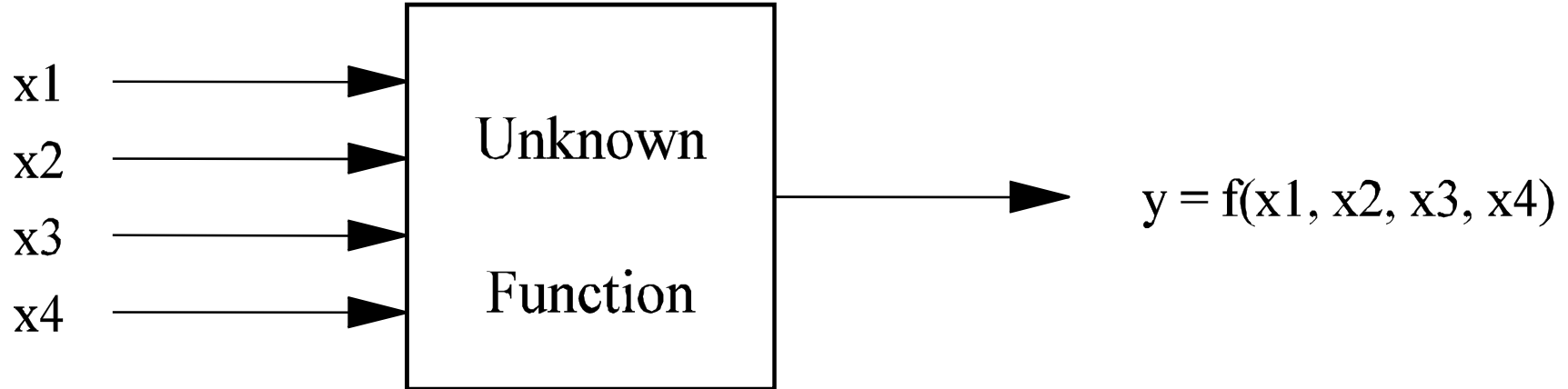

A few methods for learning binary classifiers

































































Fundamental Problem of Machine Learning: It is ill-posed



Learning Appears Impossible

- There are $2^{16} = 65536$ possible boolean functions over four input features.
- Why? Such a function is defined by $2^4 = 16$ rows. So its output column has 16 slots for answers. There are 2^{16} ways it could fill those in.








































































spam detection

x 	x 	x 	x 	y
				?
				?
				?
				?
				?
				?
				?
				?
				?
				?
				?
				?
				?
				?
				?

Learning Appears Impossible

- There are $2^{16} = 65536$ possible boolean functions over four input features.
- Why? Such a function is defined by $2^4 = 16$ rows. So its output column has 16 slots for answers. There are 2^{16} ways it could fill those in.
- We can't figure out which one is correct until we've seen every possible input-output pair.
- After 7 examples, we still have 9 slots to fill in, or 2^9 possibilities.

spam detection

x_1	x_2	x_3	x_4	y
				?
				?
				
				
				
				
				
				?
				?
				
				?
				?
				
				?
				?
				?

Solution: Work with a restricted hypothesis space

- We need to generalize from our few training examples!
- Either by applying prior knowledge or by guessing, we choose a space of hypotheses H that is smaller than the space of **all possible Boolean functions**:
 - simple conjunctive rules
 - *m-of-n* rules
 - linear functions
 - multivariate Gaussian joint probability distributions
 - etc.

Illustration: Simple Conjunctive Rules

- There are only 16 simple conjunctions (no negation)
- Try them all!
- But no simple rule explains our 7 training examples.
- The same is true for simple disjunctions.

Rule	Counterexample
$\text{true} \Leftrightarrow y$	1
$x_1 \Leftrightarrow y$	3
$x_2 \Leftrightarrow y$	2
$x_3 \Leftrightarrow y$	1
$x_4 \Leftrightarrow y$	7
$x_1 \wedge x_2 \Leftrightarrow y$	3
$x_1 \wedge x_3 \Leftrightarrow y$	3
$x_1 \wedge x_4 \Leftrightarrow y$	3
$x_2 \wedge x_3 \Leftrightarrow y$	3
$x_2 \wedge x_4 \Leftrightarrow y$	3
$x_3 \wedge x_4 \Leftrightarrow y$	4
$x_1 \wedge x_2 \wedge x_3 \Leftrightarrow y$	3
$x_1 \wedge x_2 \wedge x_4 \Leftrightarrow y$	3
$x_1 \wedge x_3 \wedge x_4 \Leftrightarrow y$	3
$x_2 \wedge x_3 \wedge x_4 \Leftrightarrow y$	3
$x_1 \wedge x_2 \wedge x_3 \wedge x_4 \Leftrightarrow y$	3

A larger hypothesis space: *m*-of-*n* rules

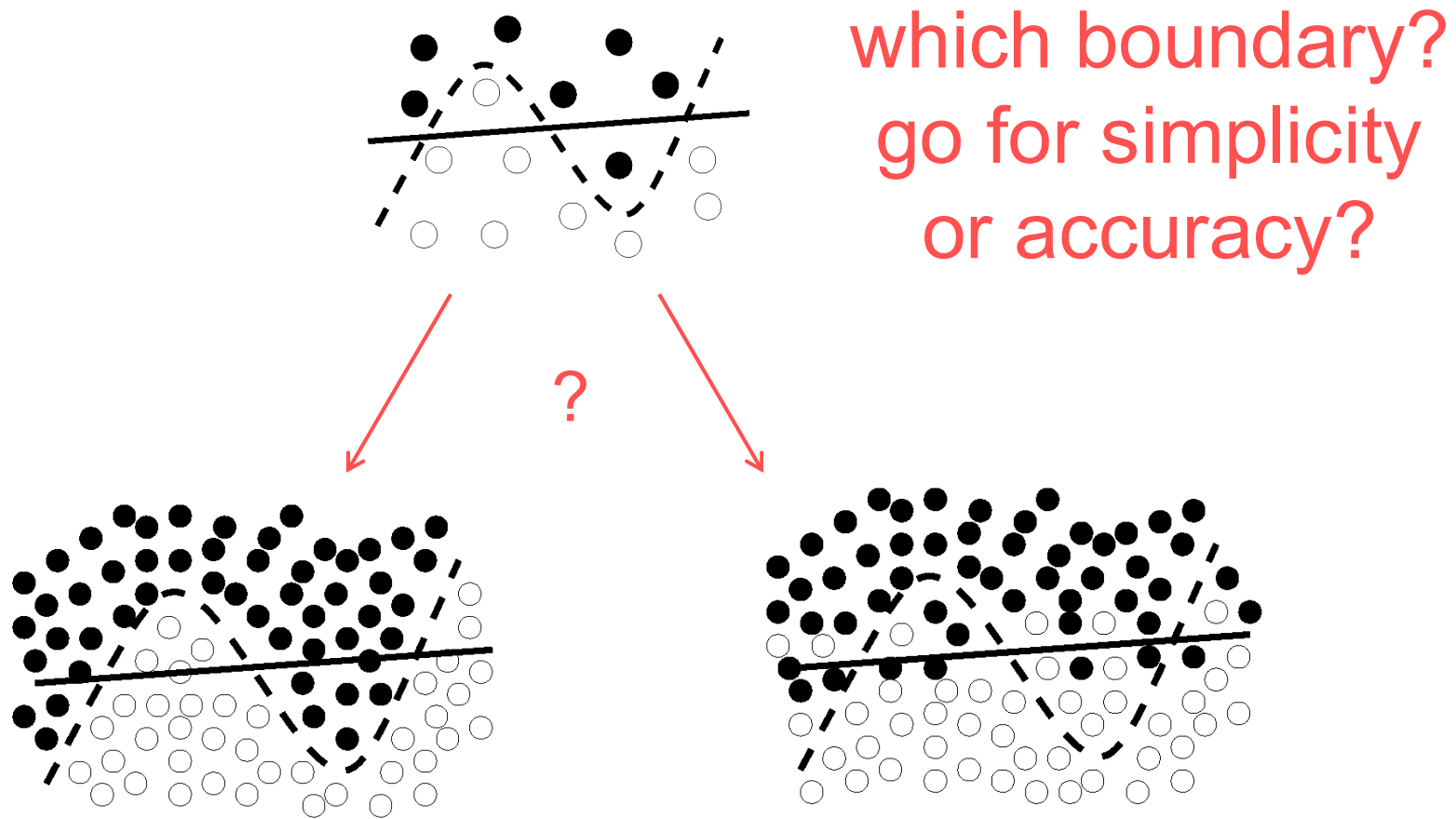
- At least *m* of *n* specified variables must be true
- There are 32 possible rules
- Only one rule is consistent!

variables	Counterexample			
	1-of	2-of	3-of	4-of
$\{x_1\}$	3	—	—	—
$\{x_2\}$	2	—	—	—
$\{x_3\}$	1	—	—	—
$\{x_4\}$	7	—	—	—
$\{x_1, x_2\}$	3	3	—	—
$\{x_1, x_3\}$	4	3	—	—
$\{x_1, x_4\}$	6	3	—	—
$\{x_2, x_3\}$	2	3	—	—
$\{x_2, x_4\}$	2	3	—	—
$\{x_3, x_4\}$	4	4	—	—
$\{x_1, x_2, x_3\}$	1	3	3	—
$\{x_1, x_2, x_4\}$	2	3	3	—
$\{x_1, x_3, x_4\}$	1	∅	3	—
$\{x_2, x_3, x_4\}$	1	5	3	—
$\{x_1, x_2, x_3, x_4\}$	1	5	3	3

Two Views of Learning

- **View 1: Learning is the removal of our remaining uncertainty about the truth**
 - Suppose we *knew* that the unknown function was an m -of- n boolean function. Then we could use the training examples to *deduce* which function it is.
- **View 2: Learning is just an engineering guess – the truth is too messy to try to find**
 - Need to pick a hypothesis class that is
 - big enough to fit the training data “well,”
 - but not so big that we overfit the data & predict test data poorly.
 - Can start with a very small class and enlarge it until it contains an hypothesis that fits the training data **perfectly**.
 - Or we could stop enlarging sooner, when there are still **some errors** on training data. (There’s a “structural risk minimization” formula for knowing when to stop! - a loose bound on the **test** data error rate.)

Balancing generalization and overfitting



more training data makes the choice more obvious

We could be wrong!

1. Multiple hypotheses in the class might fit the data
2. Our guess of the hypothesis class could be wrong
 - Within our class, the only answer was
 - “ $y = \text{true [spam]}$ iff at least 2 of $\{x_1, x_3, x_4\}$ say so”
 - But who says the right answer is an m-of-n rule at all?
 - Other hypotheses outside the class also work:
 - $y = \text{true}$ iff ... $(x_1 \text{ xor } x_3) \wedge x_4$
 - $y = \text{true}$ iff ... $x_4 \wedge \sim x_2$

Two Strategies for Machine Learning

- Use a “little language” to define a hypothesis class H that’s tailored to your problem’s structure (likely to contain a winner)
 - Then use a learning algorithm for that little language
 - Rule grammars; stochastic models (HMMs, PCFGs ...); graphical models (Bayesian nets, Markov random fields ...)
 - Dominant view in 600.465 Natural Language Processing
 - Note: Algorithms for graphical models are closely related to algorithms for constraint programming! So you’re on your way.
- Just pick a flexible, generic hypothesis class H
 - Use a standard learning algorithm for that hypothesis class
 - Decision trees; neural networks; nearest neighbor; SVMs
 - What we’ll focus on this week
 - It’s now crucial how you encode your problem as a feature vector

Memory-Based Learning

E.g., k-Nearest Neighbor

Also known as “case-based” or
“example-based” learning

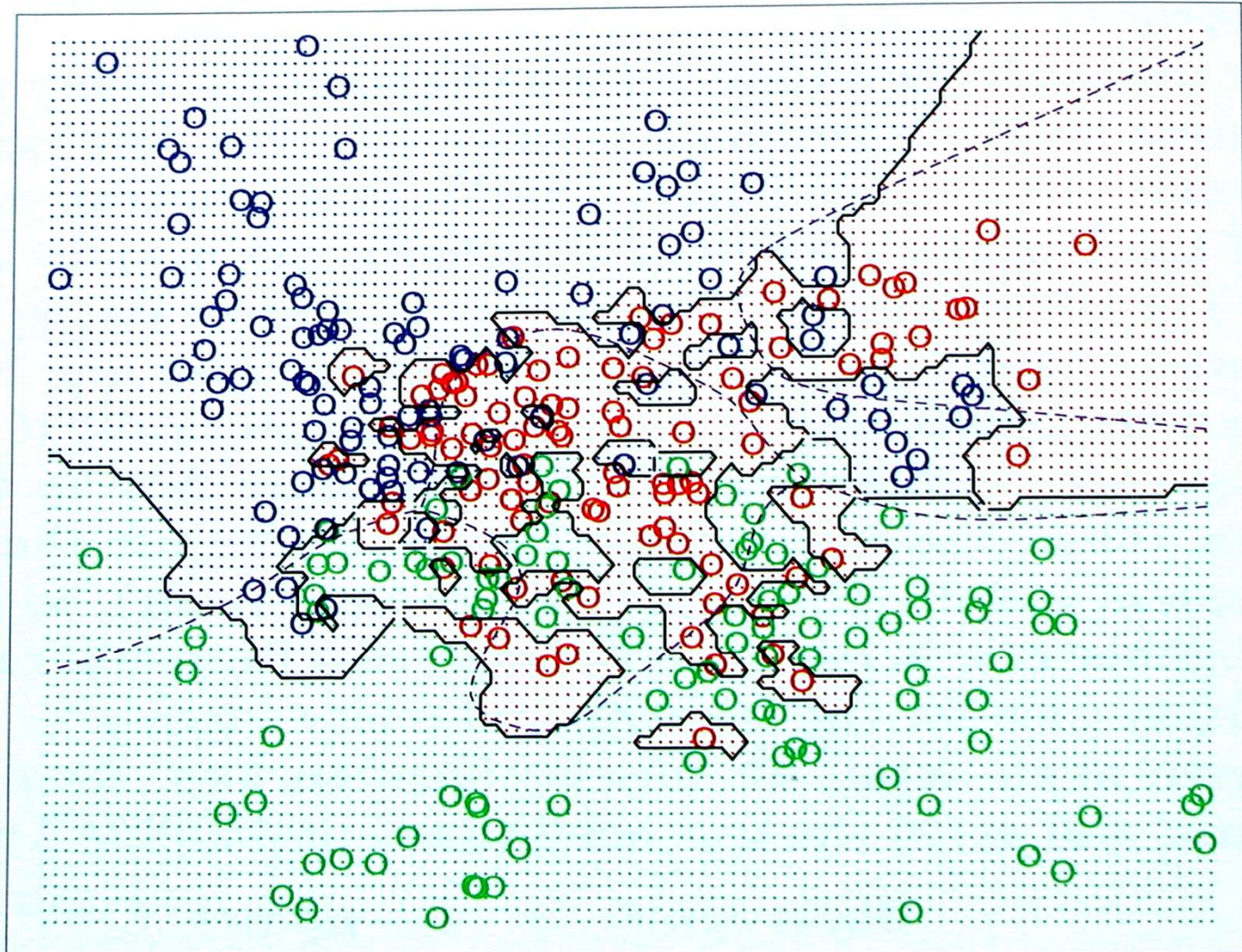
Intuition behind memory-based learning

- Similar inputs map to similar outputs
 - If not true → learning is impossible
 - If true → learning reduces to defining “*similar*”
 - Not all similarities created equal
 - guess J. D. Salinger’s weight
 - who are the similar people?
 - similar occupation, age, diet, genes, climate, ...
 - guess J. D. Salinger’s IQ
 - similar occupation, writing style, fame, SAT score, ...
 - Superficial vs. deep similarities?
 - B. F. Skinner and the behaviorism movement
- what do brains actually do?

1-Nearest Neighbor

- Define a distance $d(x_1, x_2)$ between any 2 examples
 - examples are feature vectors
 - so could just use Euclidean distance ...
- **Training:** Index the training examples for fast lookup.
- **Test:** Given a new x , find the closest x_1 from training. Classify x the same as x_1 (positive or negative)
- Can learn complex decision boundaries
- As training size $\rightarrow \infty$, error rate is at most 2x the Bayes-optimal rate (i.e., the error rate you'd get from knowing the true model that generated the data – whatever it is!)

