting summary

- For $t = 1, 2, \dots$:
 - ▶ compute weights $\alpha_t^{(i)}$ for each point: $\alpha_1^{(i)}$ uniform, $\alpha_t^{(i)}$ based on influence of point i on loss at step $t - 1$
 - ▶ find a good weak learner f_t (an approximate functional gradient): correlation $> \epsilon$ on weighted data
 - ▶ take a step in direction f_t : add f_t to our vote, weight η_t
- Different choices of ℓ and η_t yield different boosting methods

***Did this
help?***

- In order to solve our classification problem, we have to repeatedly reweight our data and solve a classification problem

***Did this
help?***

- In order to solve our ***strong*** classification problem, we have to repeatedly reweight our data and solve a ***weak*** classification problem

Did this help?

- In order to solve our **strong** classification problem, we have to repeatedly reweight our data and solve a **weak** classification problem
- Idea: if base algorithm (the weak learner) can keep doing just a bit better than chance, then final boosted classifier (the strong learner) can keep improving
 - ▶ eventually will get zero error on the training set

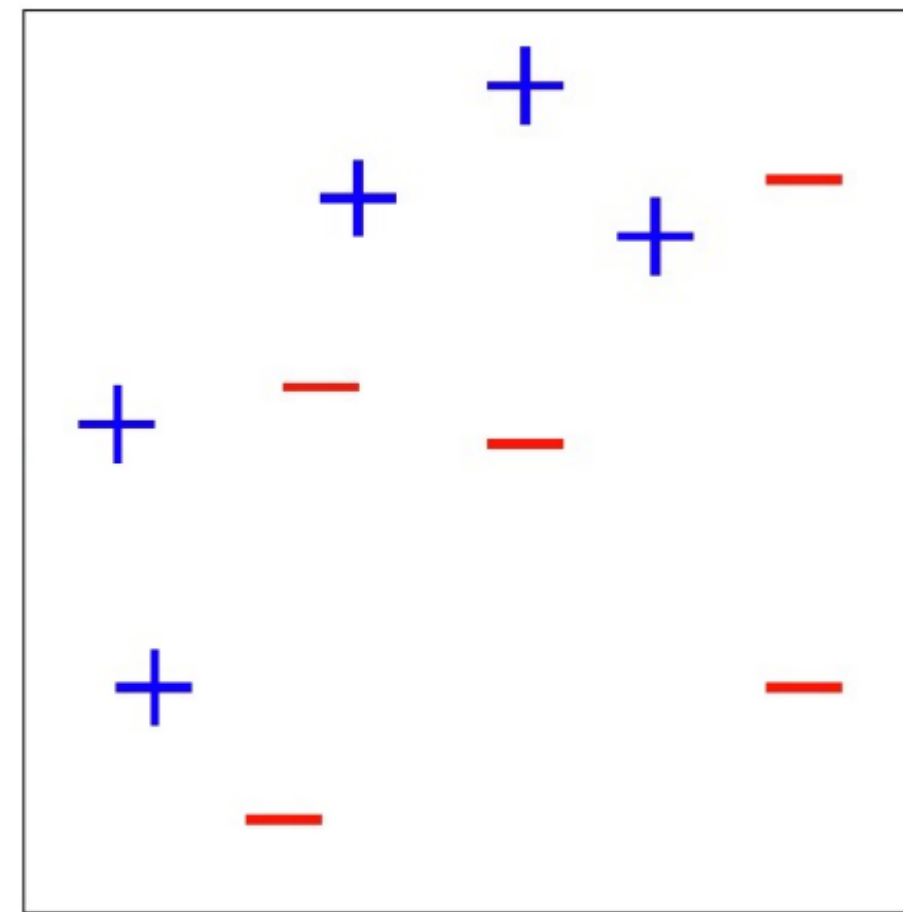
AdaBoost

uses exponential loss e^{-z}

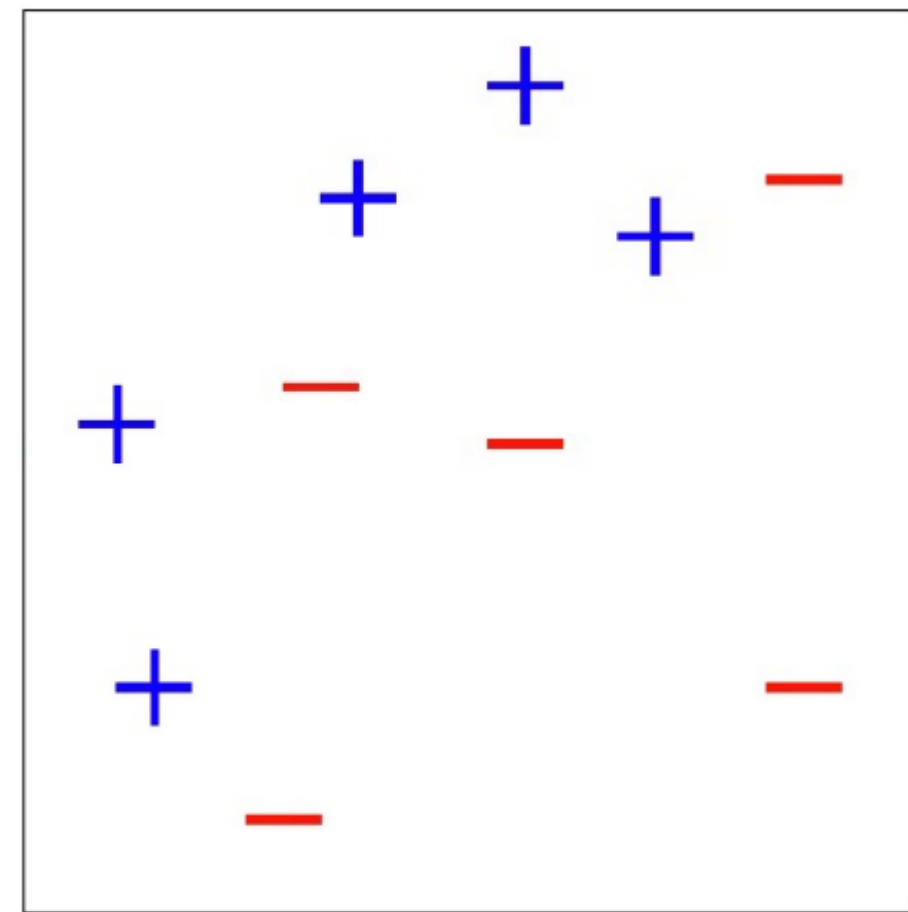
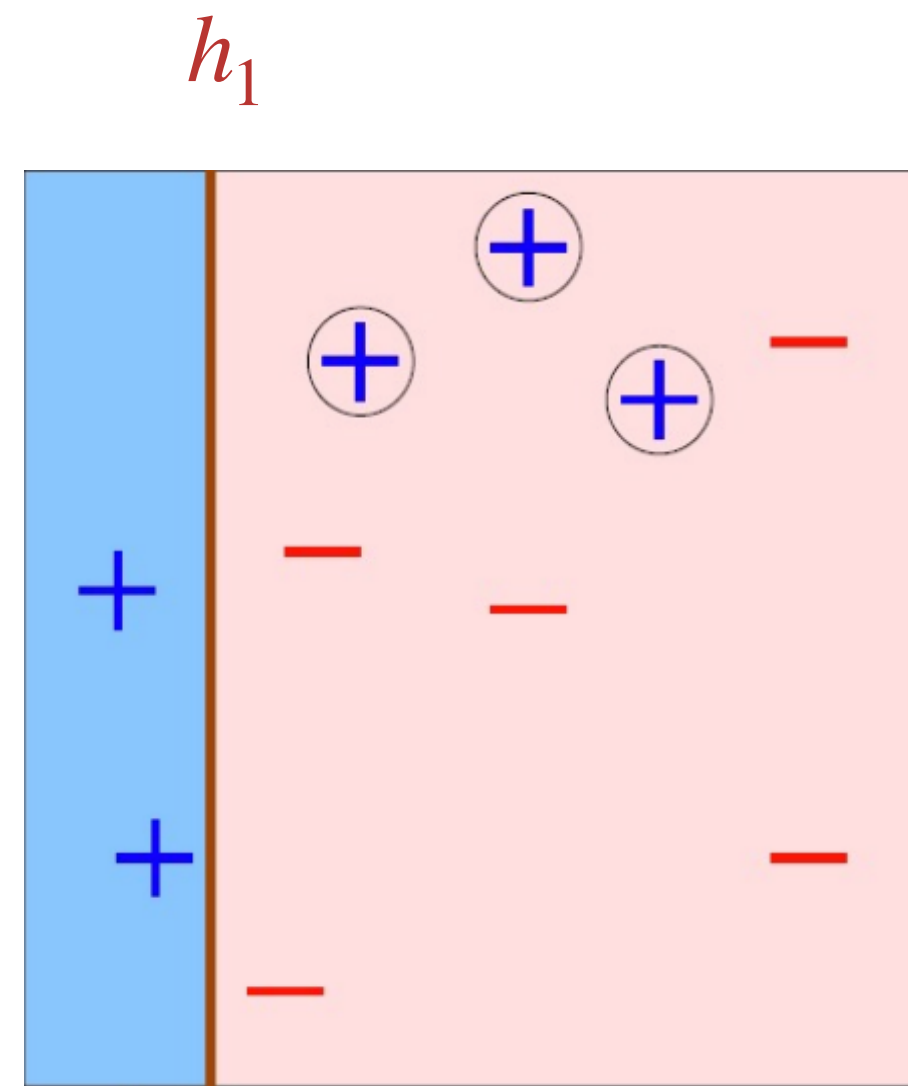
$$L = e^{-z}$$

- Input: number of rounds T , dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^N$
- Initialize datapoint weights $\alpha_0^{(i)} = \frac{1}{N}$ for all i
- For $t = 1, \dots, T$:
 - ▶ Train a weak learner f_t by minimizing *weighted* training error on \mathcal{D} (weights $\alpha_{t-1}^{(i)}$)
 - ▶ Find weighted training error of f_t :
$$\epsilon_t = \sum_{i=1}^N \alpha_{t-1}^{(i)} \mathbb{1}(y^{(i)} \neq f_t(\mathbf{x}^{(i)}))$$
 - ▶ Compute the vote weight of f_t : $\eta_t = \frac{1}{2} \ln \frac{1 - \epsilon_t}{\epsilon_t}$ *optimal step size*
 - ▶ Update datapoint weights:
$$\alpha_t^{(i)} = \frac{\alpha_{t-1}^{(i)}}{Z_t} \cdot \begin{cases} e^{-\eta_t} & y^{(i)} = f_t(\mathbf{x}^{(i)}) \\ e^{\eta_t} & \text{o/w} \end{cases} \quad (Z_t \text{ is normalizer})$$
- Return: weighted vote classifier $\hat{y} = \text{sign} \left(\sum_{t=1}^T \eta_t f_t(\mathbf{x}) \right)$