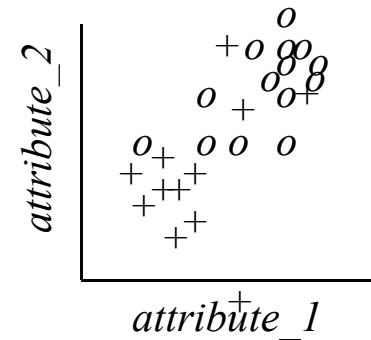
1-Nearest Neighbor – decision boundary



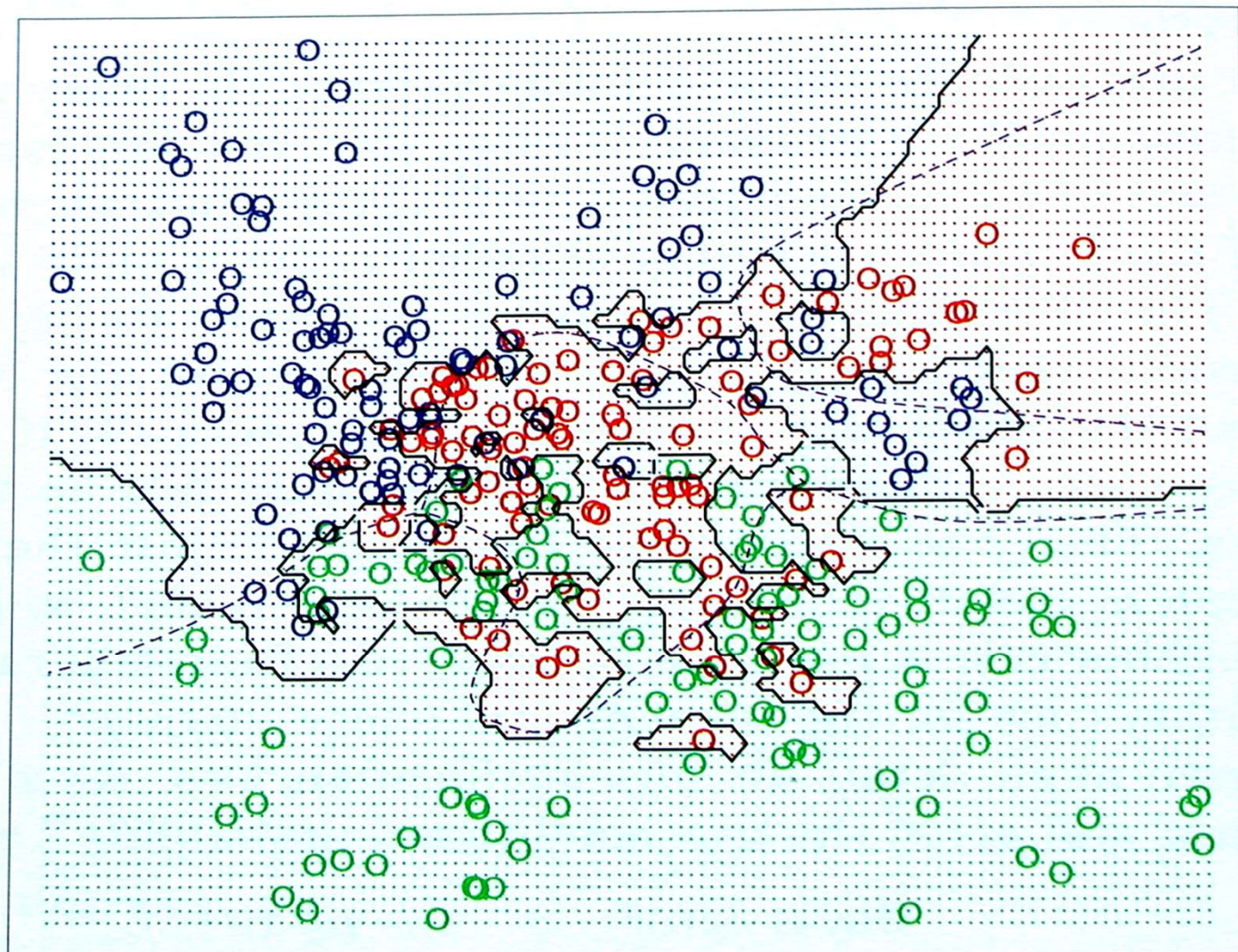
slide thanks to Rich Caruana (modified)

k-Nearest Neighbor

- Instead of picking just the single nearest neighbor, pick the k nearest neighbors and have them vote
- Average of k points more reliable when:
 - noise in training vectors x
 - noise in training labels y
 - classes partially overlap



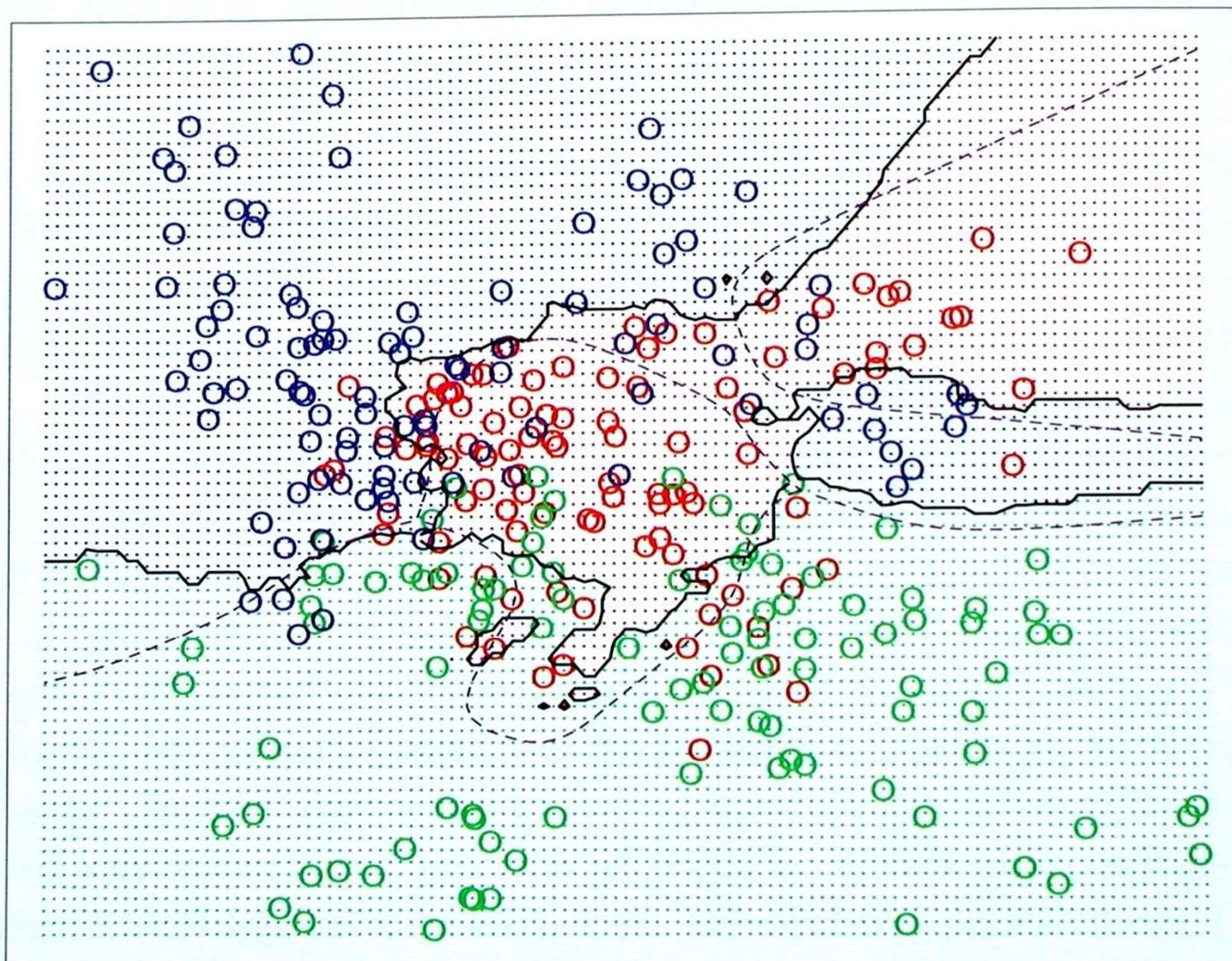
1 Nearest Neighbor – decision boundary



From Hastie, Tibshirani, Friedman 2001 p418

slide thanks to Rich Caruana (modified)

15 Nearest Neighbors – it's smoother!



From Hastie, Tibshirani, Friedman 2001 p418

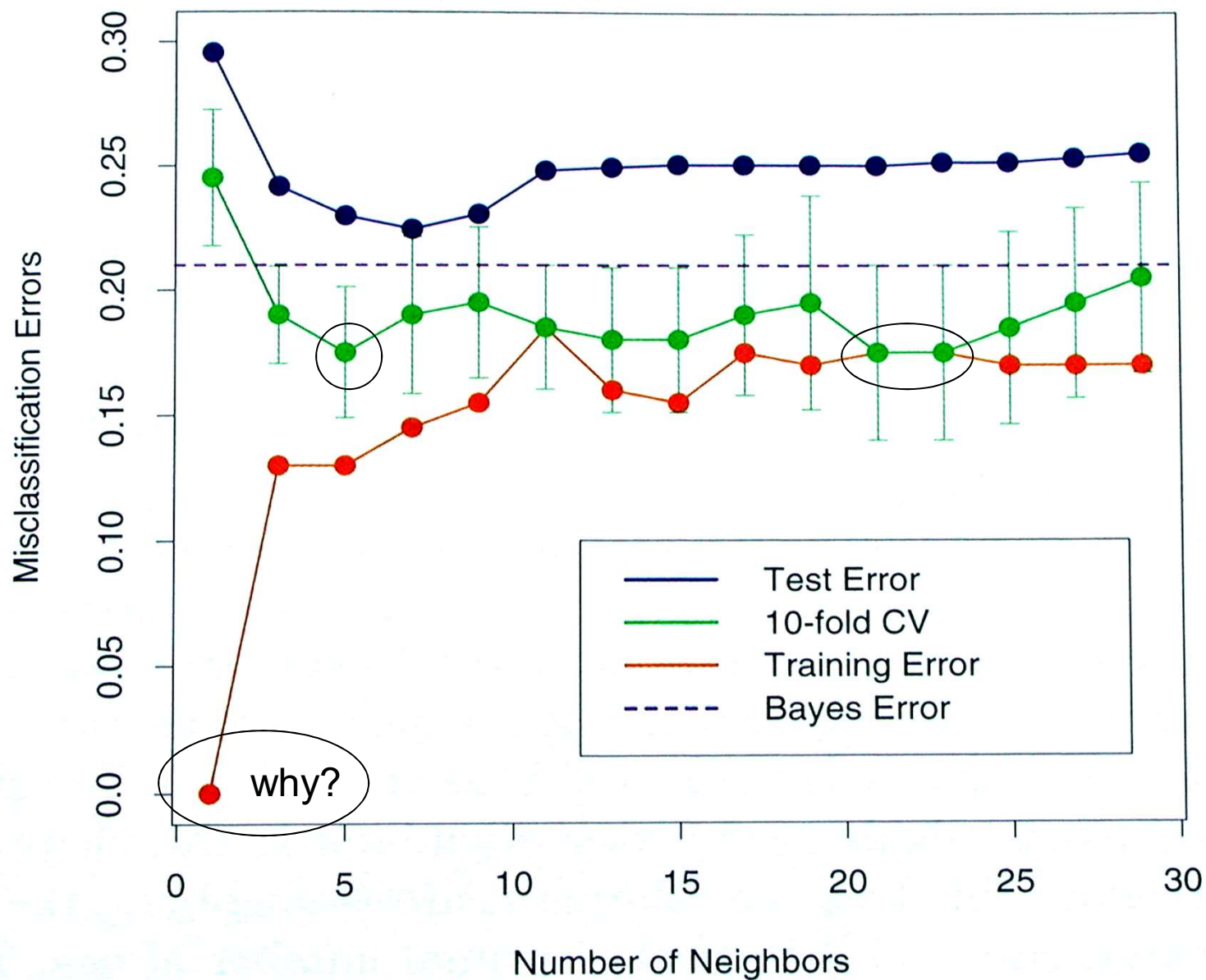
600.525/425 Declarative Methods - J. Elshet

slide thanks to Rich Caruana (modified)

How to choose “k”

- Odd k (often 1, 3, or 5):
 - Avoids problem of breaking ties (in a binary classifier)
- Large k:
 - less sensitive to noise (particularly class noise)
 - better probability estimates for discrete classes
 - larger training sets allow larger values of k
- Small k:
 - captures fine structure of problem space better
 - may be necessary with small training sets
- Balance between large and small k
 - What does this remind you of?
- As training set approaches infinity, and k grows large, kNN becomes Bayes optimal

From Hastie, Tibshirani, Friedman 2001 p419



Cross-Validation

- Models usually perform better on training data than on future test cases
- 1-NN is 100% accurate on training data!
- “Leave-one-out” cross validation:
 - “remove” each case one-at-a-time
 - use as test case with remaining cases as train set
 - average performance over all test cases
- LOOCV is impractical with most learning methods, but extremely efficient with MBL!

Distance-Weighted kNN

- hard to pick large vs. small k
 - may not even want k to be constant
- use large k , but more emphasis on nearer neighbors?

$$\text{prediction}(x) = \frac{\sum_{i=1}^k w_i \cdot y_i}{\sum_{i=1}^k w_i}$$

where x_1, \dots, x_k are the k - NN and y_1, \dots, y_k their labels

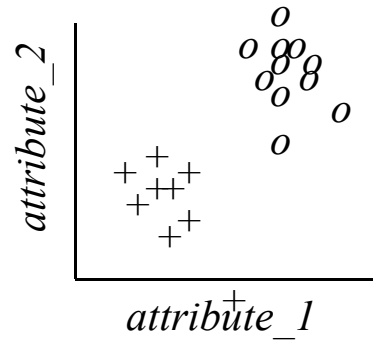
We define relative weights for the k - NN :

$$w_i = \frac{1}{\text{Dist}(x_i, x)} \text{ or maybe } \frac{1}{\text{Dist}(x_i, x)^\beta} \text{ or often } \frac{1}{\exp \beta \cdot \text{Dist}(x_i, x)}$$

Combining k-NN with other methods, #1

- Instead of having the k-NN simply vote, put them into a little machine learner!
- To classify x , train a “local” classifier on its k nearest neighbors (maybe weighted).
 - polynomial, neural network, ...

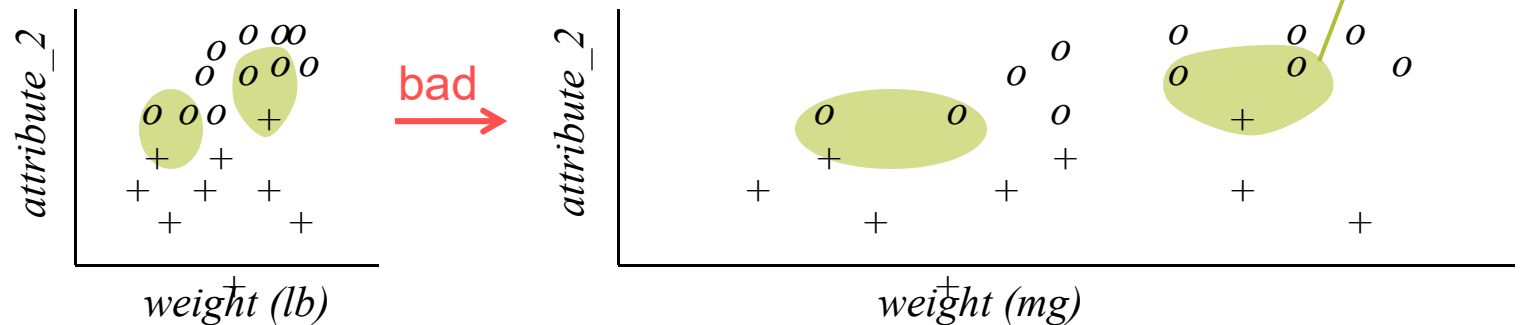
Now back to that distance function



- Euclidean distance treats all of the input dimensions as equally important

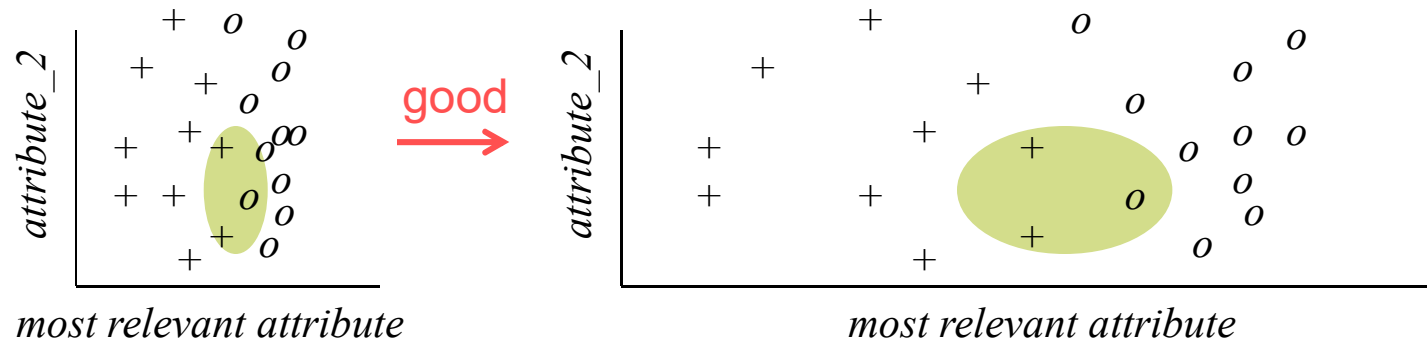
These o's are now "closer" to + than to each other ☹️

Now back to that distance function



- Euclidean distance treats all of the input dimensions as equally important
- Problem #1:
 - What if the input represents physical weight not in pounds but in milligrams?
 - Then small differences in physical weight dimension have a huge effect on distances, overwhelming other features.
 - Should really correct for these arbitrary “scaling” issues.
 - One simple idea: rescale weights so that standard deviation = 1.

Now back to that distance function



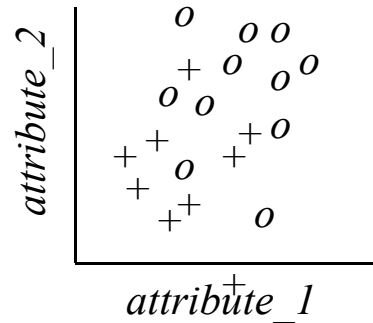
- Euclidean distance treats all of the input dimensions as equally important
- Problem #2:
 - What if some dimensions more **correlated** with true label?
 - (more relevant, or less noisy)
 - Stretch those dimensions out so that they are more important in determining distance.
 - One common technique is called “information gain.”

Weighted Euclidean Distance

$$d(x, x') = \sqrt{\sum_{i=1}^N s_i \cdot (x_i - x'_i)^2}$$

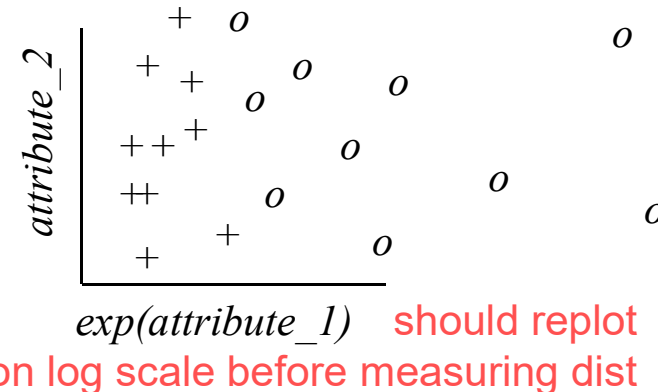
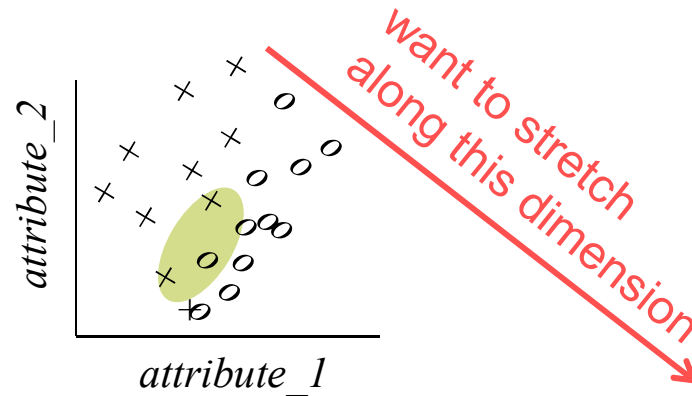
- large weight s_i \rightarrow attribute i is more important
- small weight s_i \rightarrow attribute i is less important
- zero weight s_i \rightarrow attribute i doesn't matter

Now back to that distance function



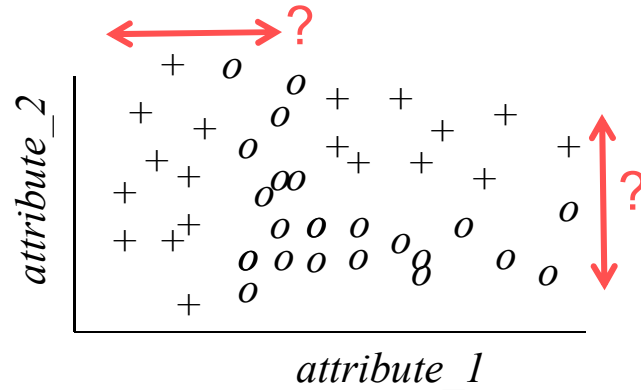
- Euclidean distance treats all of the input dimensions as equally important
- Problem #3:
 - Do we really want to decide separately and theoretically how to scale each dimension?
 - Could simply pick dimension scaling factors to maximize performance on development data. (maybe do leave-one-out)
 - Similarly, pick number of neighbors k and how to weight them.
 - Especially useful if performance measurement is complicated (e.g., 3 classes and differing misclassification costs).

Now back to that distance function



- Euclidean distance treats all of the input dimensions as equally important
 - Problem #4:
 - Is it the original input dimensions that we want to scale?
 - What if the true clusters run diagonally? Or curve?
 - We can transform the data first by extracting a different, useful set of features from it:
 - Linear discriminant analysis
 - Hidden layer of a neural network
- i.e., redescribe the data by how a **different** type of learned classifier internally sees it

Now back to that distance function



- Euclidean distance treats all of the input dimensions as equally important
- Problem #5:
 - Do we really want to transform the data globally?
 - What if different regions of the data space behave differently?
 - Could find 300 “nearest” neighbors (using *global* transform), then *locally* transform that subset of the data to redefine “near”
 - Maybe could use decision trees to split up the data space first

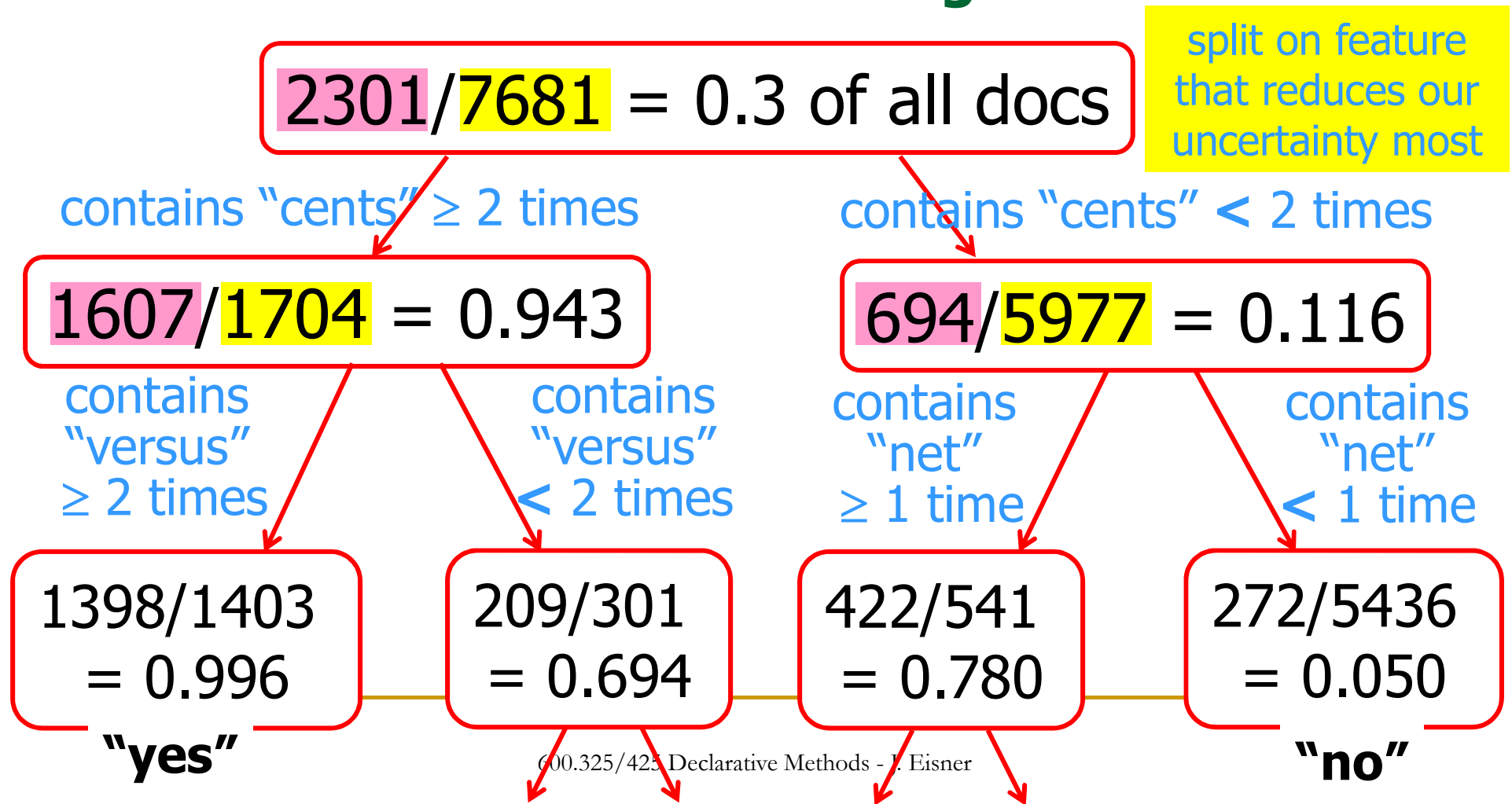
Why are we doing all this preprocessing?

- Shouldn't the user figure out a smart way to transform the data before giving it to k-NN?
- Sure, that's always good, but what will the user try?
 - Probably a lot of the same things we're discussing.
 - She'll stare at the training data and try to figure out how to transform it so that close neighbors tend to have the same label.
 - To be nice to her, we're trying to automate the most common parts of that process – like scaling the dimensions appropriately.
 - We may still miss patterns that her visual system or expertise can find. So she may still want to transform the data.
 - On the other hand, we may find patterns that would be hard for her to see.

Tangent: Decision Trees

(a different simple method)

Is this Reuters article an Earnings Announcement?



Booleans, Nominals, Ordinals, and Reals

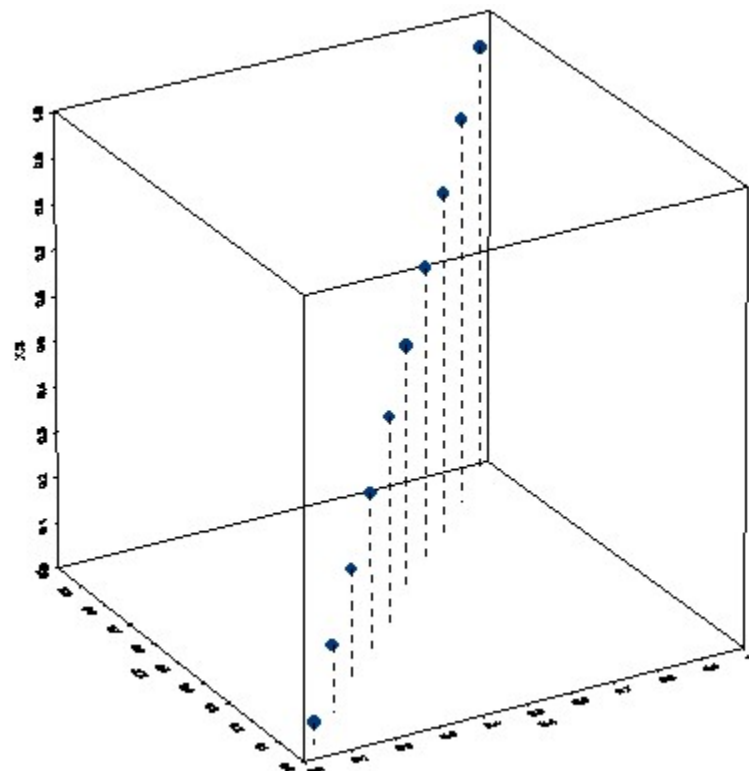
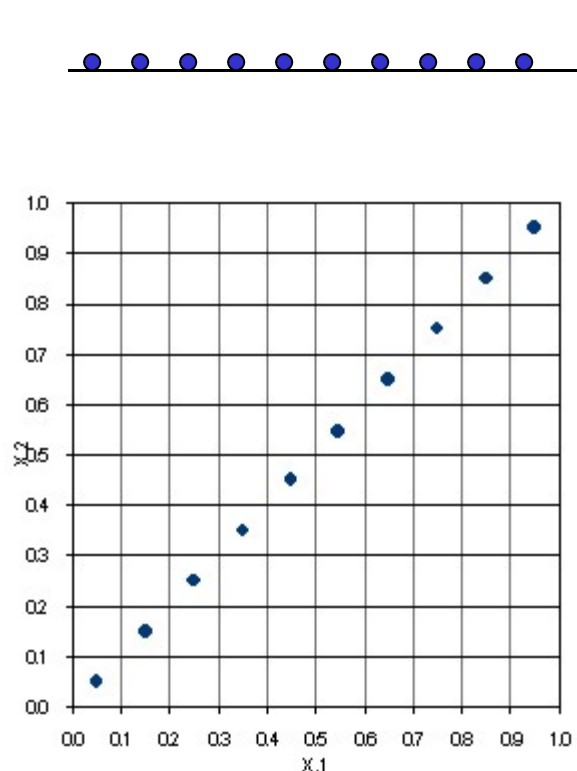
- Consider attribute value differences:

$(x_i - x'_i)$: what does subtraction do?

- Reals: easy! full continuum of differences
- Integers: not bad: discrete set of differences
- Ordinals: not bad: discrete set of differences
- Booleans: awkward: hamming distances 0 or 1
- Nominals? not good! recode as Booleans?

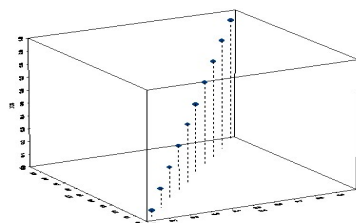
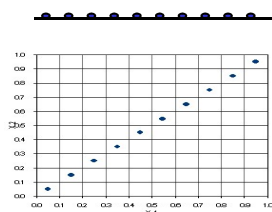
“Curse of Dimensionality”

- Pictures on previous slides showed 2-dimensional data
- What happens with lots of dimensions?
- 10 training samples cover the space less & less well ...



“Curse of Dimensionality”

- Pictures on previous slides showed 2-dimensional data
- What happens with lots of dimensions?
- 10 training samples cover the space less & less well ...



- A deeper perspective on this:
 - Random points chosen in a high-dimensional space tend to all be pretty much equidistant from one another!
 - (Proof: in 1000 dimensions, the squared distance between two random points is a sample variance of 1000 coordinate distances. Since 1000 is large, this sample variance is usually close to the true variance.)
 - So each test example is about **equally** close to **most** training examples.
 - We need a lot of training examples to expect one that is **unusually** close to the test example.

“Curse of Dimensionality”

- also, with lots of dimensions/attributes/features, the irrelevant ones may overwhelm the relevant ones:

$$d(x, x') = \sqrt{\sum_{i=1}^{relevant} (x_i - x'_i)^2 + \sum_{j=1}^{irrelevant} (x_j - x'_j)^2}$$

- So the ideas from previous slides grow in importance:
 - feature weights (scaling)
 - feature selection (try to identify & discard irrelevant features)
 - but with lots of features, some irrelevant ones will probably accidentally look relevant on the training data
 - smooth by allowing more neighbors to vote (e.g., larger k)

Advantages of Memory-Based Methods

- Lazy learning: don't do any work until you know what you want to predict (and from what variables!)
 - never need to learn a global model
 - many simple local models taken together can represent a more complex global model
- Learns arbitrarily complicated decision boundaries
- Very efficient cross-validation
- Easy to explain to users how it works
 - ... and why it made a particular decision!
- Can use **any** distance metric: string-edit distance, ...
 - handles missing values, time-varying distributions, ...

Weaknesses of Memory-Based Methods

- Curse of Dimensionality
 - often works best with 25 or fewer dimensions
- Classification runtime scales with training set size
 - clever indexing may help (K-D trees? locality-sensitive hash?)
 - large training sets will not fit in memory
- Sometimes you wish NN stood for “neural net” instead of “nearest neighbor” 😊
 - Simply averaging nearby training points isn’t very subtle
 - Naive distance functions are overly respectful of the input encoding
- For regression (predict a number rather than a class), the extrapolated surface has discontinuities

Current Research in MBL

- Condensed representations to reduce memory requirements and speed-up neighbor finding to scale to 10^6 – 10^{12} cases
- Learn better distance metrics
- Feature selection
- Overfitting, VC-dimension, ...
- MBL in higher dimensions
- MBL in non-numeric domains:
 - Case-Based or Example-Based Reasoning
 - Reasoning by Analogy

References

- *Locally Weighted Learning* by Atkeson, Moore, Schaal
- *Tuning Locally Weighted Learning* by Schaal, Atkeson, Moore

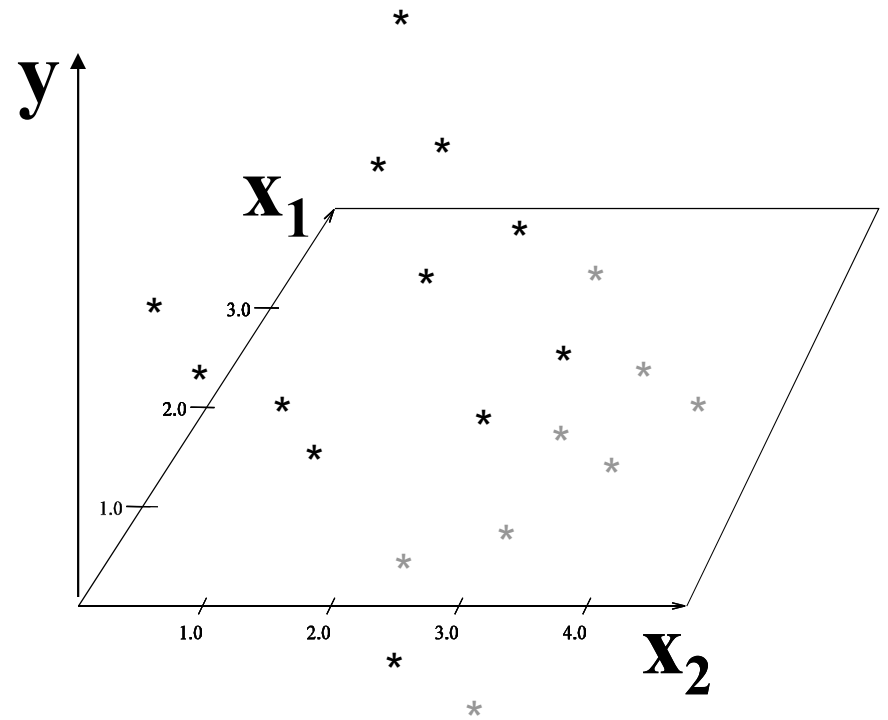
Closing Thought

- In many supervised learning problems, all the information you ever have about the problem is in the training set.
- Why do most learning methods discard the training data after doing learning?
- Do neural nets, decision trees, and Bayes nets capture *all* the information in the training set when they are trained?
- Need more methods that combine MBL with these other learning methods.
 - to improve accuracy
 - for better explanation
 - for increased flexibility