AdaBoost *example*



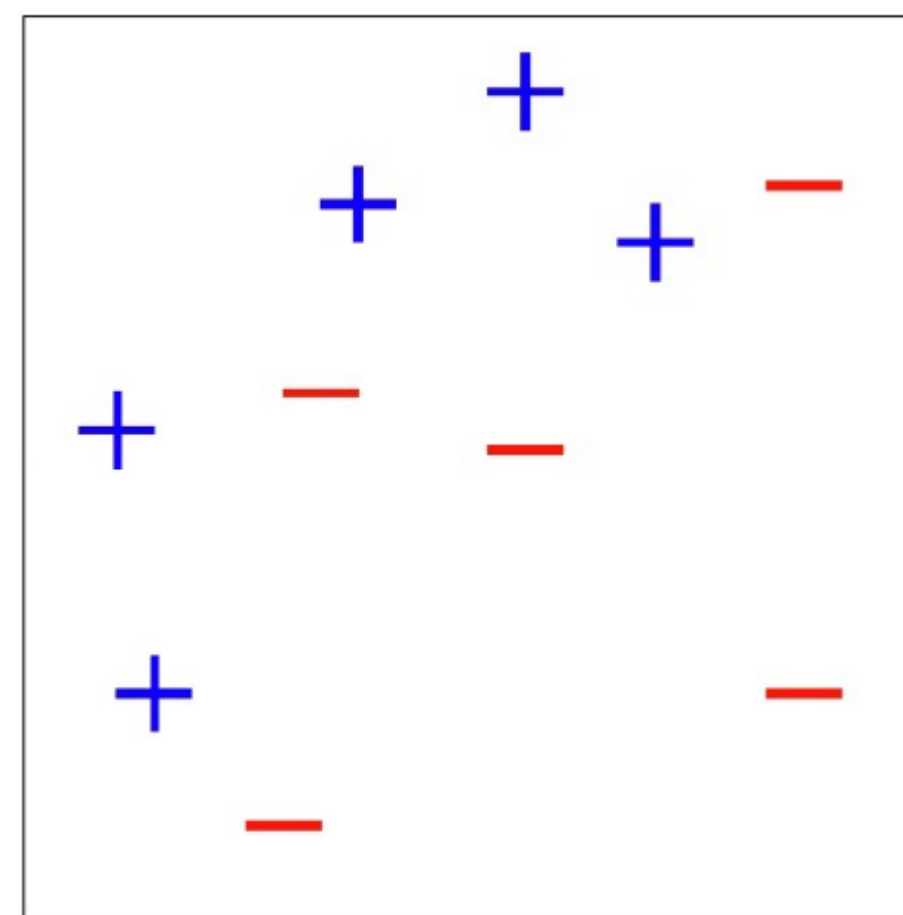
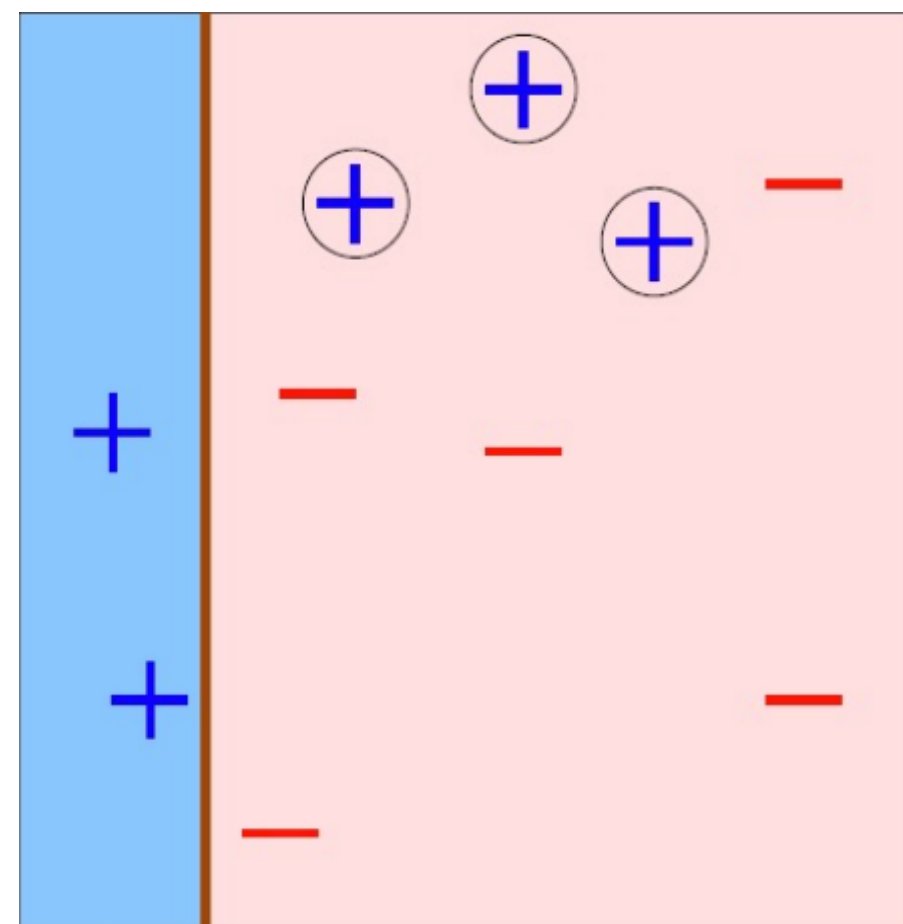
AdaBoost *example*



AdaBoost example

\mathcal{M}
 h_1

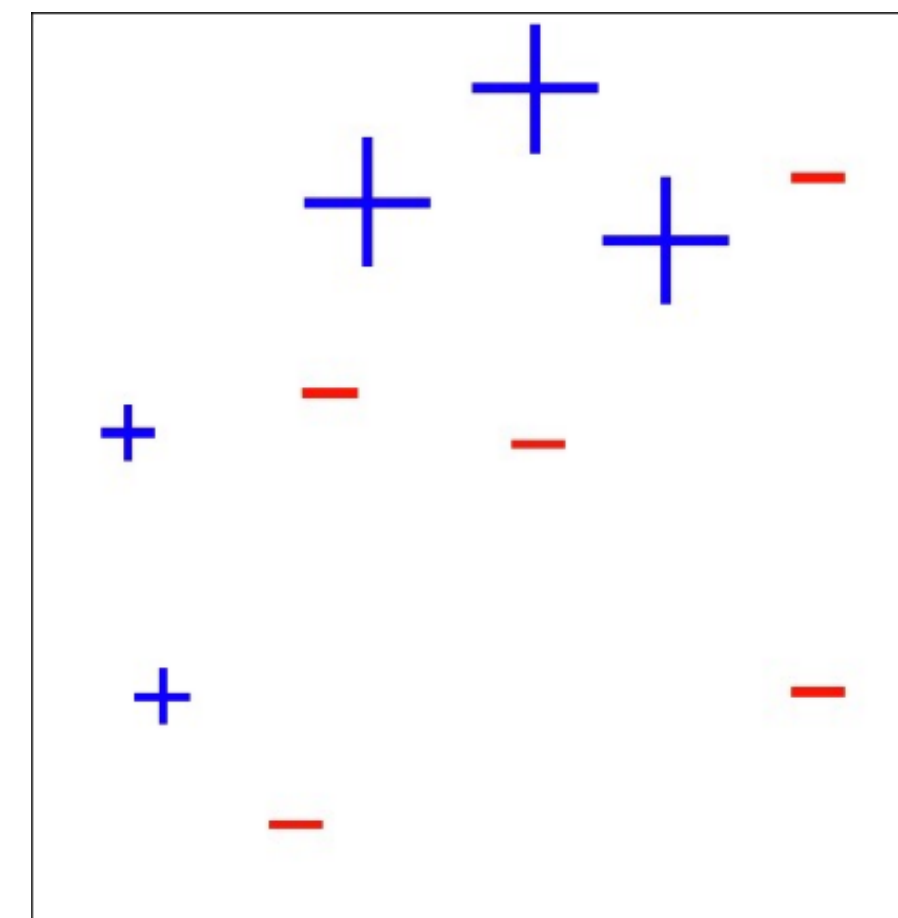
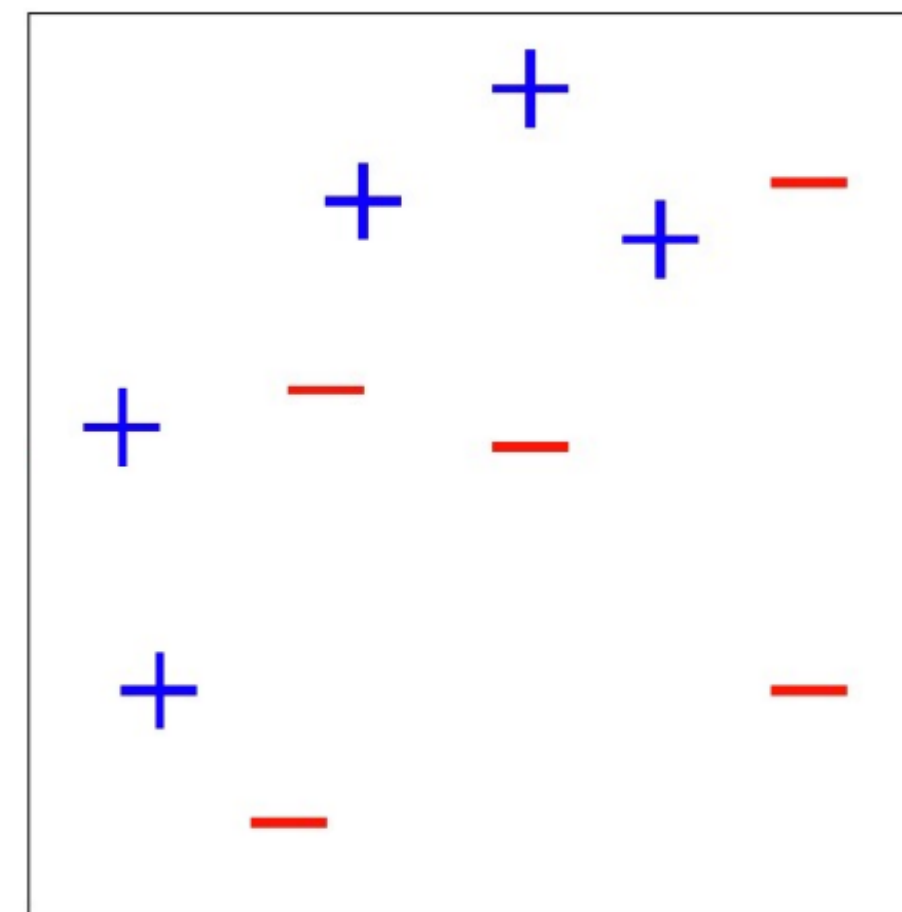
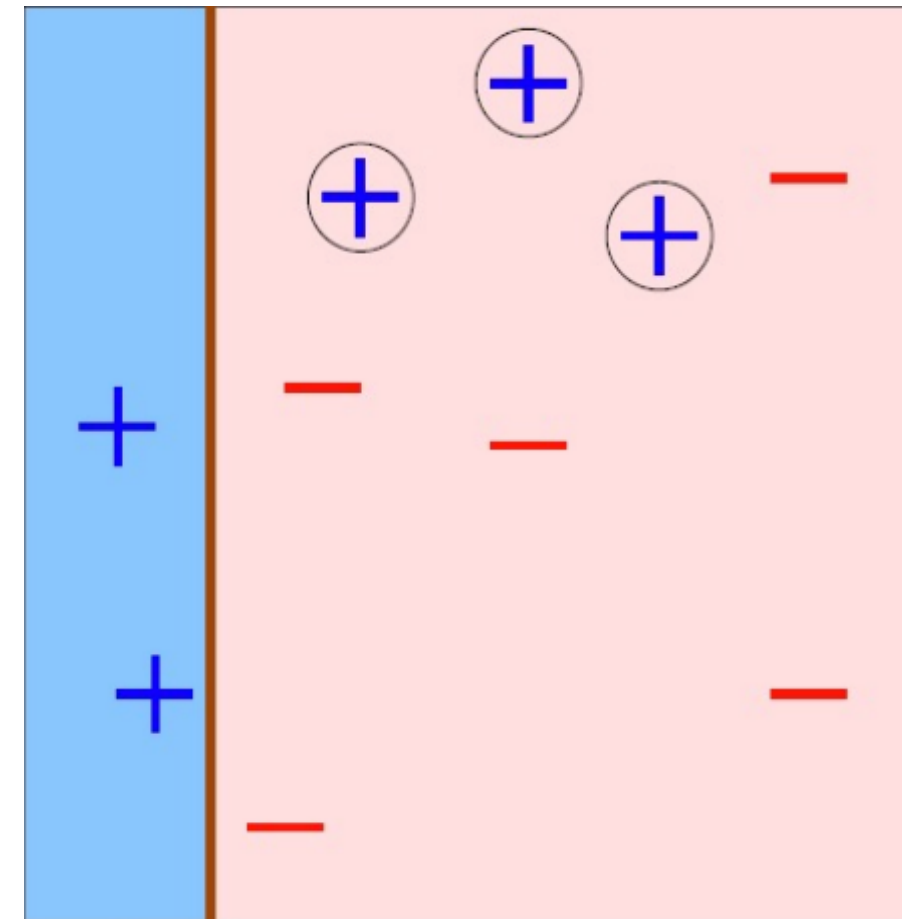
$\epsilon_1 = 0.3$
 $\alpha = 0.42$