Linear Classifiers

Linear regression – standard statistics

- As usual, input is a vector x
- Output is a number $y=f(x)$

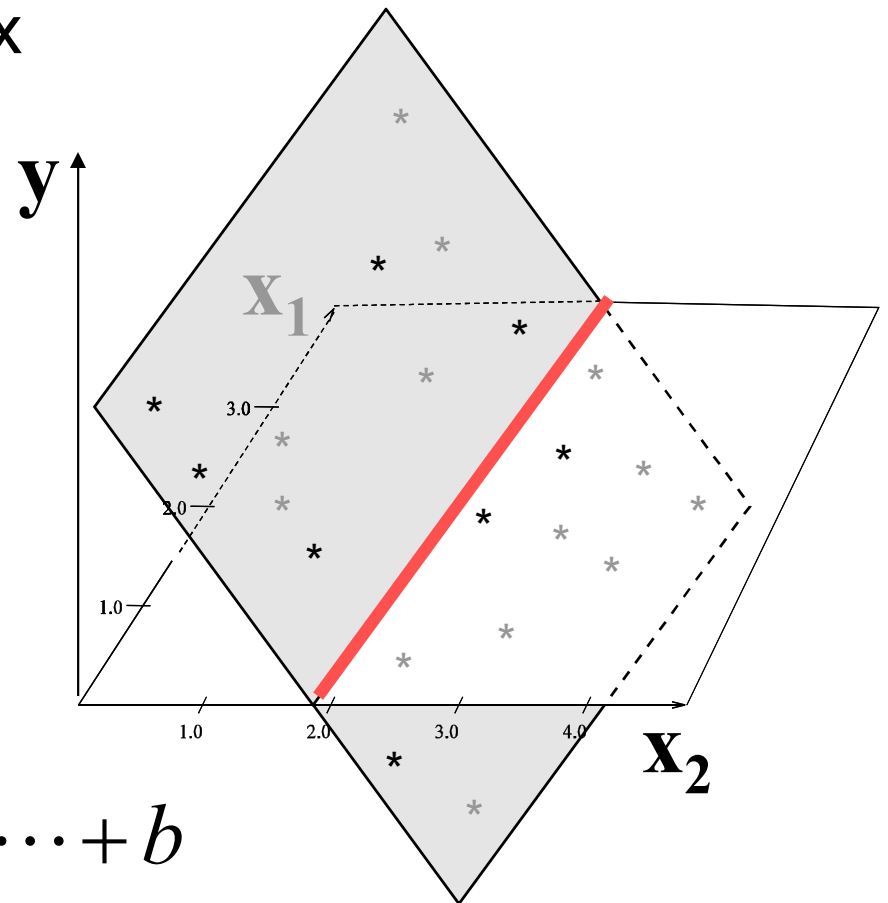


Linear regression – standard statistics

- As usual, input is a vector x
- Output is a number $y=f(x)$
- **Linear** regression:

$$y = \vec{w} \cdot \vec{x} + b$$
$$= \sum_i w_i x_i + b$$

$$= w_1 x_1 + w_2 x_2 + w_3 x_3 + \dots + b$$



Linear classification

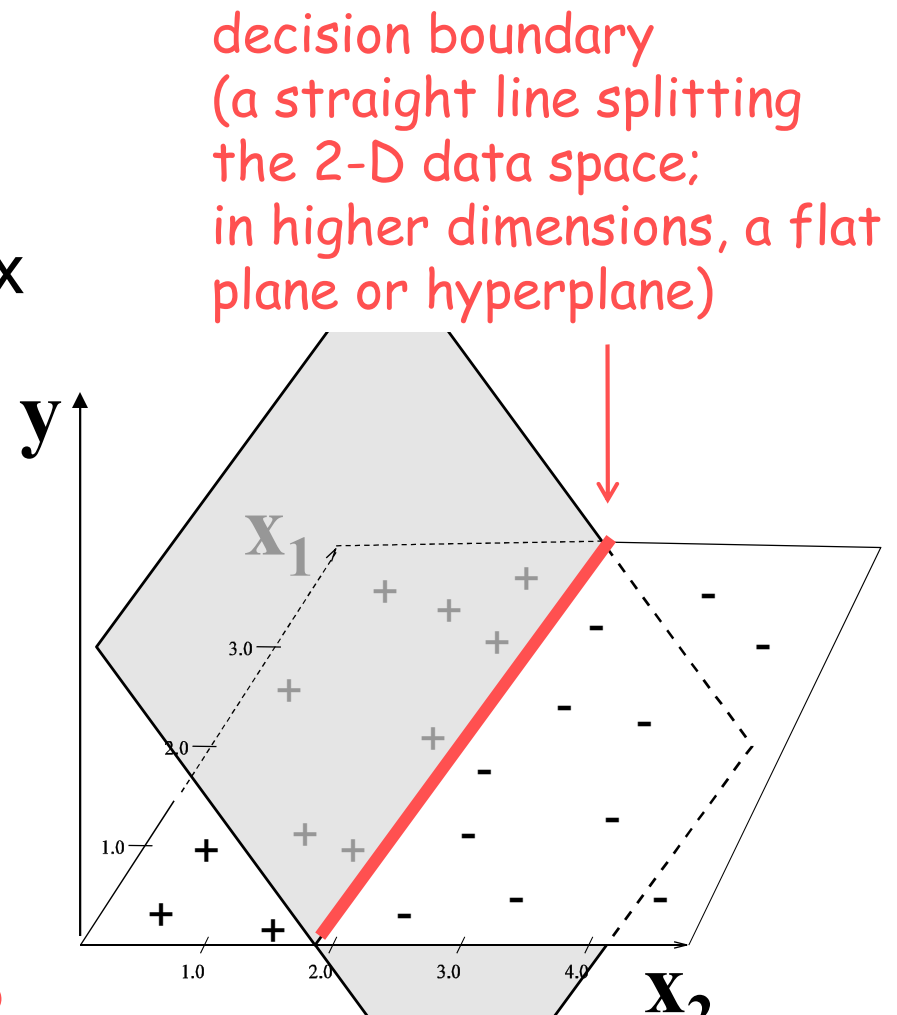
- As usual, input is a vector x
- Output is a class $y \in \{-, +\}$
- **Linear** classification:

$$y = + \quad \text{if} \quad \vec{w} \cdot \vec{x} + b > 0$$

$$y = - \quad \text{if} \quad \vec{w} \cdot \vec{x} + b < 0$$

weight vector w

threshold b
(since classifier asks: does $w \cdot x$ exceed $-b$, crossing a threshold?
 b often called "bias term," since adjusting it will bias the classifier toward picking $+$ or $-$)



Simplify the notation: Eliminate b

(just another reduction: problem may look easier without b , but isn't)

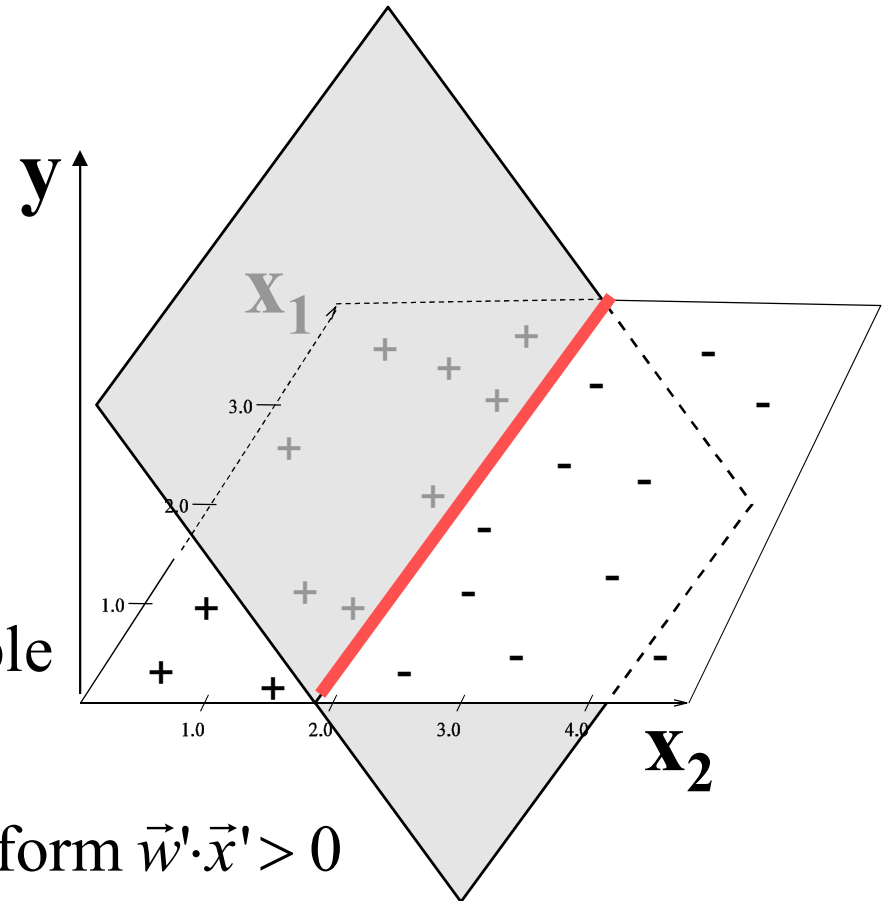
Rewrite $\vec{w} \cdot \vec{x} + b > 0$

as $(b, \vec{w}) \cdot (1, \vec{x}) > 0$

call this \vec{w}'

call this \vec{x}'

(so w'_0 is the bias term)



In other words, replace each example

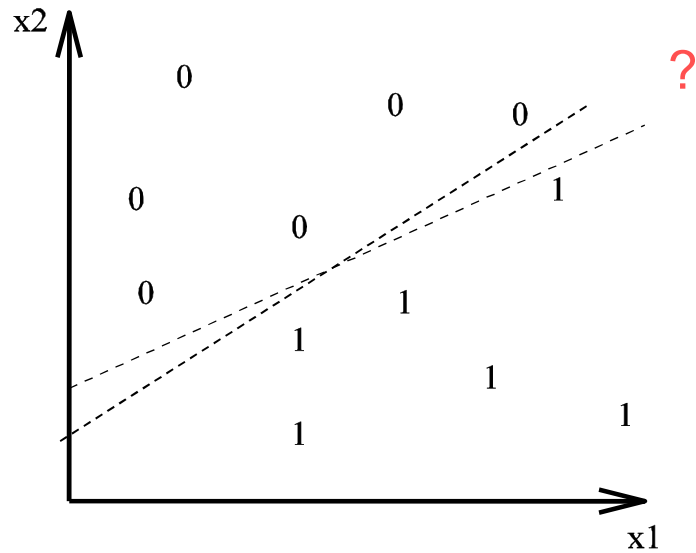
$\vec{x} = (x_1, x_2, \dots)$ with $\vec{x}' = (1, x_1, x_2, \dots)$

and then look for a classifier of the form $\vec{w}' \cdot \vec{x}' > 0$

Training a linear classifier

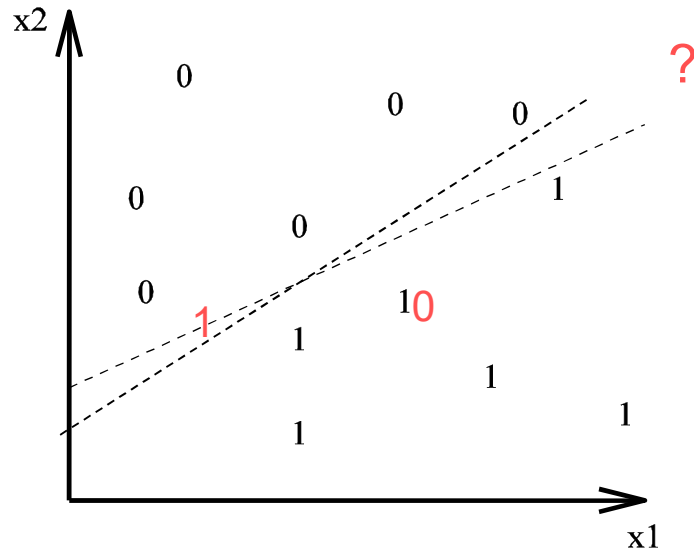
- Given some supervised training data (usually high-dimensional)
- What is the **best** linear classifier (defined by weight vector w')?
- Surprisingly, lots of algorithms!
- Three cases to think about:
 1. The training data are **linearly separable**
(\exists a hyperplane that perfectly divides + from -;
then there are probably many such hyperplanes; how to pick?)
 2. The training data are **almost linearly separable**
(but a few noisy or unusual training points get in the way;
we'll just allow some error on the training set)
 3. The training data are **linearly inseparable**
(the right decision boundary doesn't look like a hyperplane at all;
we'll have to do something smarter)

Linear separability

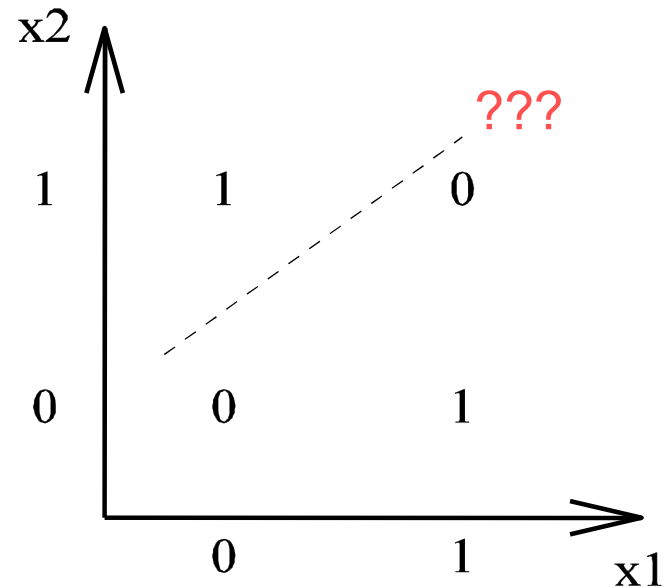


linearly separable

Linear separability



almost linearly separable



linearly inseparable

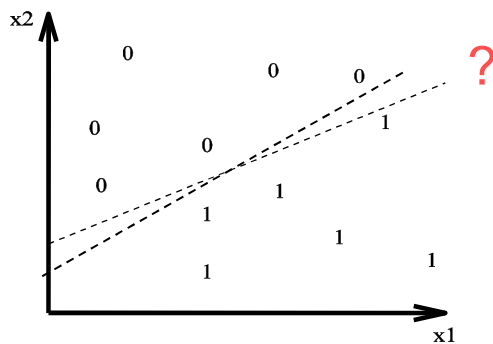
In fact, the simplest case:

$$y = x_1 \text{ xor } x_2$$

- Can learn e.g. concepts in the “at least m-of-n” family (what are w and b in this case?)
- But can't learn arbitrary boolean concepts like xor

Finding a separating hyperplane

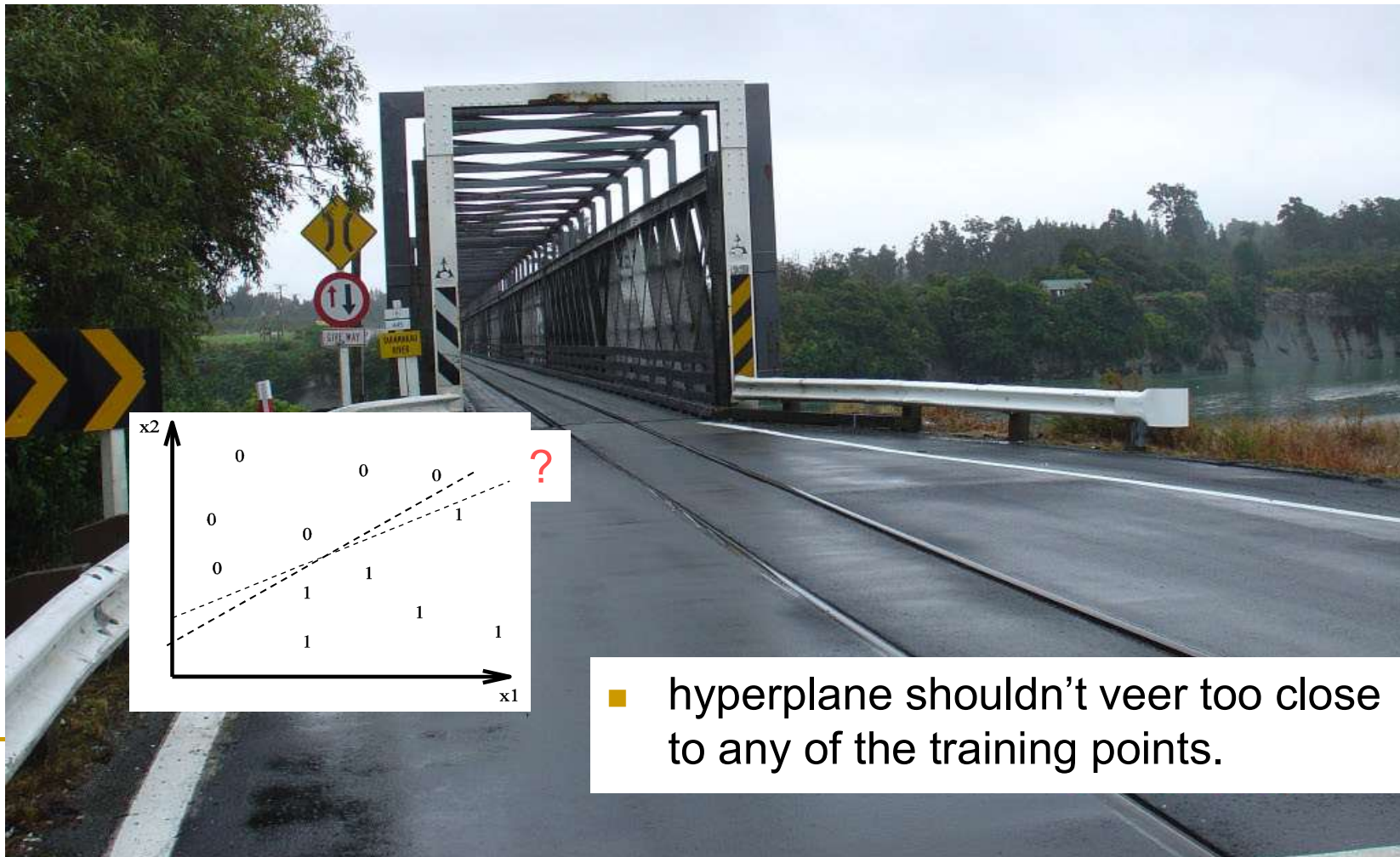
- If the data really are separable, can set this up as a linear constraint problem with real values:
 - Training data: $(x_1, +)$, $(x_2, +)$, $(x_3, -)$, ...
 - \rightarrow Constraints: $w \cdot x_1 \geq 0$, $w \cdot x_2 \geq 0$, $w \cdot x_3 \leq 0$, ...
 - \rightarrow Variables: the numeric components of vector w
 - But infinitely many solutions for solver to find ...



- ... luckily, the standard linear programming problem gives a declarative way to say which solution we want: minimize some cost subject to the constraints.
- **So what should the cost be?**

Finding a separating hyperplane

- Advice: stay in the middle of your lane;
drive in the center of the space available to you

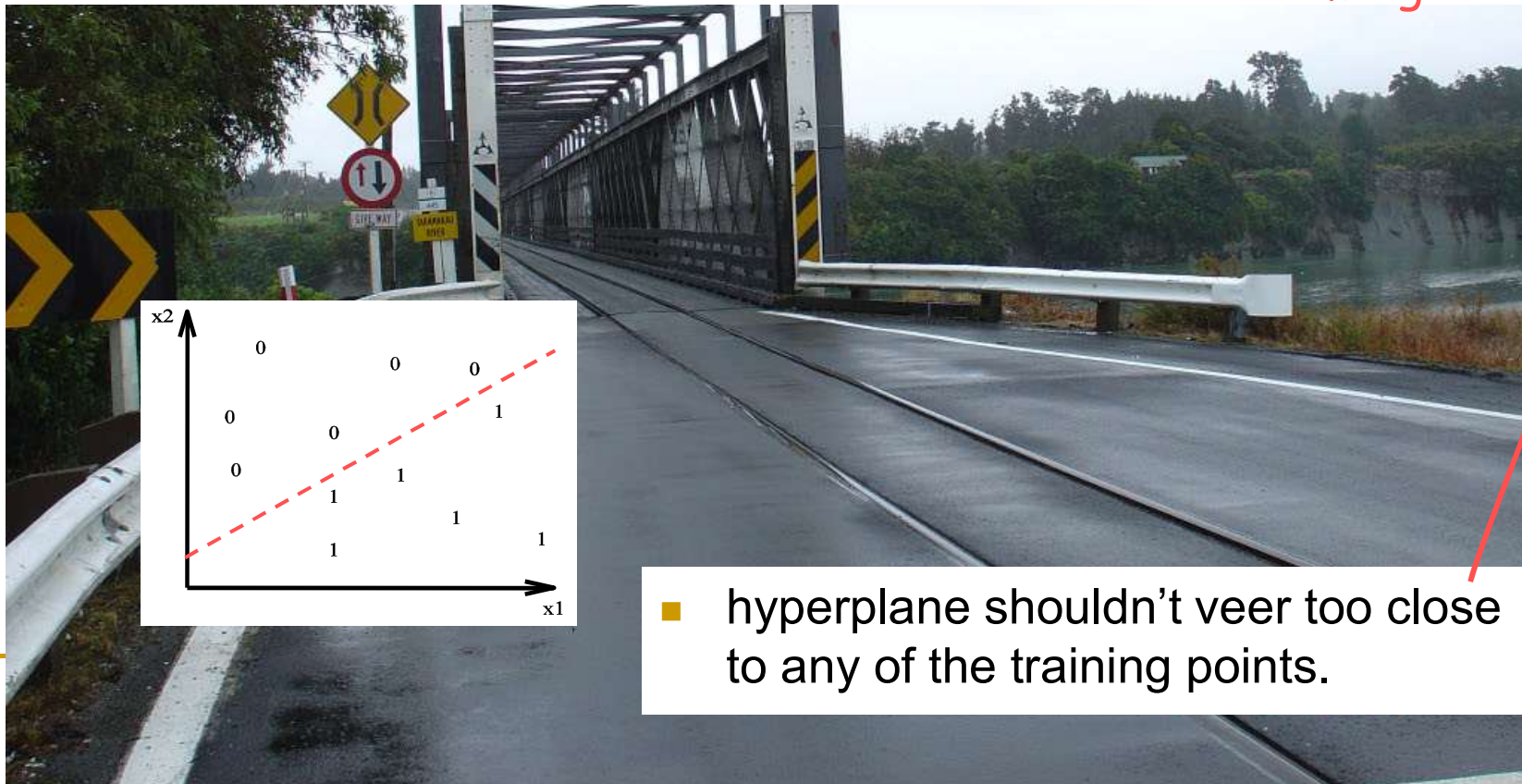


- hyperplane shouldn't veer too close to any of the training points.

Finding a separating hyperplane

- Advice: stay in the middle of your lane; drive in the center of the space available to you
- Define cost of a separating hyperplane = the distance to the nearest training point.

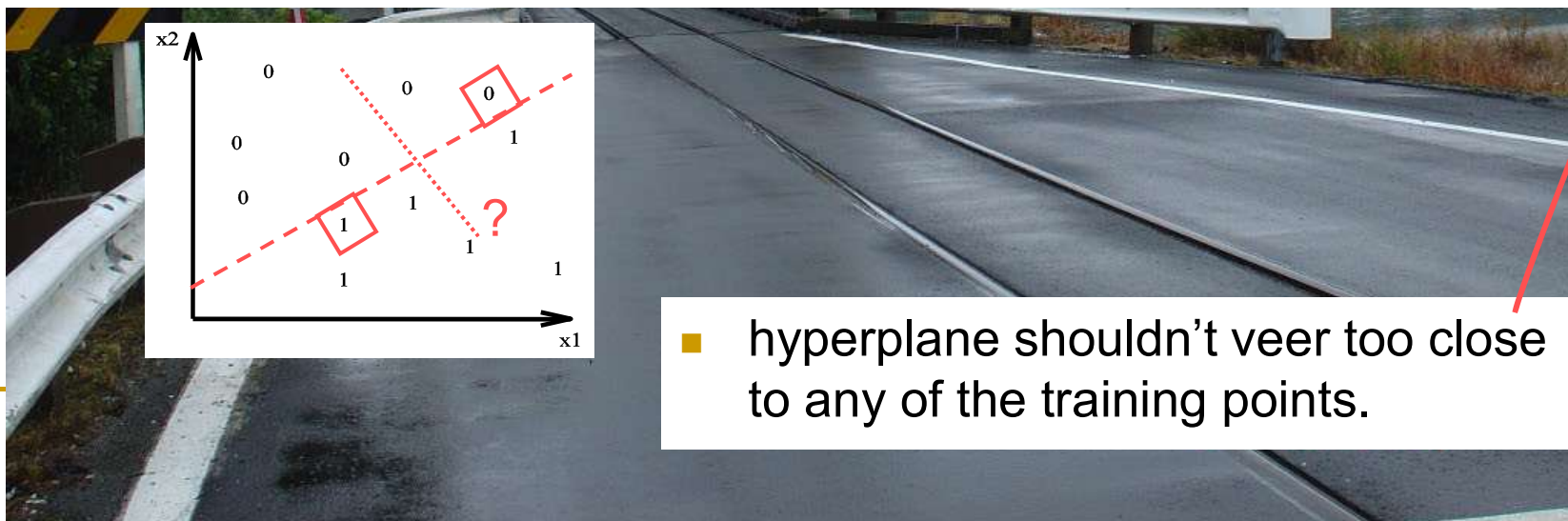
maximize this
"margin"



- hyperplane shouldn't veer too close to any of the training points.

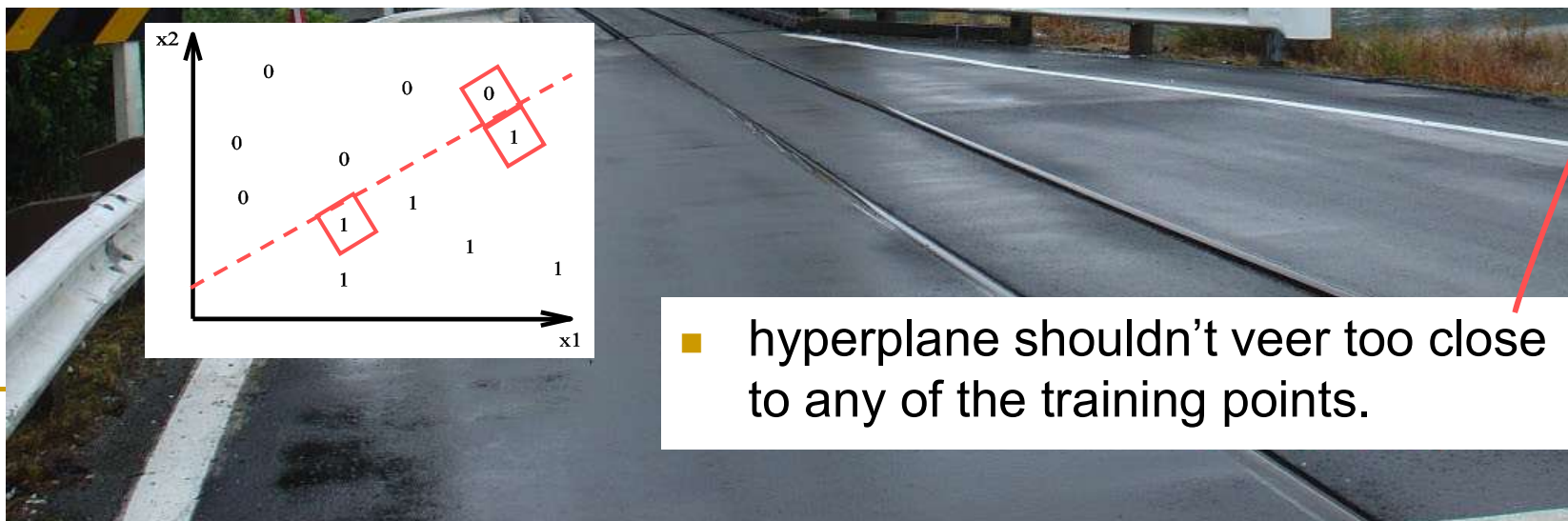
Finding a separating hyperplane

- Advice: stay in the middle of your lane; drive in the center of the space available to you
- Define cost of a separating hyperplane = **the distance to the nearest training point.** ← maximize this "margin"
- In the 2-dimensional case, usually at most 3 points can be nearest: hyperplane drives right between them.
 - The nearest training points to the hyperplane are called the "support vectors" (more in more dims), and are enough to define it.



Finding a separating hyperplane

- Advice: stay in the middle of your lane; drive in the center of the space available to you
- Define cost of a separating hyperplane = the distance to the nearest training point. ← maximize this "margin"
- In the 2-dimensional case, usually at most 3 points can be nearest: hyperplane drives right between them.
 - The nearest training points to the hyperplane are called the "support vectors" (more in more dims), and are enough to define it.

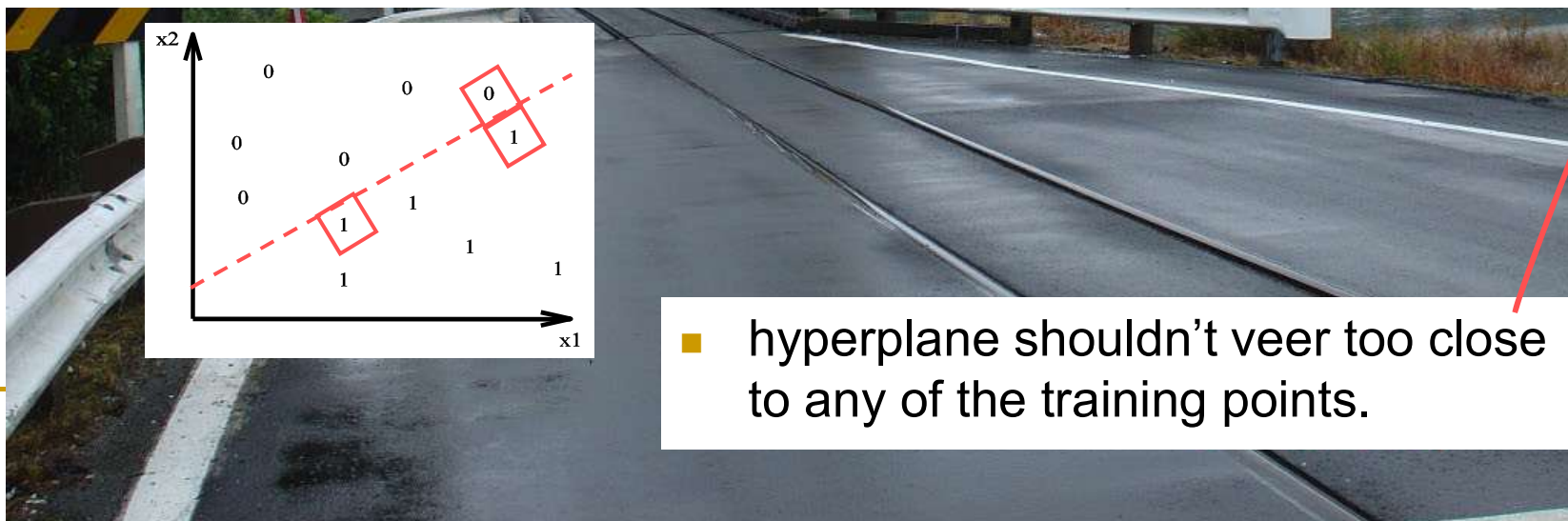


Finding a separating hyperplane

- <http://www.site.uottawa.ca/~gcaron/LinearApplet/LinearApplet.htm>
- Question: If you believe in nearest neighbors, why would the idea of maximizing the margin also be attractive?
 - Compare SVMs and nearest neighbor for test points that are near a training point.
 - Compare SVMs and nearest neighbor for test points that are far from a training point.

Finding a separating hyperplane

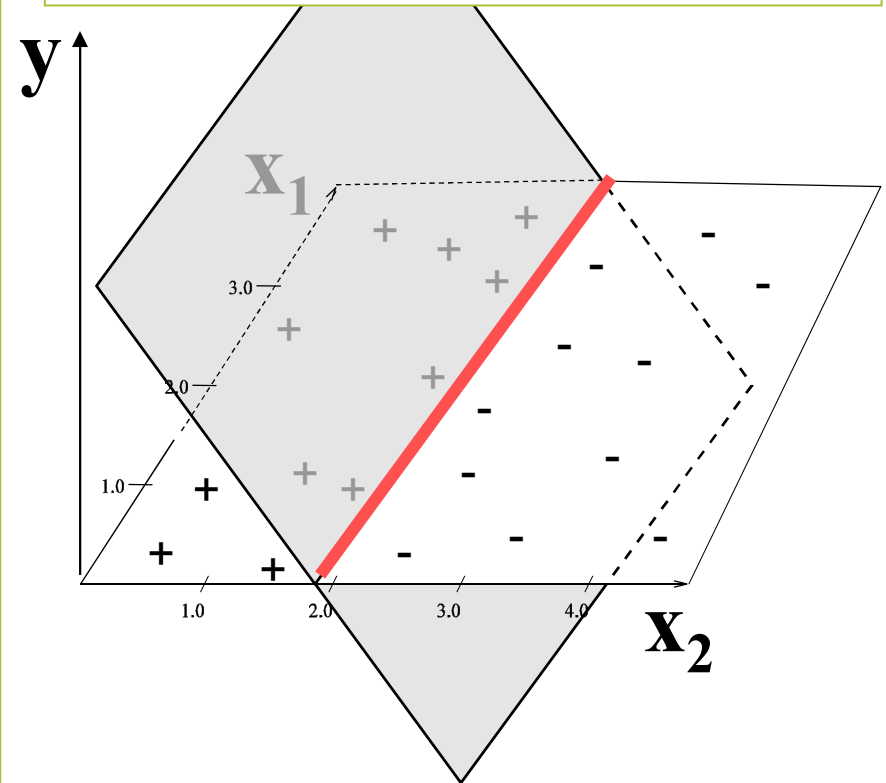
- Advice: stay in the middle of your lane;
drive in the center of the space available to you
- Define cost of a separating hyperplane =
the distance to the nearest training point. ← maximize this "margin"
- How do we define this cost in our constraint program??
 - Cost $\$ = \min(\text{distance to point 1, distance to point 2, ...})$
Big nasty distance formulas.
 - Instead we'll use a trick that lets us use a specialized solver.