AdaBoost example

$$\epsilon_1 = 0.3$$
$$\alpha_1 = 0.42$$

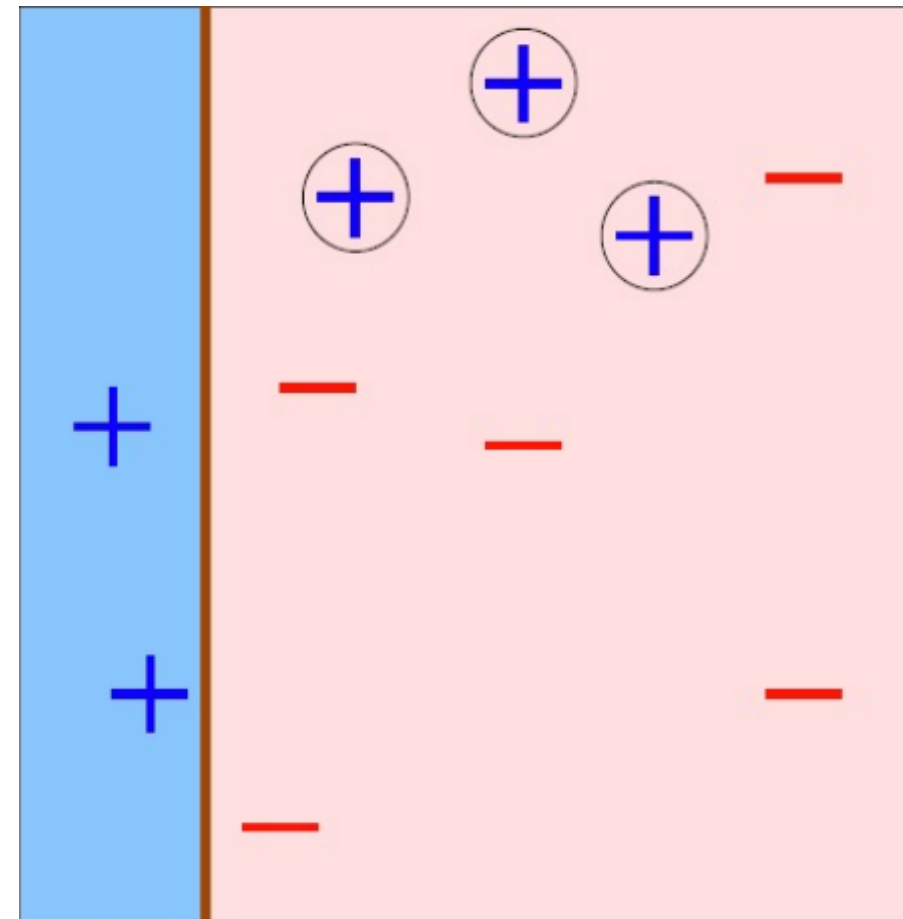
h_1



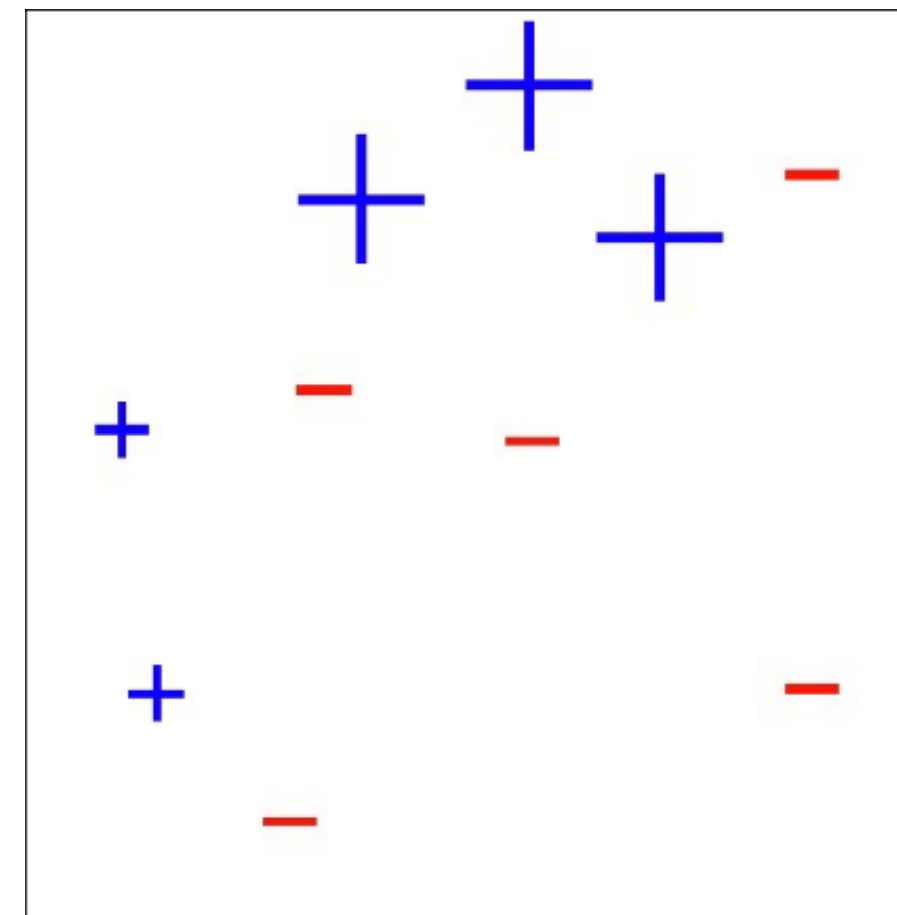
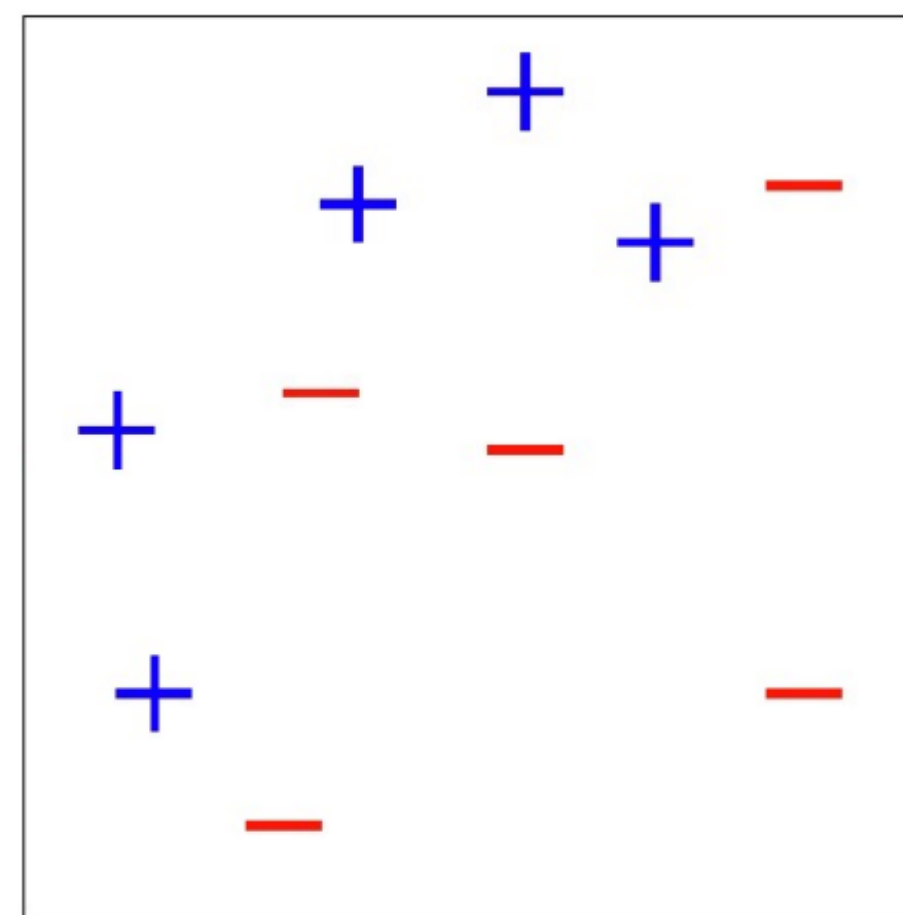
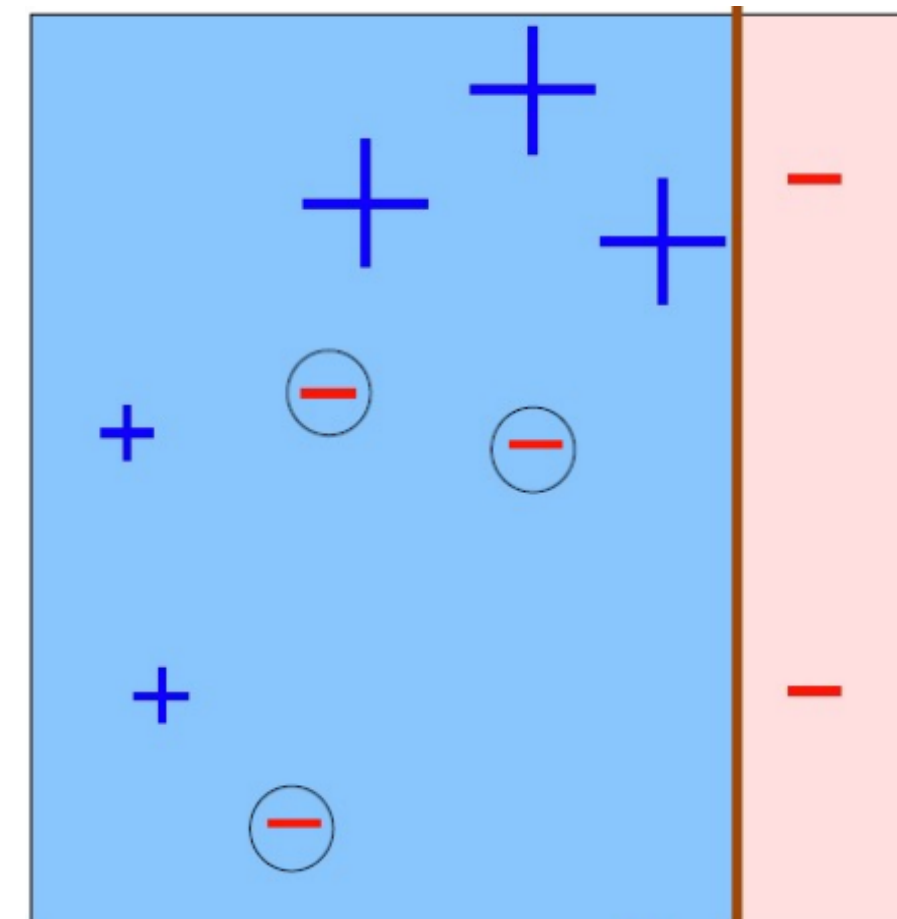
AdaBoost example

$$\epsilon_1 = 0.3$$
$$\alpha_1 = 0.42$$

h_1



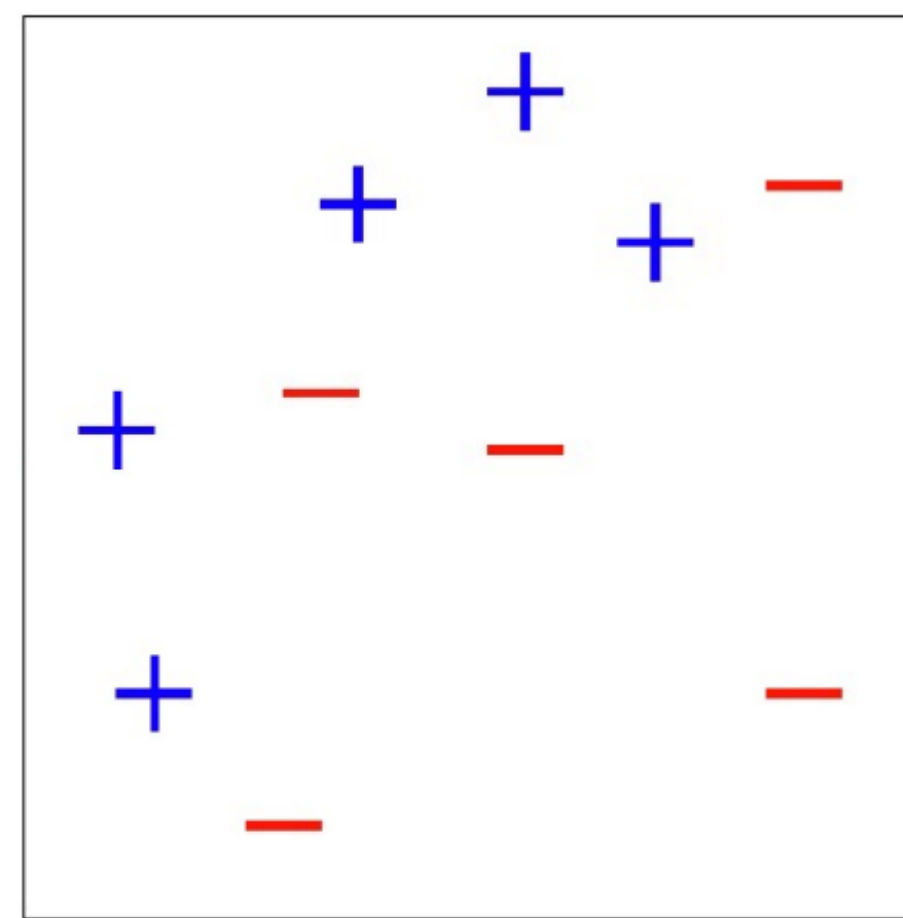
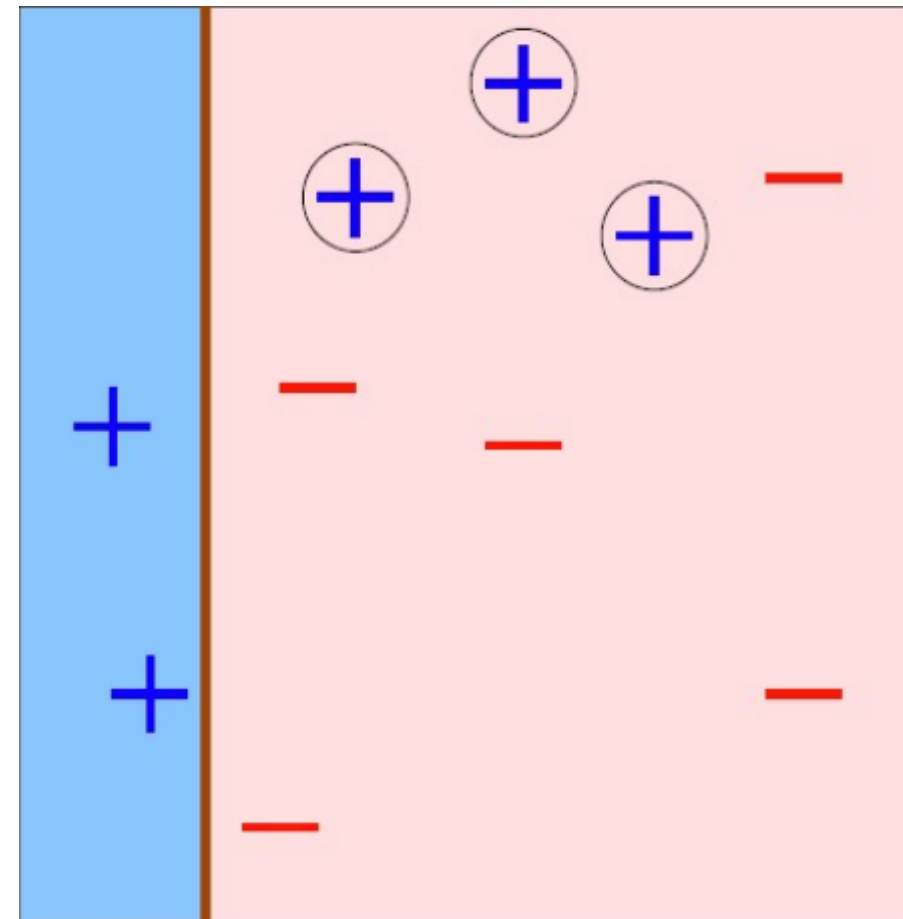
h_2