Finding a separating hyperplane: trick

- To get a positive example x on the correct side of the red line, pick w so that $w \cdot x > 0$.
- To give it an extra margin, try to get $w \cdot x$ to be **as big as possible!**
- Good: $w \cdot x$ is bigger for points farther from red boundary (it is the height of the gray surface).
- **Oops! It's easier to double $w \cdot x$ simply by doubling w .**
 - That only changes the slope of the gray plane.
 - It doesn't change the red line where $w \cdot x = 0$. So same margin.

or for negative examples, $w \cdot x < 0$
and as negative as possible



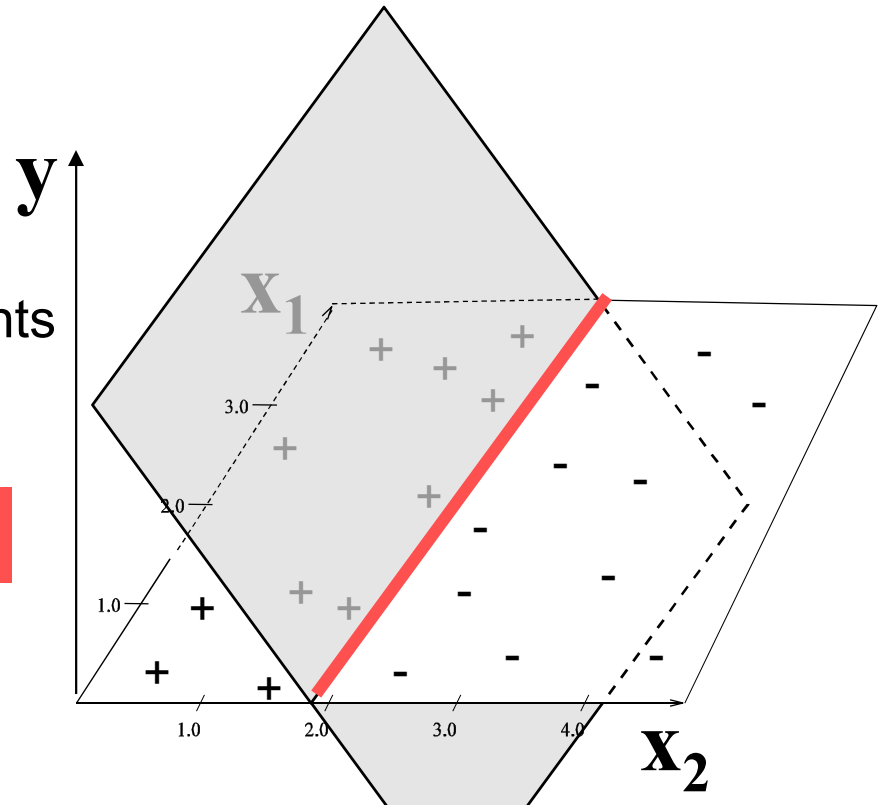
Finding a separating hyperplane: trick

To keep x far from the red line, must make the height $w \cdot x$ big **while keeping the slope $\|w\|$ small!**

- One option:
keep $\text{slope} \leq 1$, maximize all(?) heights
- Option with clearer meaning:
keep all heights ≥ 1 , minimize slope

minimize **slope**, subject to
 $w \cdot x \geq 1$ for each $(x, +)$
 $w \cdot x \leq -1$ for each $(x, -)$

$$\|\vec{w}\| = \sqrt{w_1^2 + w_2^2 \dots}$$



- Equivalently, instead of minimizing $\|w\|$, minimize $\|w\|^2$ (gets rid of the $\sqrt{}$) under the same linear constraints.
- $\|w\|^2$ is quadratic function: can use a quadratic programming solver!

Support Vector Machines (SVMs)

- That's what you just saw!
- SVM = linear classifier that's chosen to maximize the “margin.”
 - Margin = distance from the hyperplane decision boundary to its closest training points (the “support vectors”).
- To choose the SVM, use a quadratic programming solver.
 - Finds the **best** solution in polynomial time.
 - Mixes **soft** and **hard** constraints:
 - Minimize $\|w\|^2$ while satisfying some $w \cdot x \geq 1$ and $w \cdot x \leq -1$ constraints.
 - That is, find a maximum-margin separating hyperplane.
- But what if the data aren't linearly separable?
 - Constraint program will have no solution ...

SVMs that tolerate noise

(if the training data aren't quite linearly separable)

- Let's stay declarative: edit the constraint program to **allow** but **penalize** misclassification.

- Instead of requiring $w \cdot x \geq 1$, only require $w \cdot x + \text{fudge_factor} \geq 1$
 - One fudge factor ("slack variable") for each example

Easy to satisfy constraints now! Major fudging everywhere!
- Better keep the fudge factors small: just **add** them to the cost $\|w\|^2$

It's not free to "move individual points" to improve separability

New total cost function is a sum trying to balance two objectives:

Costs

Moving one point by one inch: **\$3**
(done by fudging the plane height $w \cdot x$ as if the point had been moved)

Benefits

Getting an extra inch of margin: **\$100**
(done by reducing $\|w\|^2$ by some amount while keeping $w \cdot x + \text{fudge} \geq 1$)

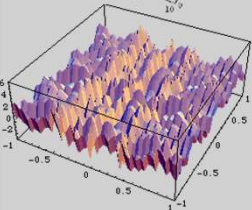
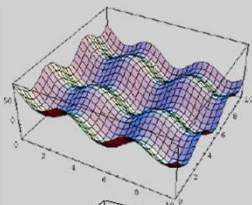
Development data to pick these relative numbers: **Priceless**

Simpler than SVMs: Just minimize cost

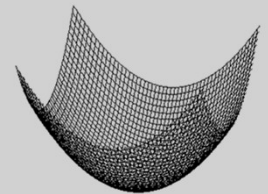
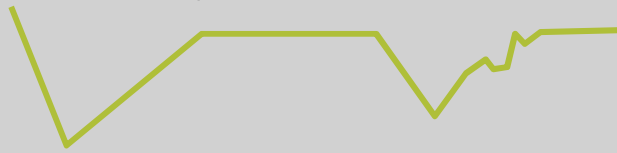
(what everyone did until SVMs were invented recently; still good)

- Don't use any hard constraints or QP solvers.
- Define the “best hyperplane” using only soft constraints on w .
 - In other words, just minimize one big cost function.
- **What should the cost function be?**

- For this, use your favorite function minimization algorithm.
 - gradient descent, conjugate gradient, variable metric, etc.
 - (Go take an optimization course: 550.{361,661,662}.)
 - (Or just download some software!)



nasty non-differentiable cost function with local minima



nice smooth and convex cost function: pick one of these

Simpler than SVMs: Just minimize cost

■ What should the cost function be?

- Training data: $(x_1, +)$, $(x_2, +)$, $(x_3, -)$, ...
- Try to ensure that $w \cdot x \approx 1$ for the positive examples,
 $w \cdot x \approx -1$ for the negative examples.
 - Impose some kind of cost for bad approximations “ \approx ”.
 - Every example contributes to the total cost.
(Whereas SVM cost only cares about closest examples.)

We're not saying “ $w \cdot x$ as big as possible” ...

Want $w \cdot x$ to be positive, not merely “big.”

*The difference between -1 and 1 is “special” (it crosses the 0 threshold)
... not like the difference between -3 and -1 or 7 and 9.*

*Anyway, one could make $w \cdot x$ bigger just by doubling w (as with SVMs).
So $w \cdot x \approx 2$ shouldn't be twice as good as $w \cdot x \approx 1$.*

(Some methods even say it's worse!)

Simpler than SVMs: Just minimize cost

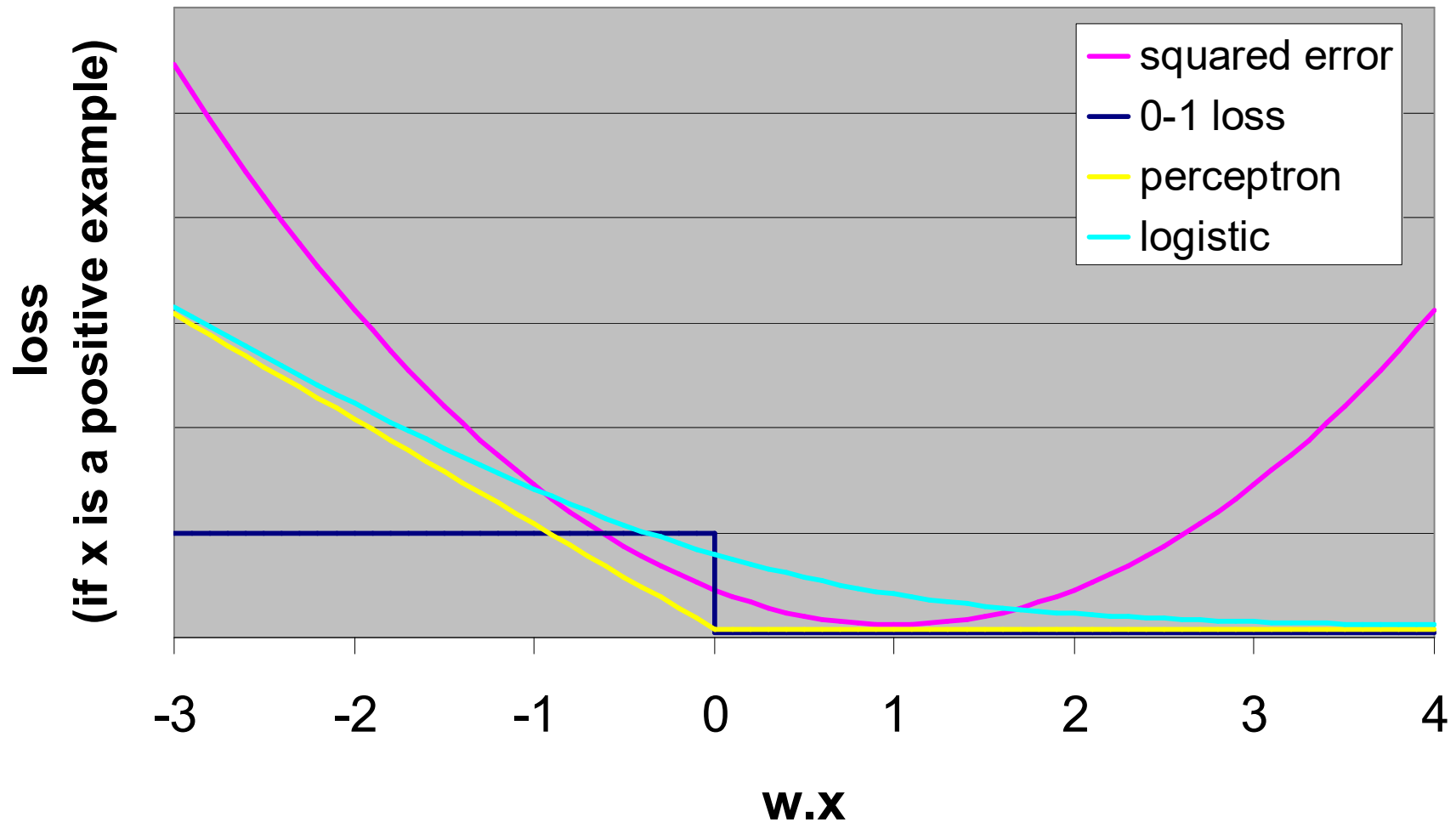
■ What should the cost function be?

- Training data: $(x_1, +)$, $(x_2, +)$, $(x_3, -)$, ...
- Try to ensure that $w \cdot x \approx 1$ for the positive examples,
 $w \cdot x \approx -1$ for the negative examples.
 - Impose some kind of cost for bad approximations “ \approx ”.
 - Every example contributes to the total cost.
(Whereas SVM cost only cares about closest examples.)

The cost that classifier w incurs on a particular example is called its “loss” on that example.

Just define it, then look for a classifier that minimizes the **total** loss over all examples.

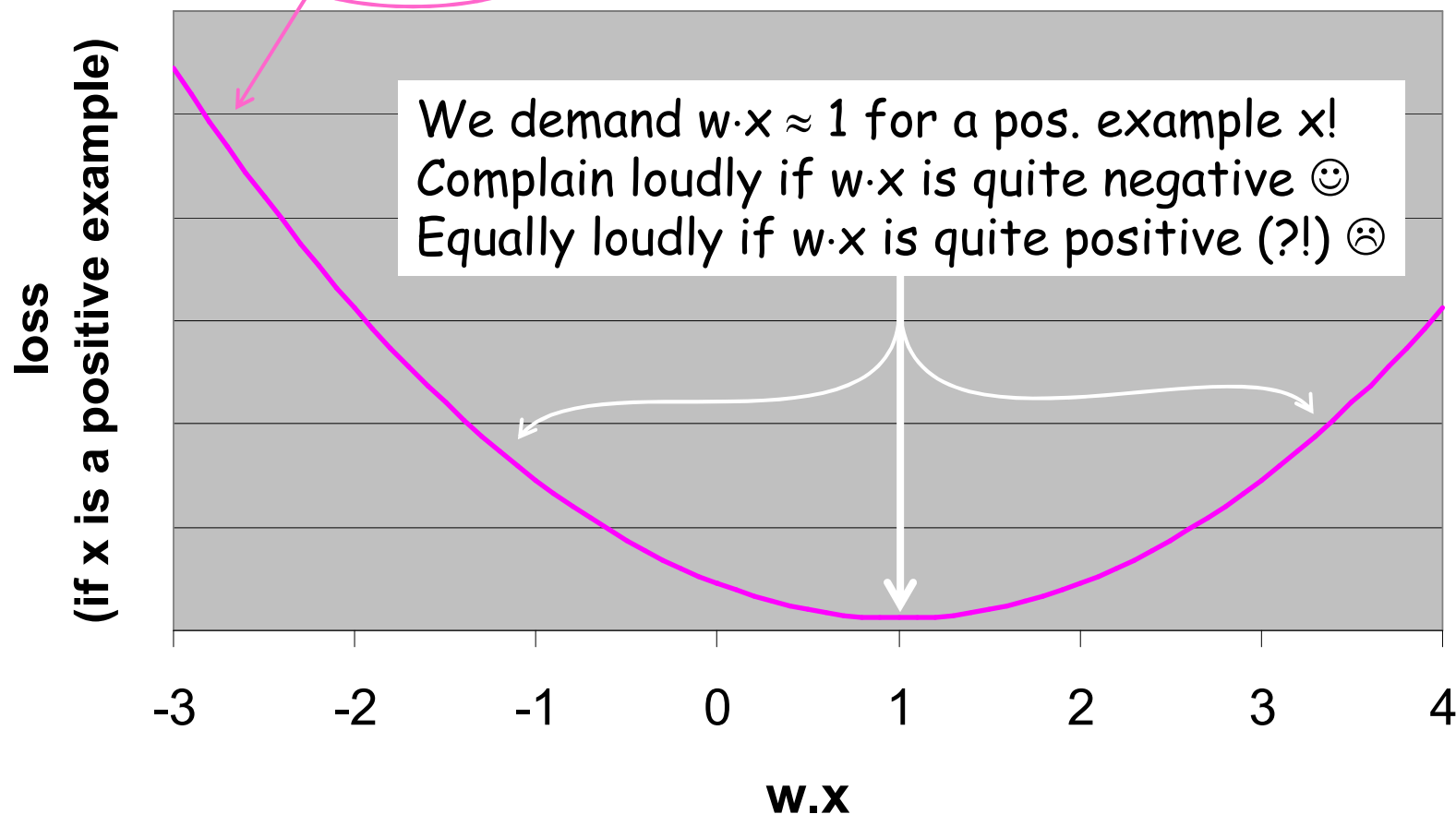
Some loss functions ...



“Least Mean Squared Error” (LMS)

Training data: $(x_1, +)$, $(x_2, +)$, $(x_3, -)$, ...

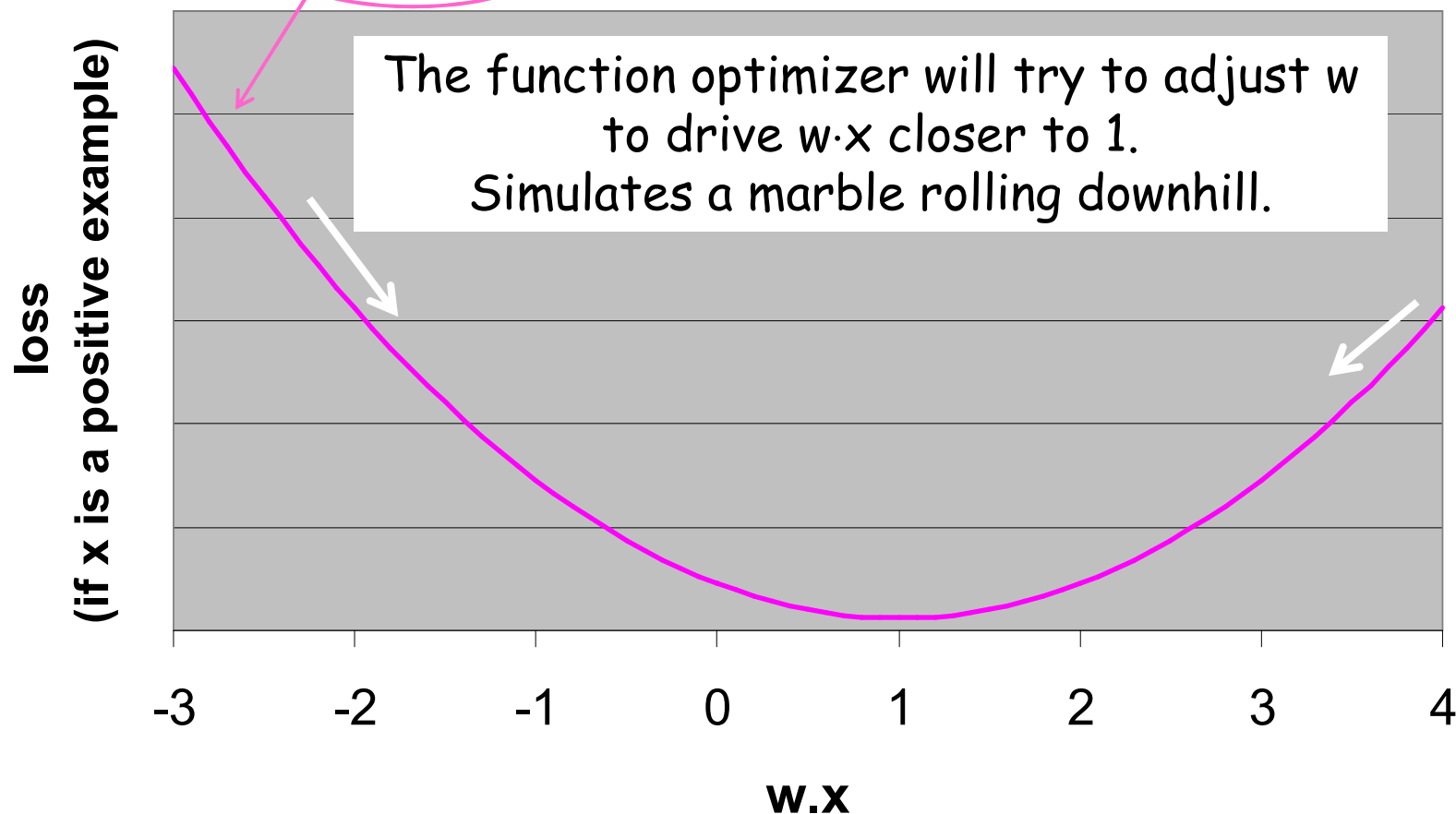
$$\rightarrow \text{Cost} = (w \cdot x_1 - 1)^2 + (w \cdot x_2 - 1)^2 + (w \cdot x_3 - (-1))^2 + \dots$$



“Least Mean Squared Error” (LMS)

Training data: $(x_1, +)$, $(x_2, +)$, $(x_3, -)$, ...