AdaBoost example

$$\epsilon_1 = 0.3$$
$$\alpha_1 = 0.42$$

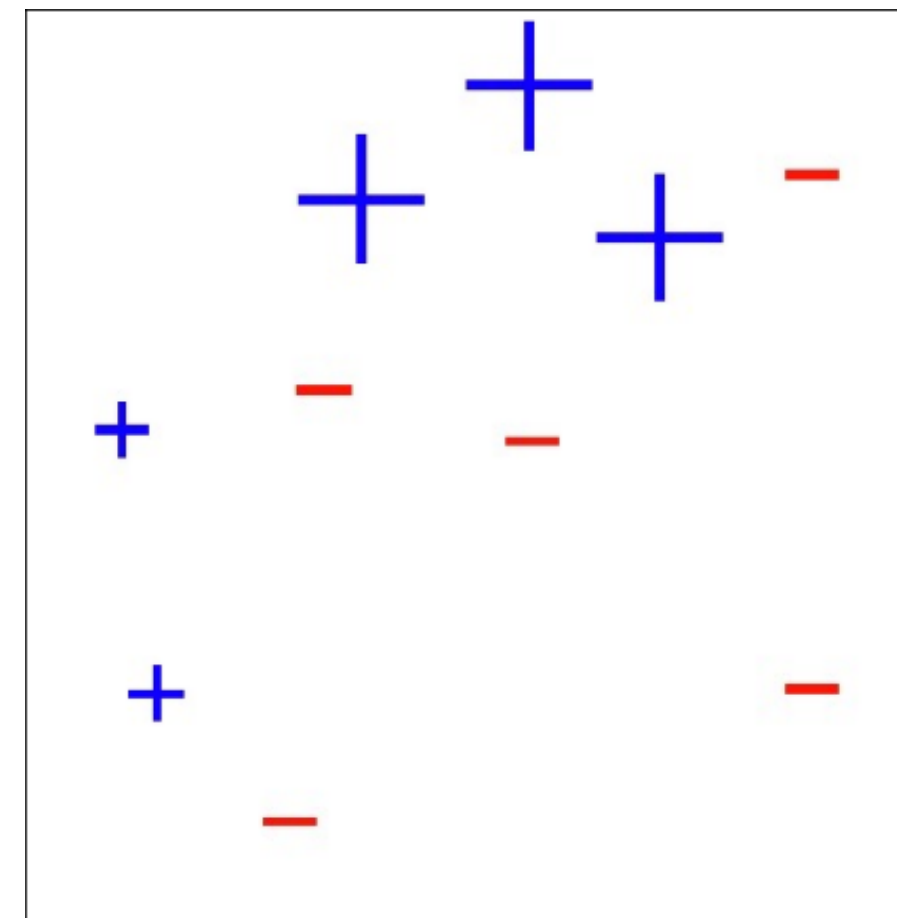
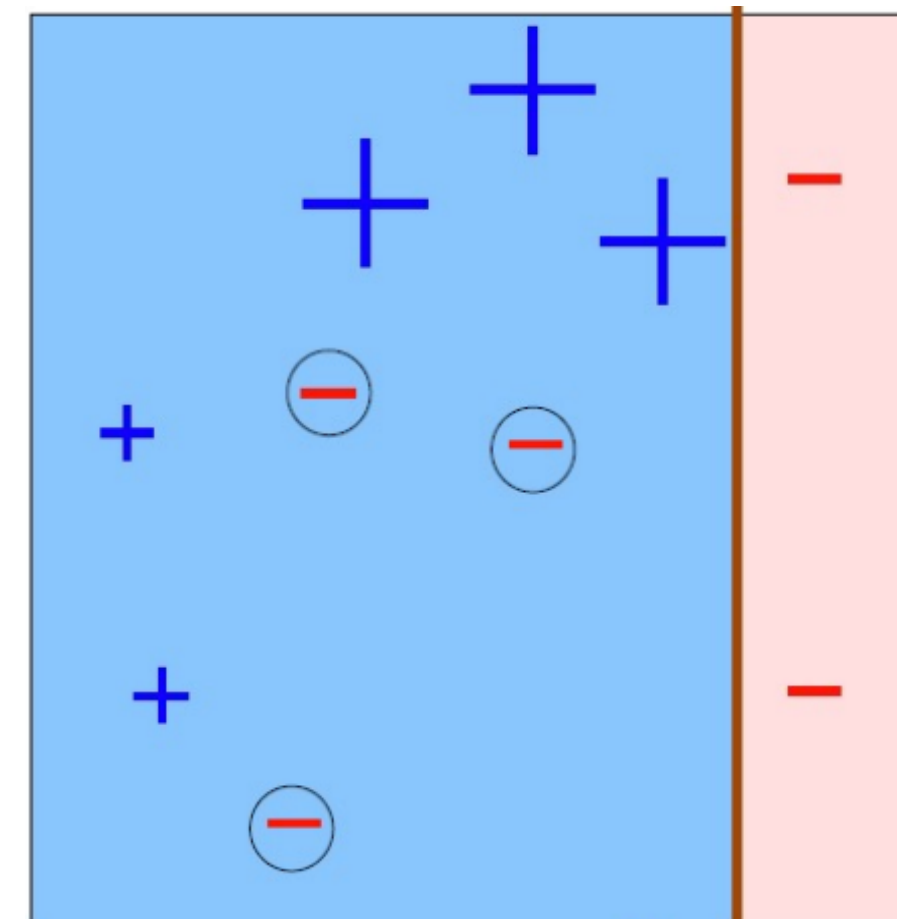
h_1



$$\epsilon_2 = 0.21$$
$$\alpha_2 = 0.65$$

h_2

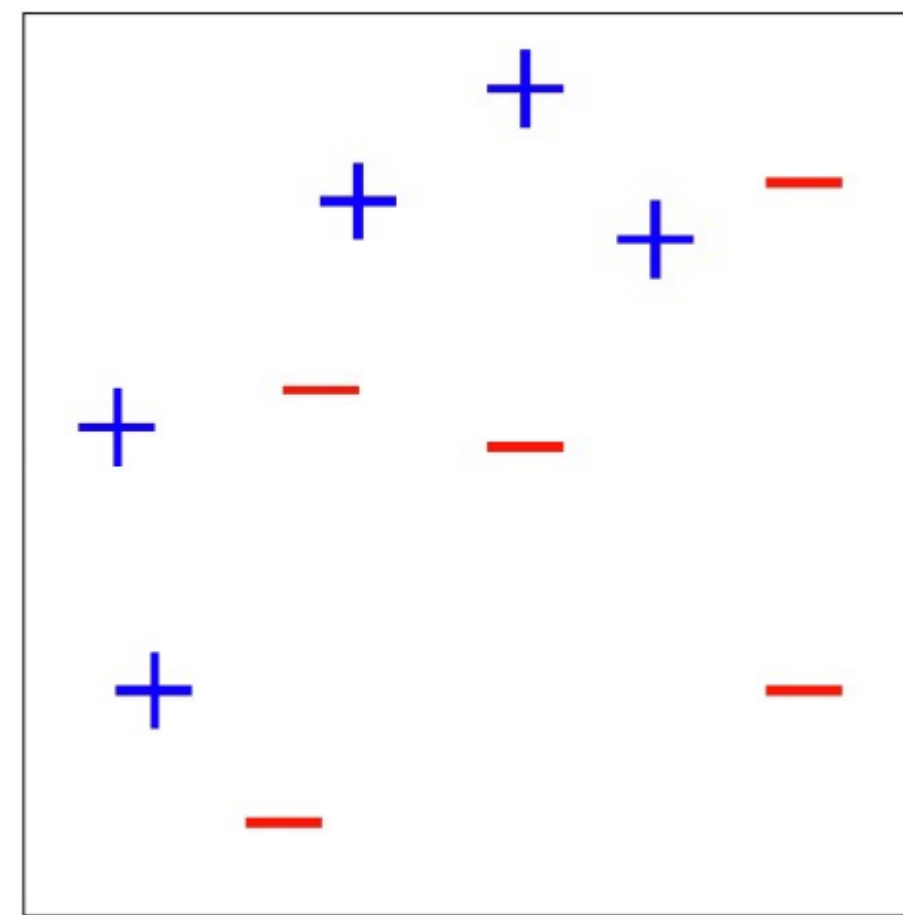
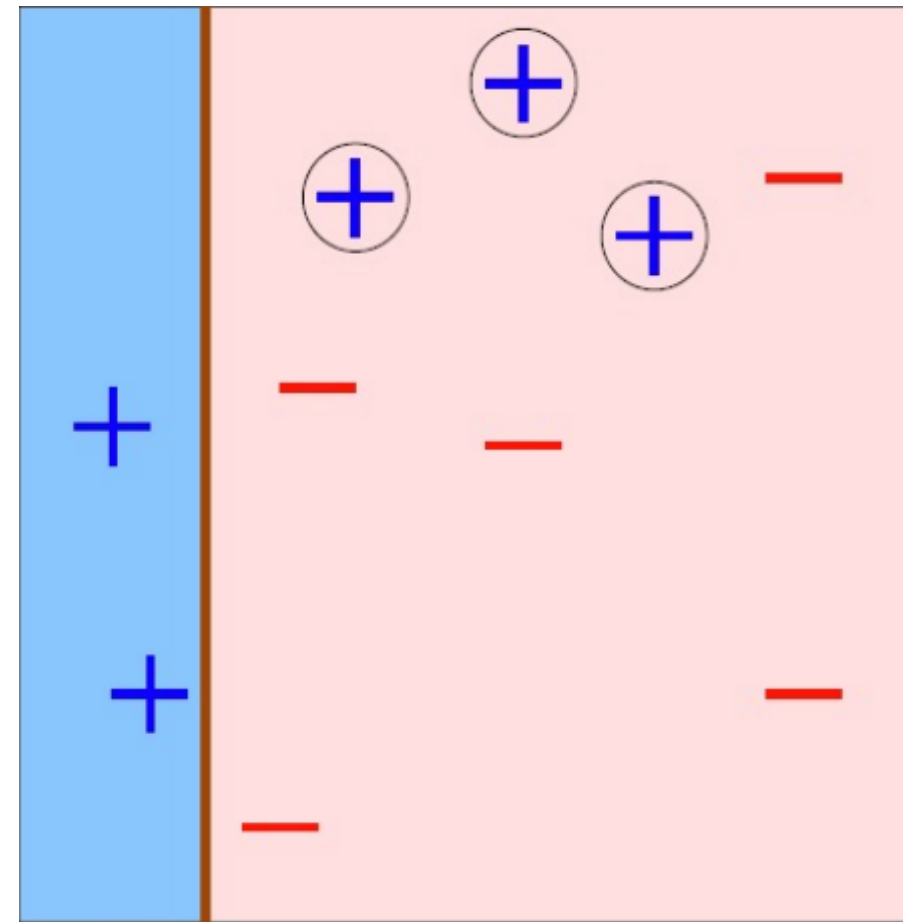
h_2