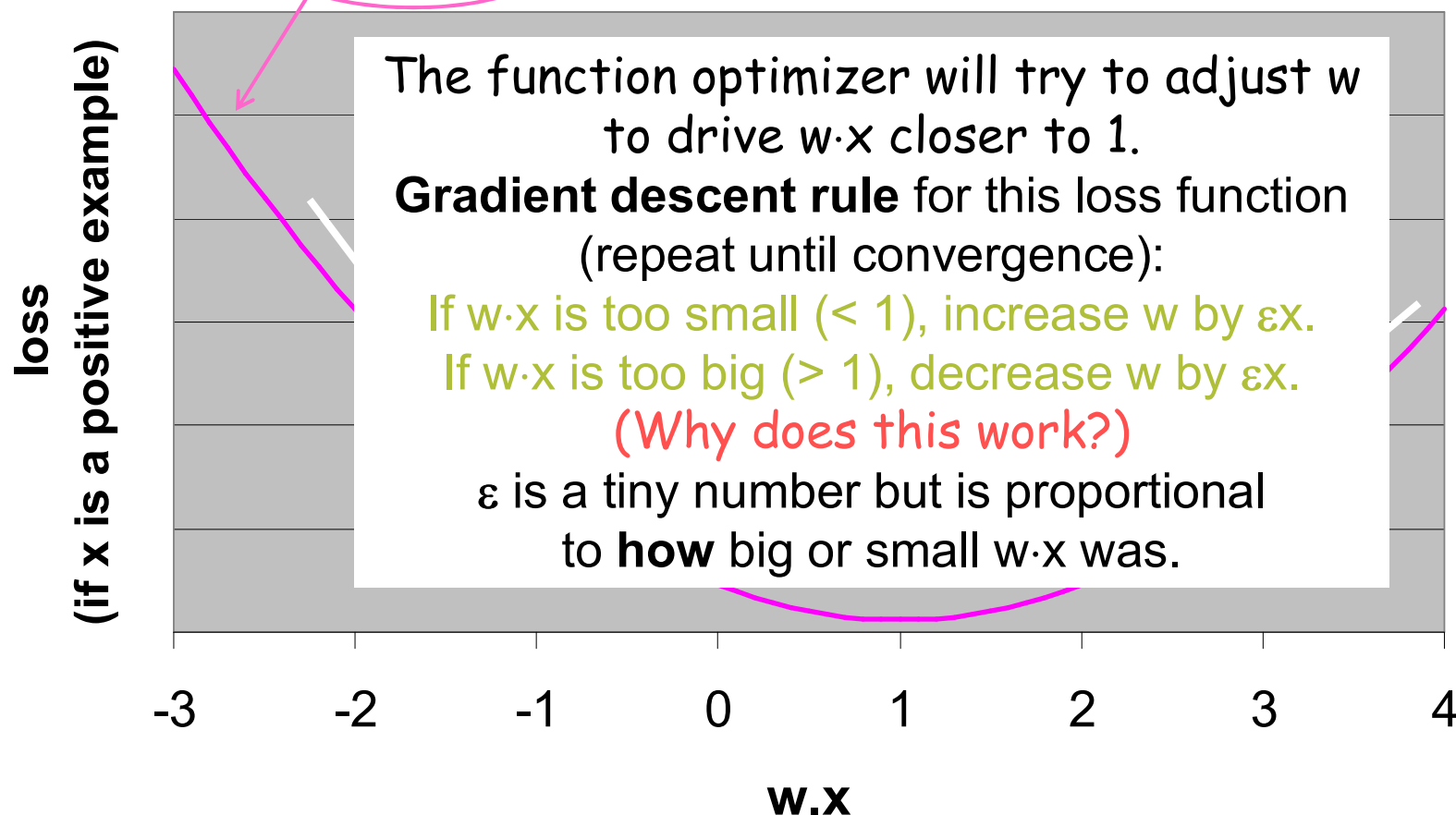
$$\rightarrow \text{Cost} = (w \cdot x_1 - 1)^2 + (w \cdot x_2 - 1)^2 + (w \cdot x_3 - (-1))^2 + \dots$$



“Least Mean Squared Error” (LMS)

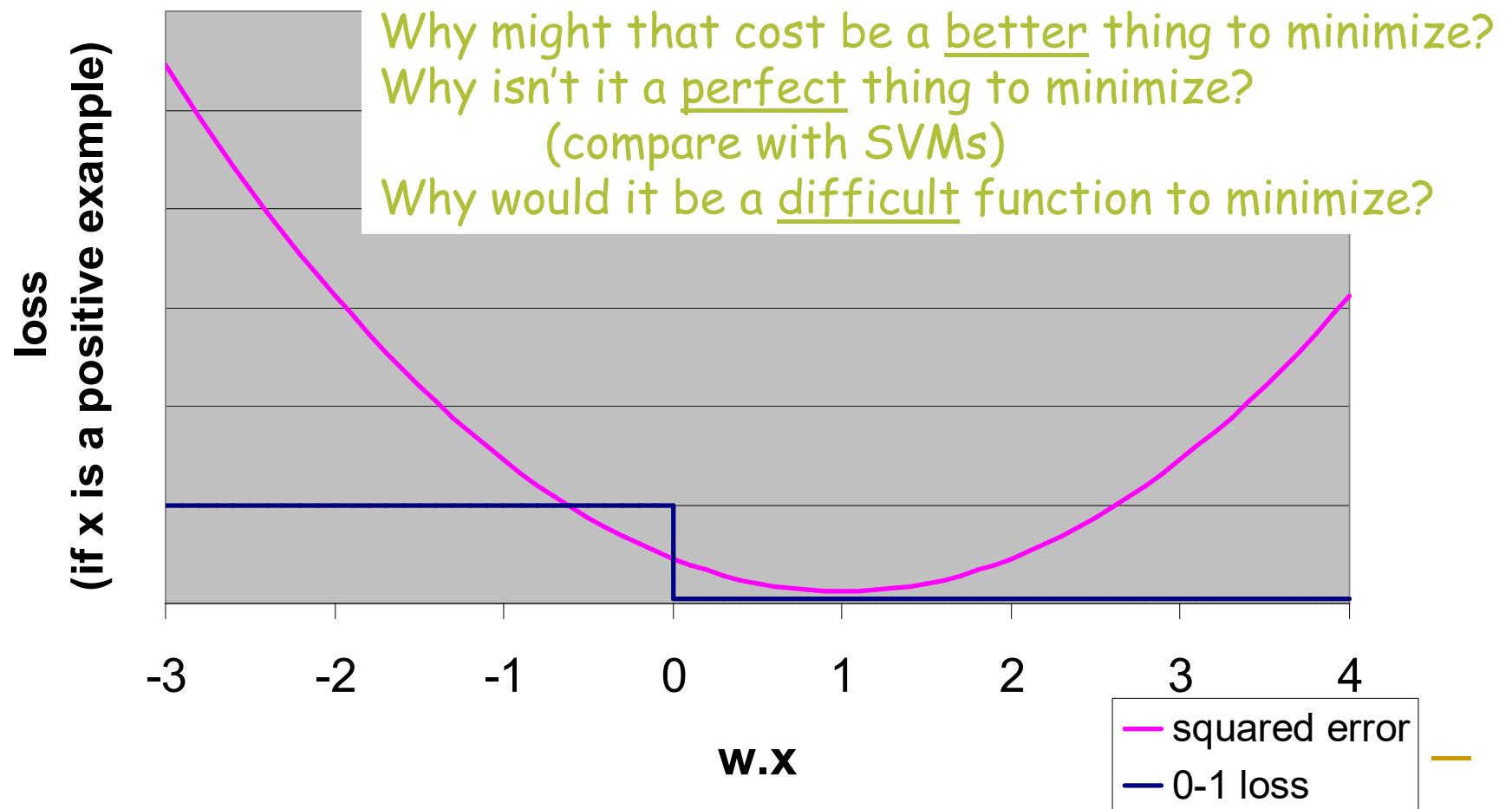
Training data: $(x_1, +)$, $(x_2, +)$, $(x_3, -)$, ...

$$\rightarrow \text{Cost} = (w \cdot x_1 - 1)^2 + (w \cdot x_2 - 1)^2 + (w \cdot x_3 - (-1))^2 + \dots$$



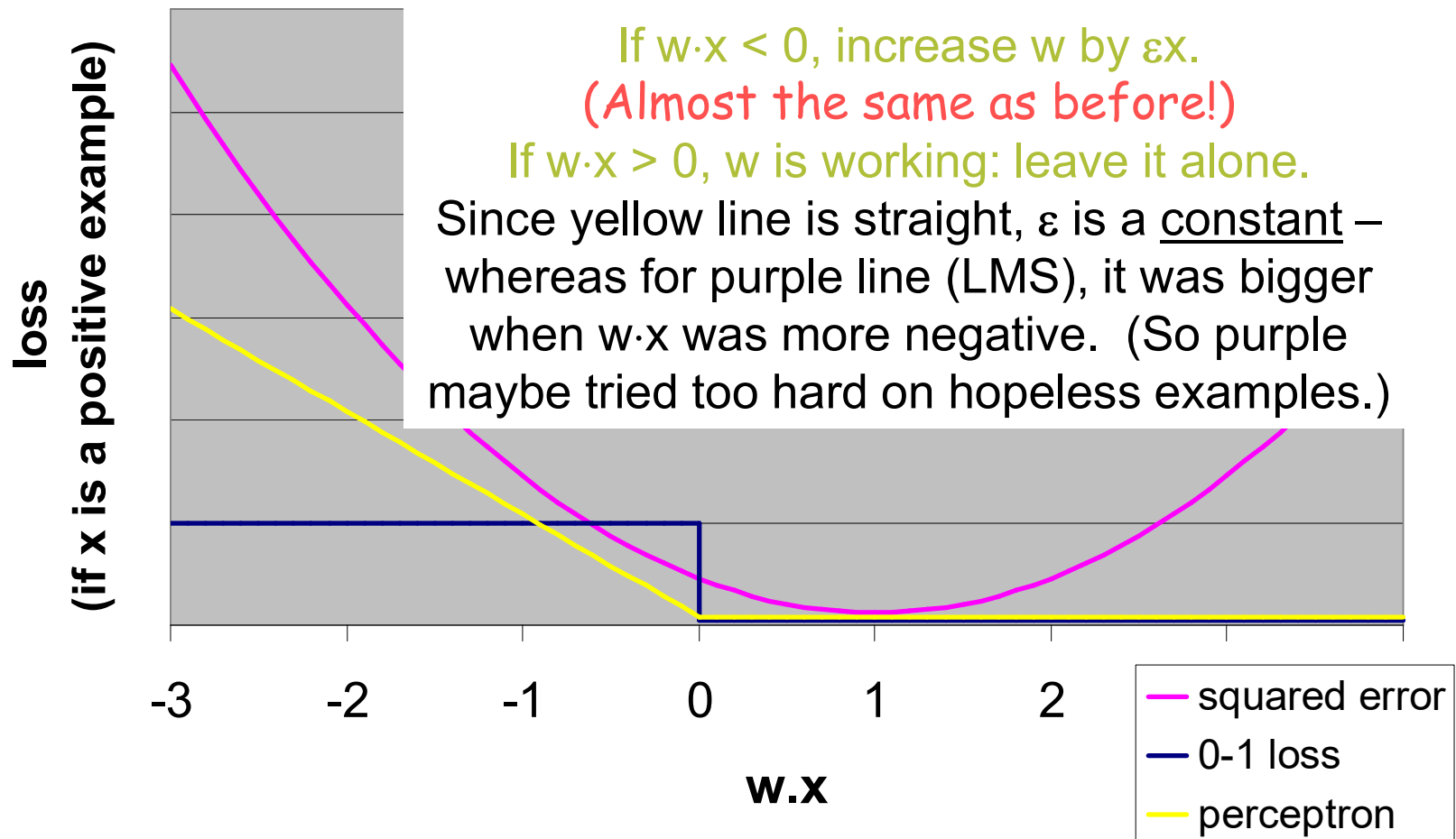
LMS versus 0-1 loss

- To see why LMS loss is weird, compare it to 0-1 loss (blue line).
- What is the total 0-1 loss over all training examples?



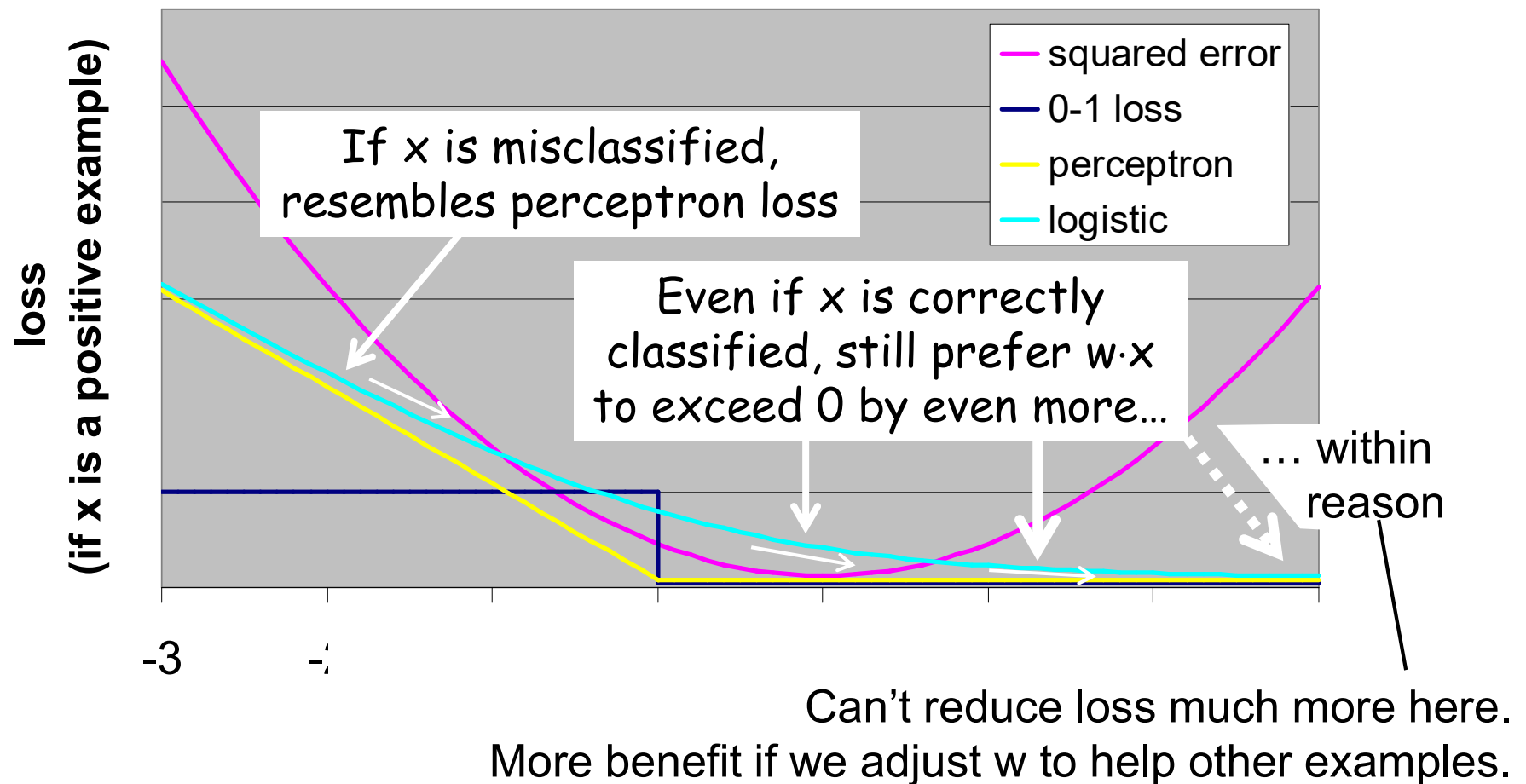
Perceptron algorithm (old!)