AdaBoost example

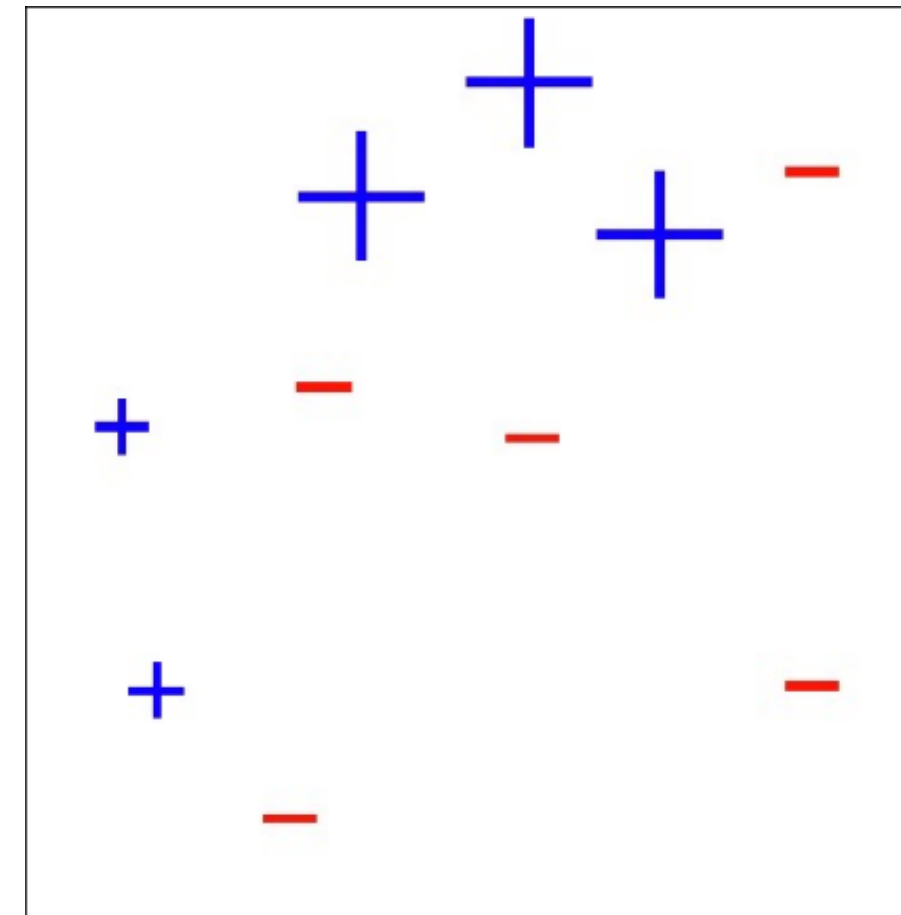
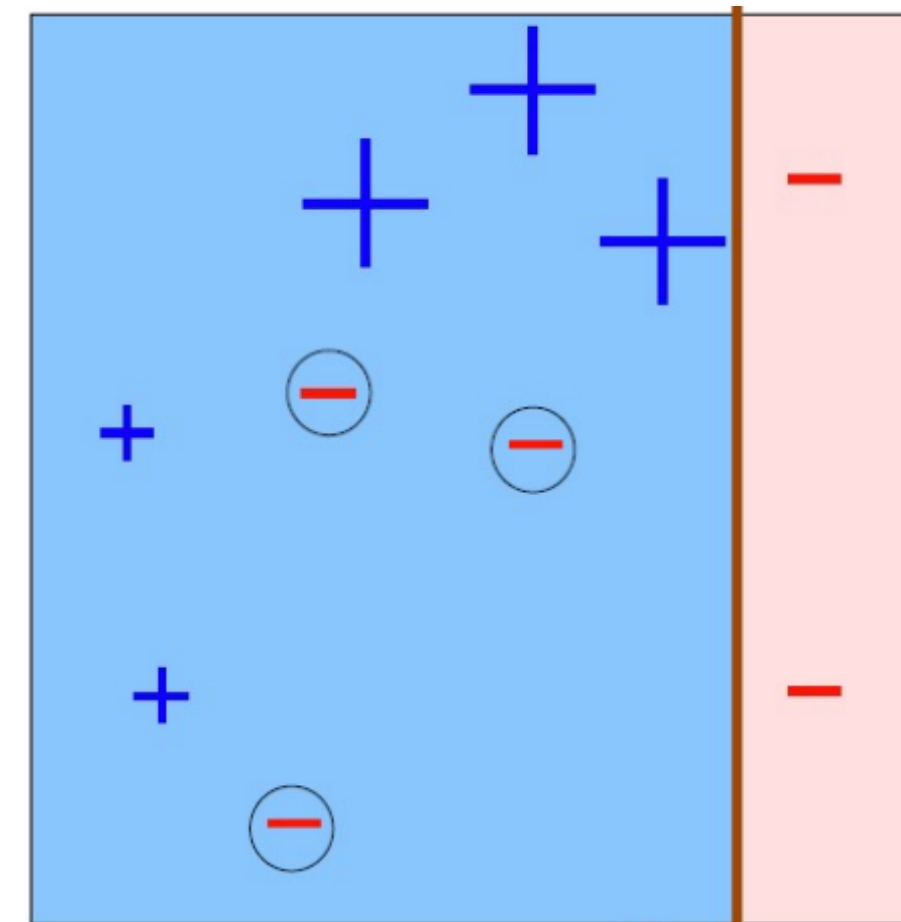
$$\epsilon_1 = 0.3$$
$$\alpha_1 = 0.42$$

h_1



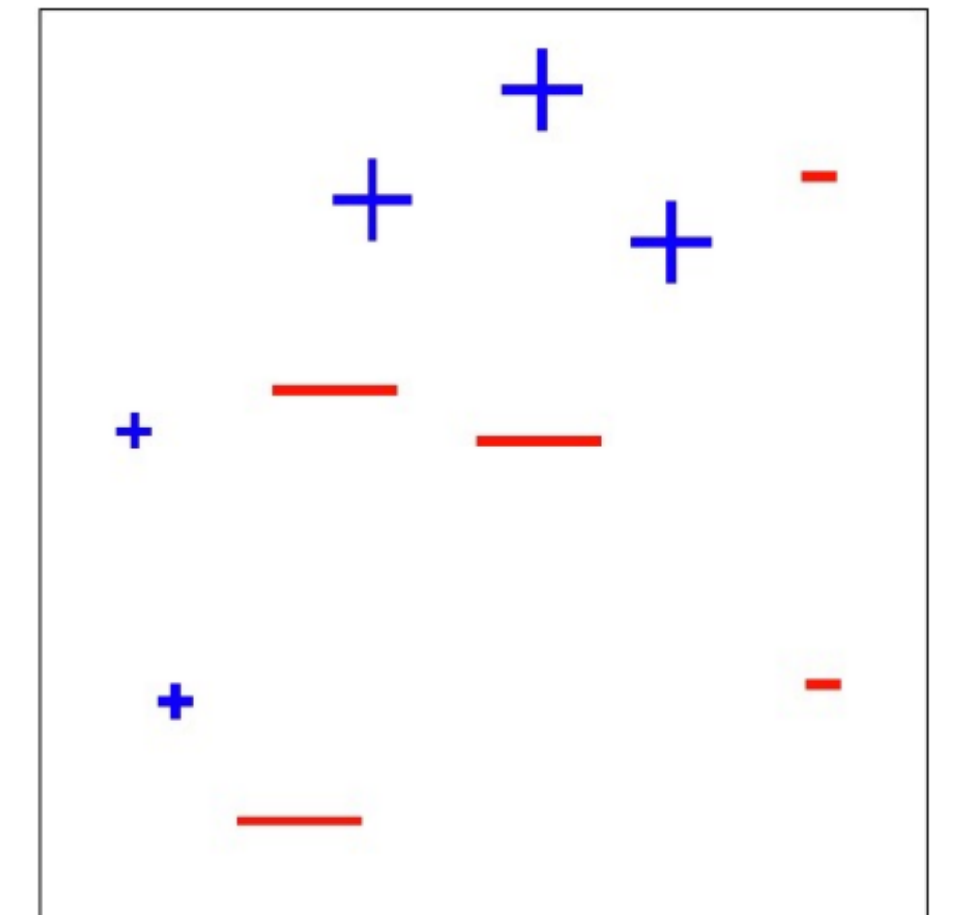
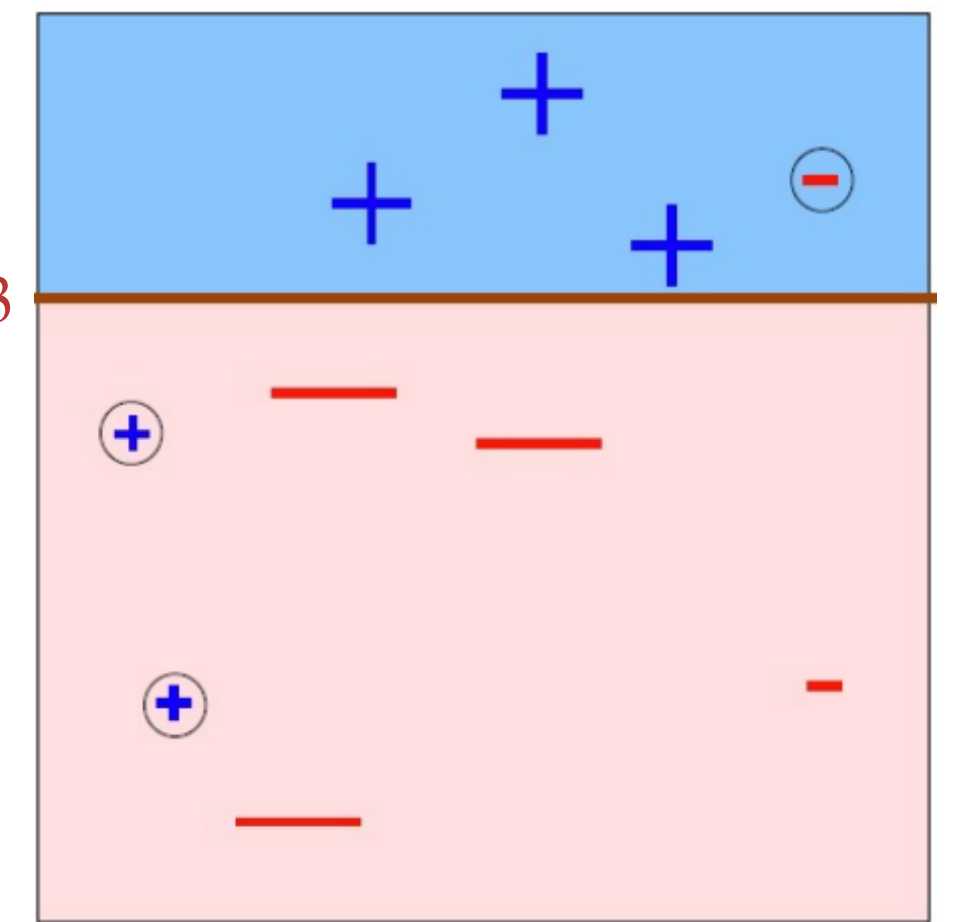
$$\epsilon_2 = 0.21$$
$$\alpha_2 = 0.65$$

h_2



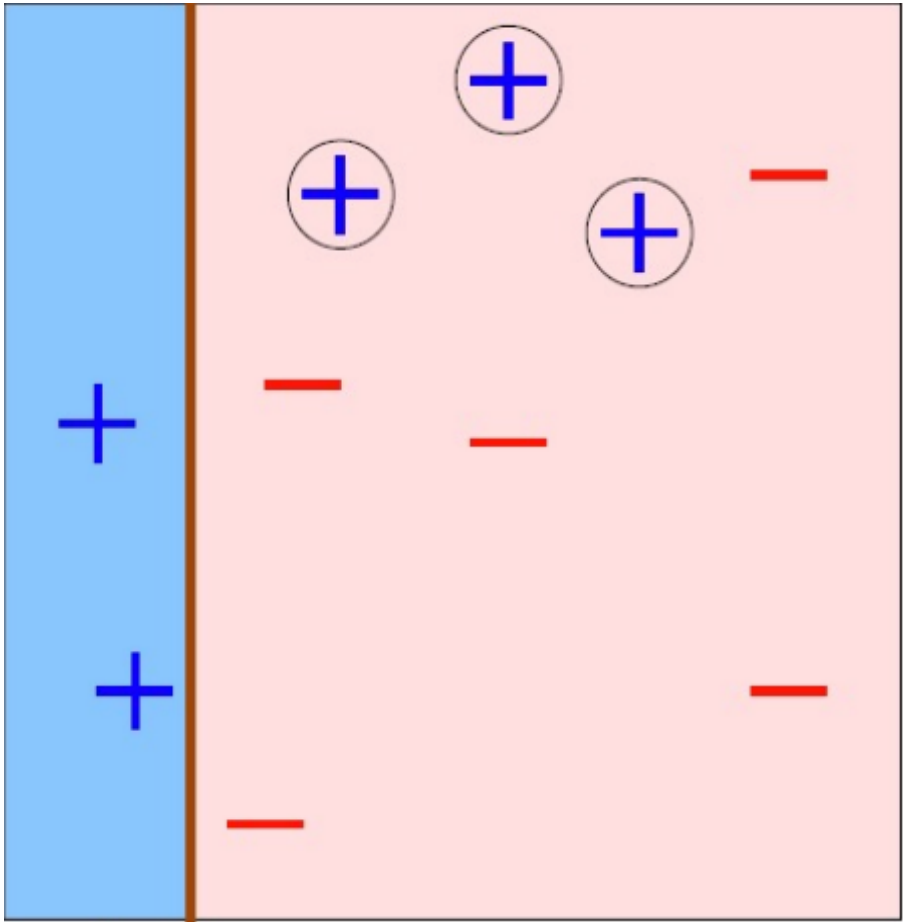
$$\epsilon_3 = 0.14$$
$$\alpha_3 = 0.92$$

h_3



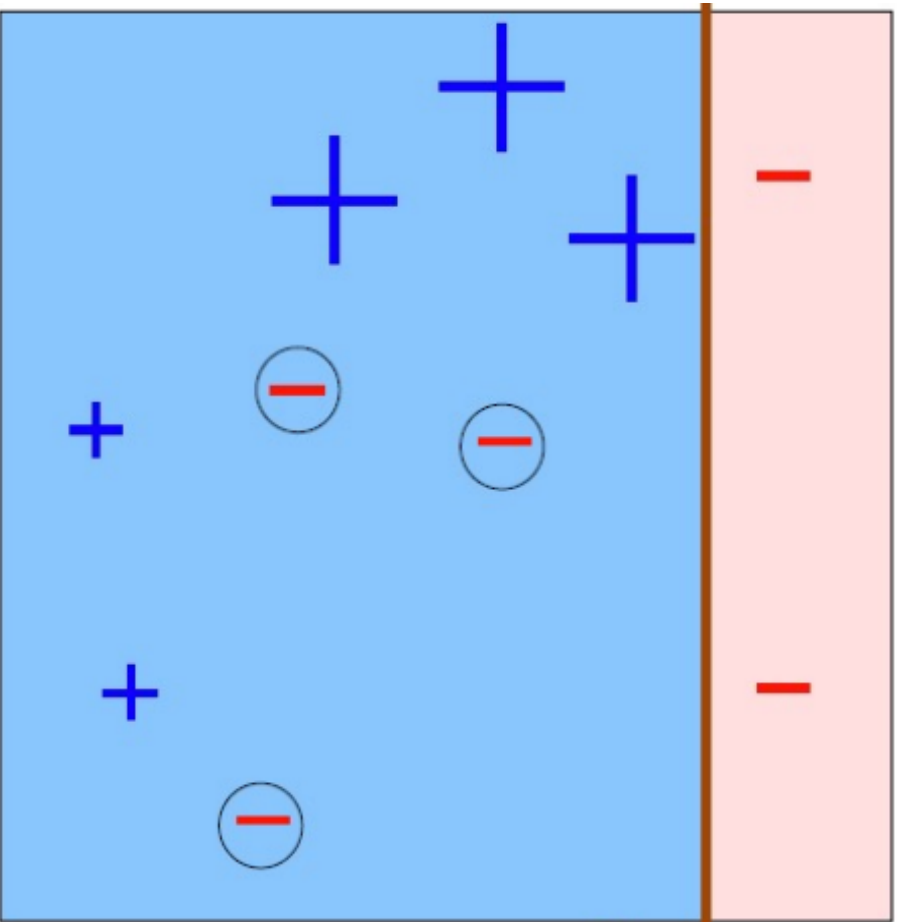
AdaBoost example

0.42 h_1



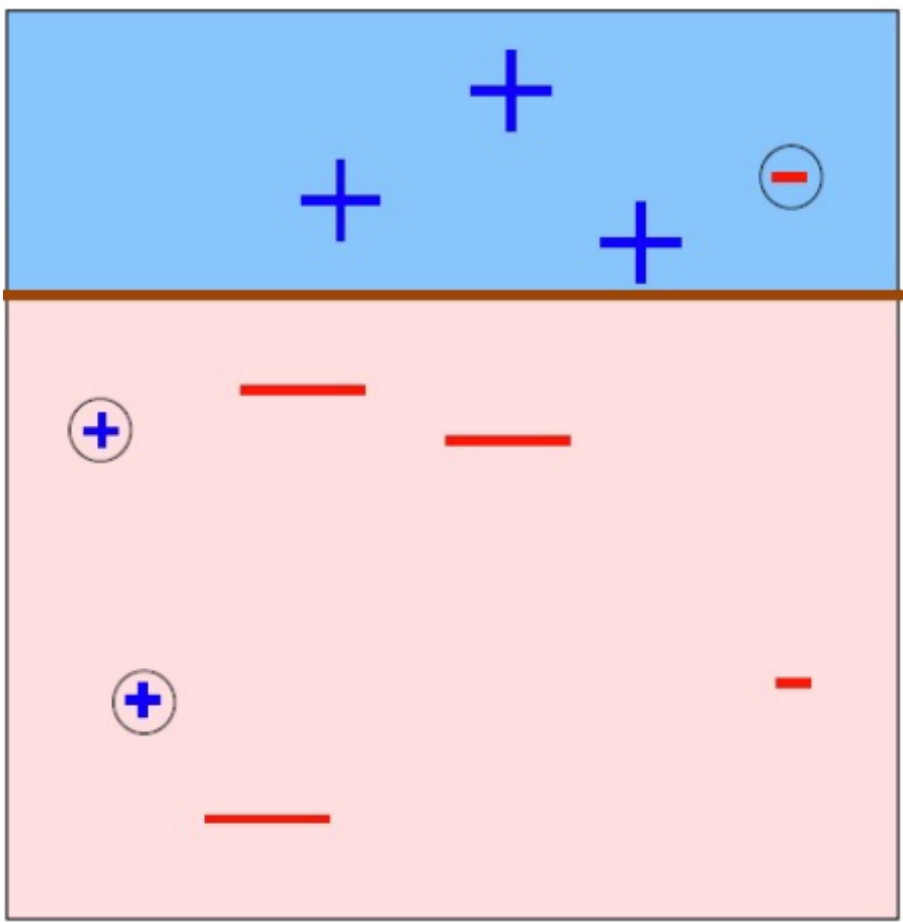
+

0.65 h_2



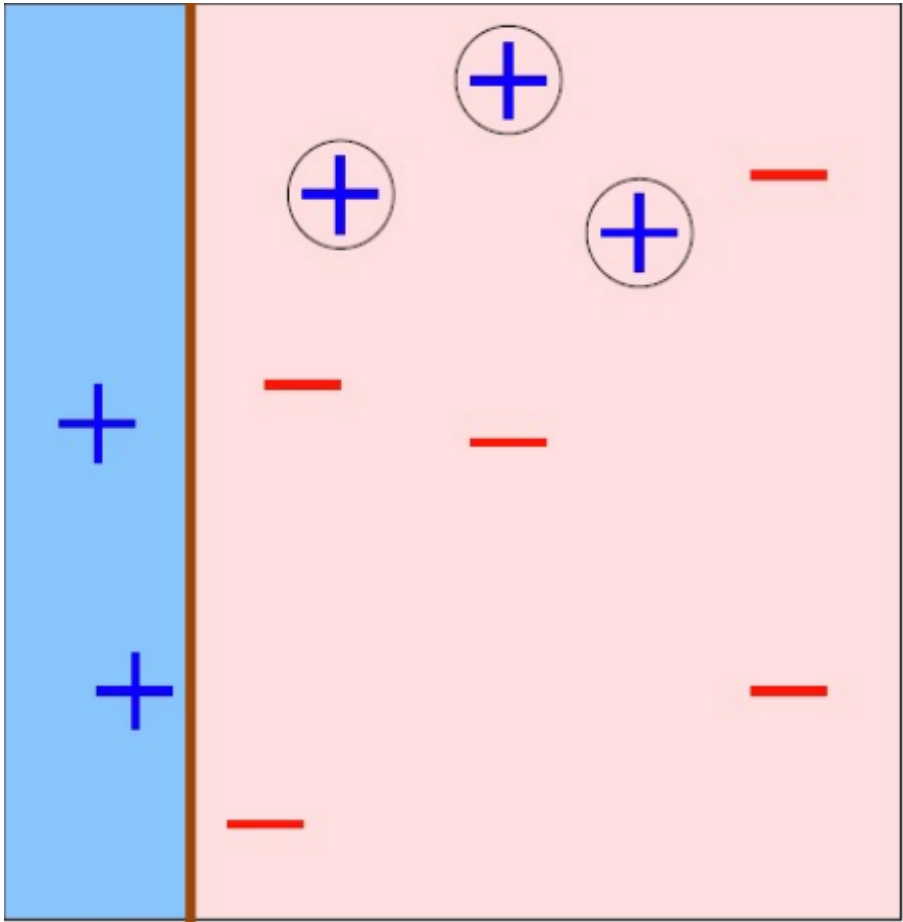
0.92
 h_3

+



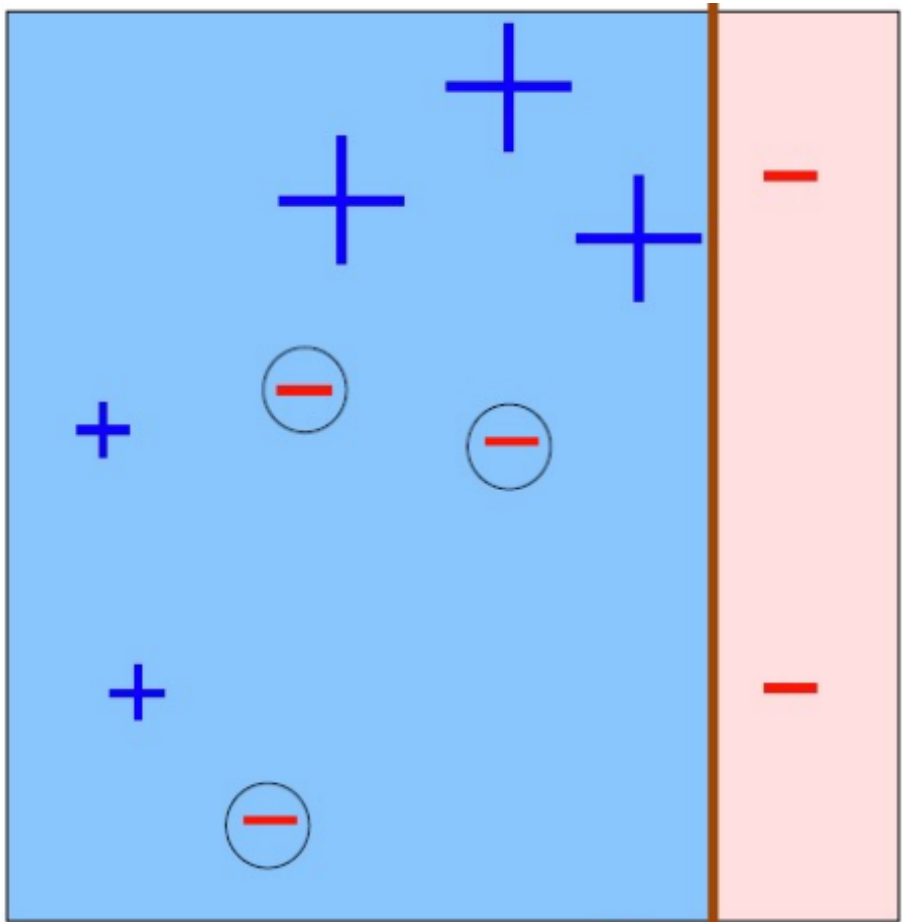
AdaBoost example

0.42 h_1



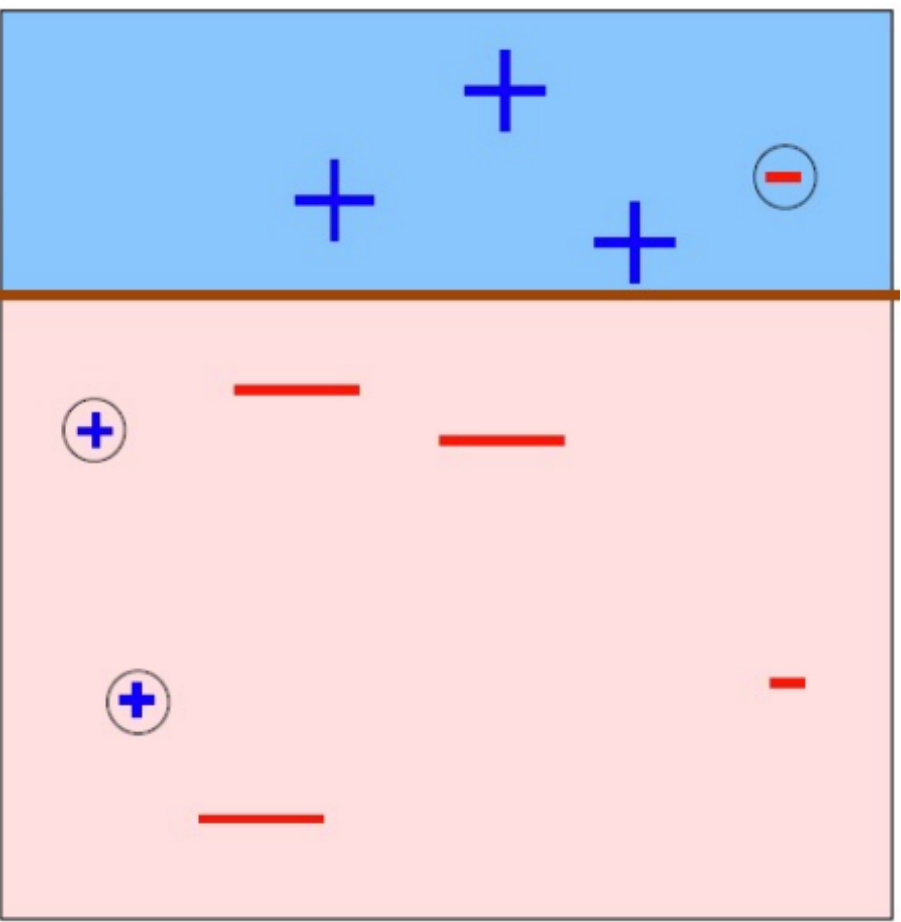
+

0.65 h_2

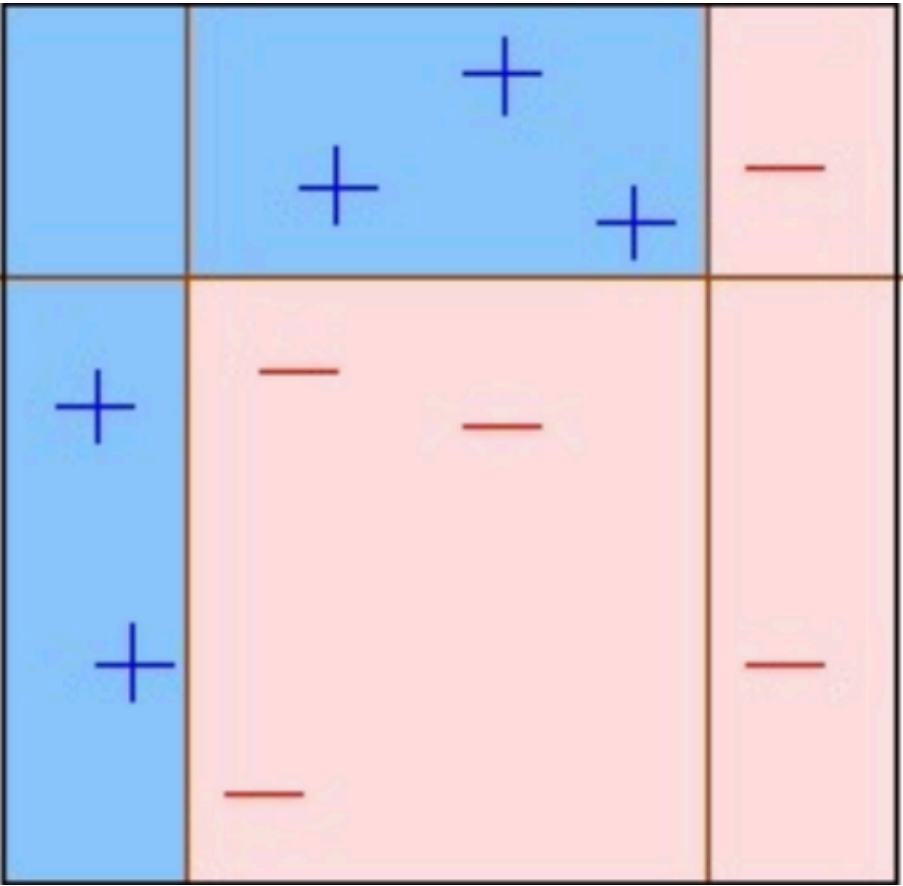


0.92
 h_3

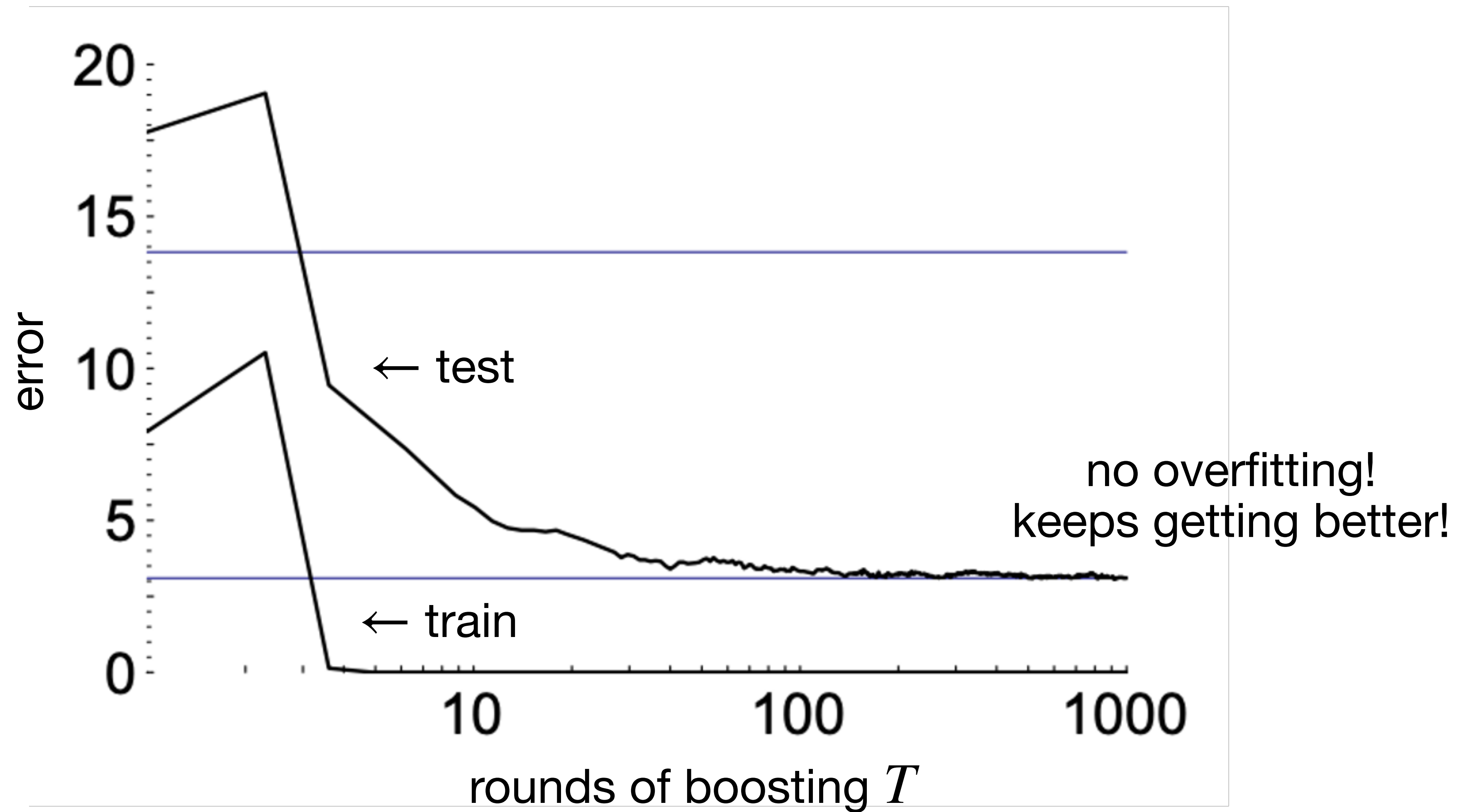
+



=



AdaBoost experiment



Source: <http://rob.schapire.net/papers/msri.pdf>