- This yellow loss function is easier to minimize.
- Its gradient descent rule is the “perceptron algorithm”:



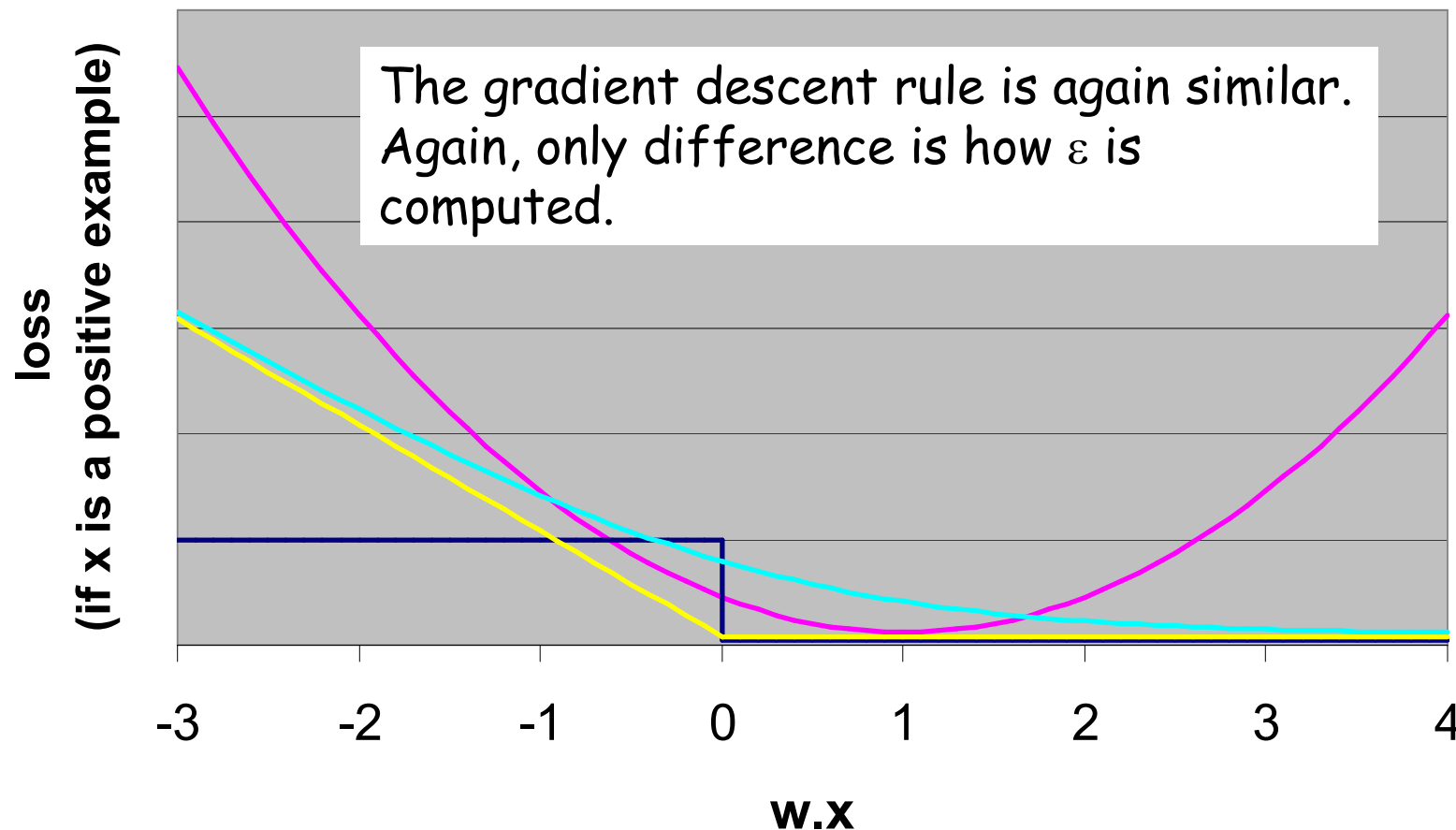
Logistic regression

- The light blue loss function gets back a “margin”-like idea:
 $\log(\exp(w \cdot x) / (1 + \exp(w \cdot x)))$



Logistic regression

- The light blue loss function gets back a “margin”-like idea:
 $\log(\exp(w \cdot x) / (1 + \exp(w \cdot x)))$

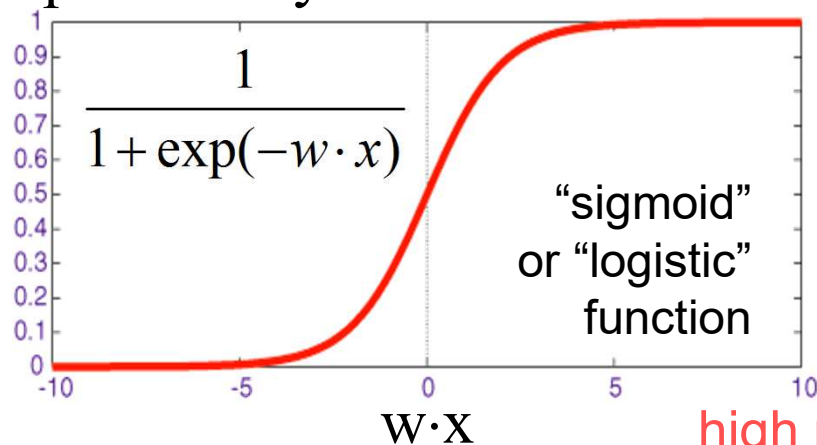


A justification of logistic regression

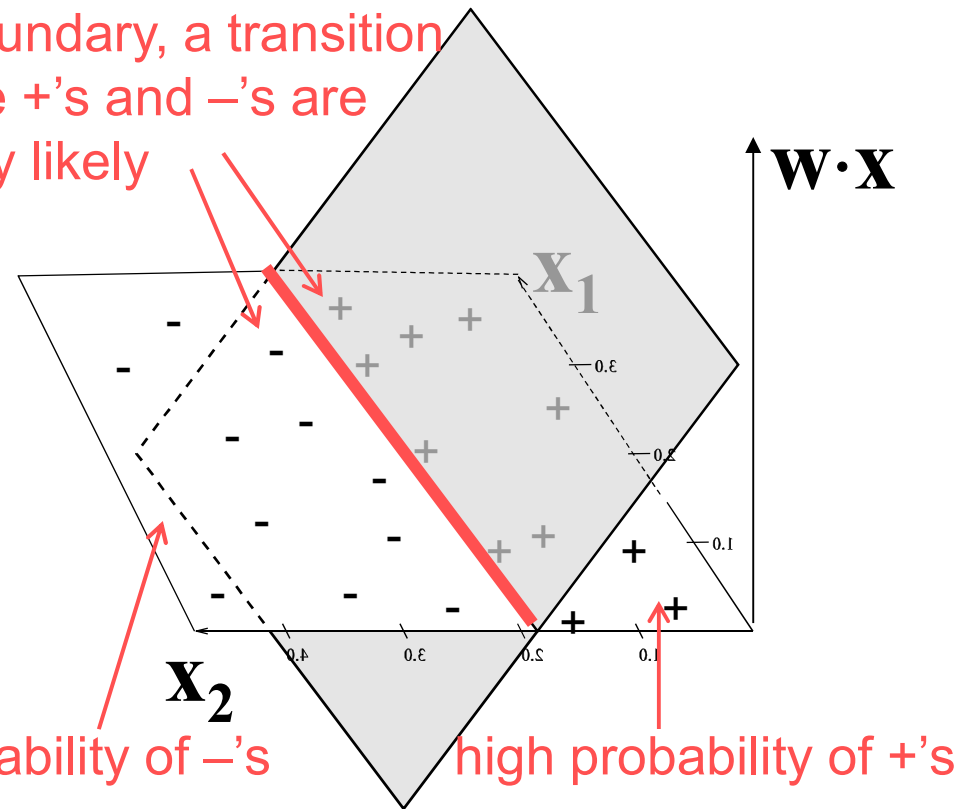
- Logistic regression is derived from the following assumption.
- Suppose a true linear boundary exists, but is not a separator. It caused the + and – labels to be assigned probabilistically ...

(w determines the boundary line and the gradualness of the transition)

probability that x is labeled +

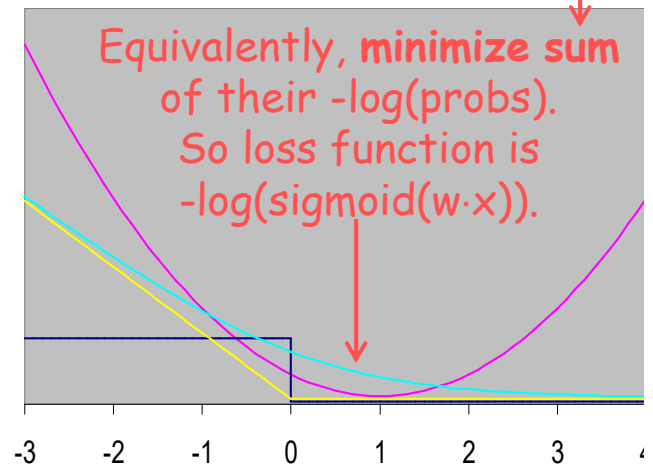


very near boundary, a transition region where +’s and –’s are about equally likely



A justification of logistic regression

- Logistic regression is derived from the following assumption.
- Suppose a true linear boundary exists, but is not a separator. It caused the + and – labels to be assigned probabilistically ...
- We want to find boundary so the + and – labels we actually saw would have been probable. Pick w to max product of their probs.
- In other words, if x was labeled as + in training data, we want its $\text{prob}(+)$ to be pretty high:
 - Prob definitely should be far from 0.
 - Going from 1% to 10% gives 10x probability.
 - Prob preferably should be close to 1.
 - Going from 90% to 99% multiplies probability by only 1.1.
 - That's “why” our blue curve is asymmetric!
 - For $\text{sigmoid}(w \cdot x)$ to be definitely far from 0, preferably close to 1, $w \cdot x$ should be definitely far from $-\infty$ and preferably close to $+\infty$.

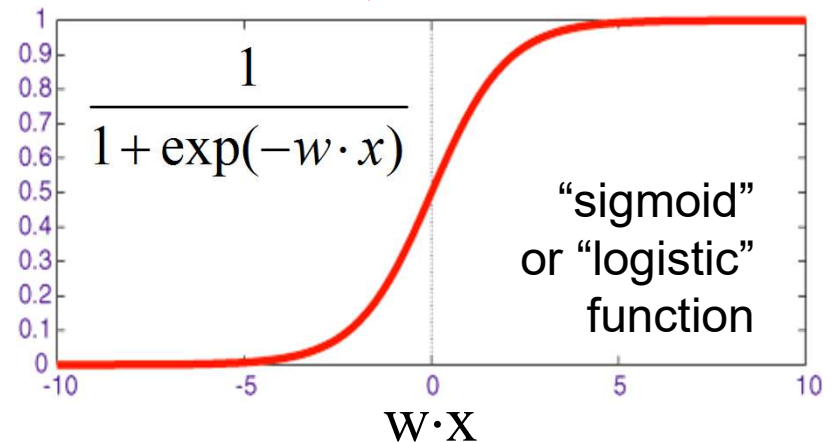


One more loss function

- Neural networks tend to just use this directly as the loss function. (Upside down: it's 1 minus this.)
- So they try to choose w so that $\text{sigmoid}(w \cdot x) \approx 1$ for positive examples ($\text{sigmoid}(w \cdot x) \approx 0$ for negative examples)
- Not the same as asking for $w \cdot x \approx 1$ directly!

It's asking $w \cdot x$ to be **large** (but again, there are diminishing returns – not much added benefit to making it very large).

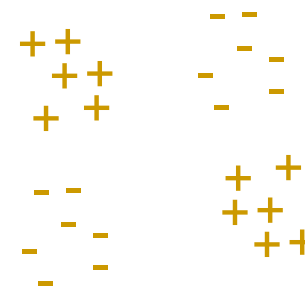
Why? It resembles 0-1 loss.
☺ Differentiable and nice instead of piecewise constant.
☹ Total cost function still has local minima.



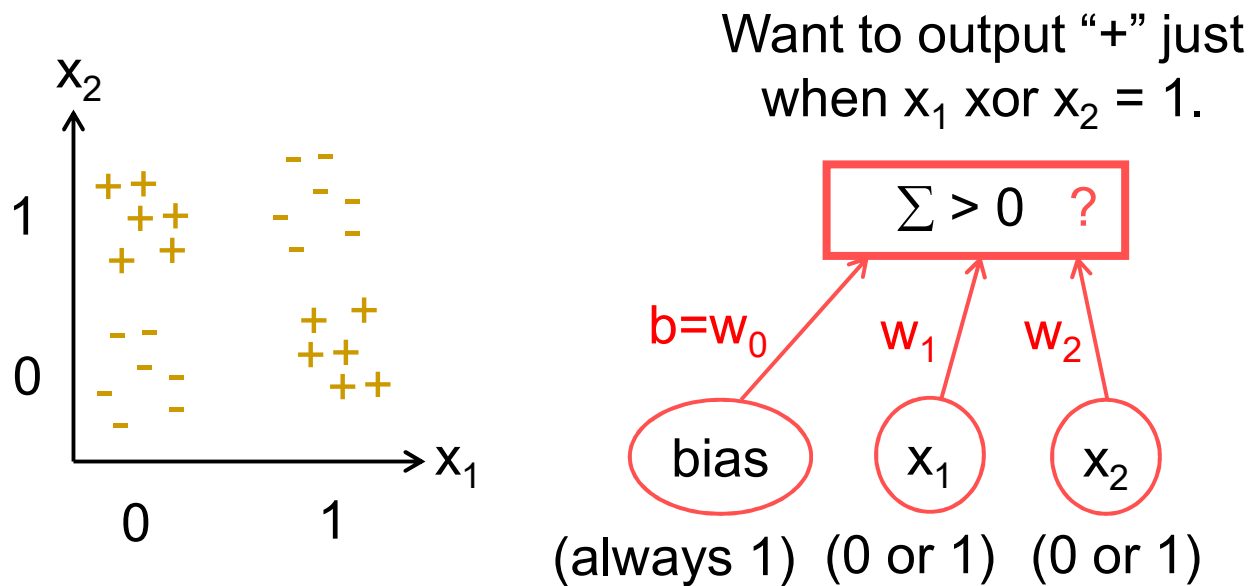
Using linear classifiers on linearly inseparable data

Isn't logistic regression enough?

- “Soft” (probabilistic) decision boundary
 - Hyperplane boundary $w \cdot x = 0$ is not so special anymore
 - It just marks where $\text{prob}(+) = 0.5$
- So logistic regression can tolerate some overlapping of + and – areas, especially near the boundary.
- **But it still assumes a single, straight boundary!**
- How will we deal with seriously inseparable data like xor?



The xor problem



A schematic way of showing how $w \cdot x + b$ is computed. The red box outputs + or – according to whether $w \cdot x + b > 0$.

Can't be done with any $w = (w_0, w_1, w_2)$!

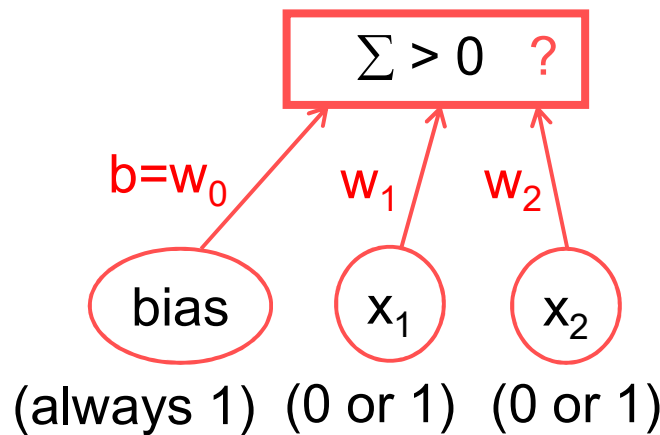
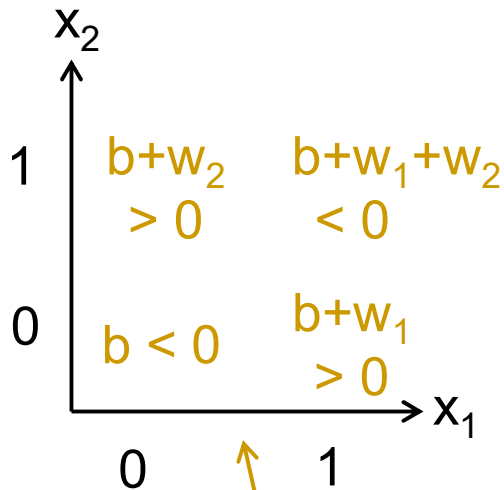
Why not?

If w is such that

turning on **either** $x_1=1$ or $x_2=1$ will push $w \cdot x + b$ above 0,
then turning on **both** $x_1=1$ and $x_2=1$ will push $w \cdot x + b$ even higher.

The xor problem

Want to output “+” just
when $x_1 \text{ xor } x_2 = 1$.



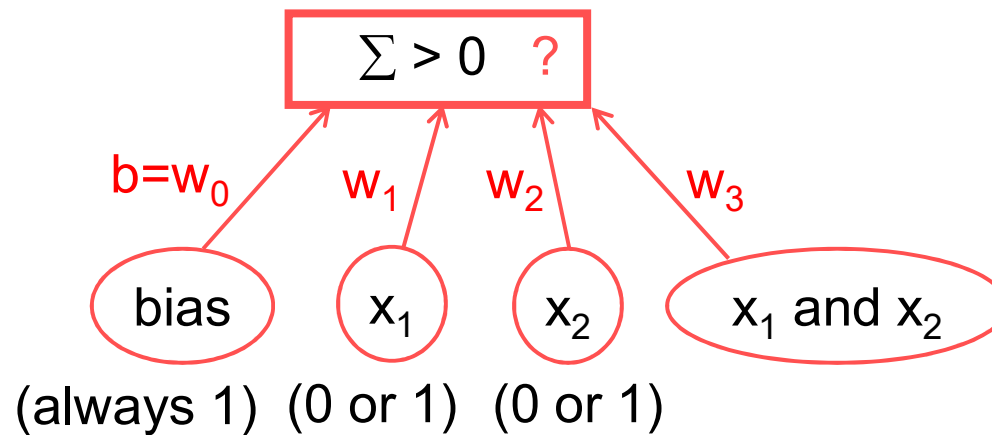
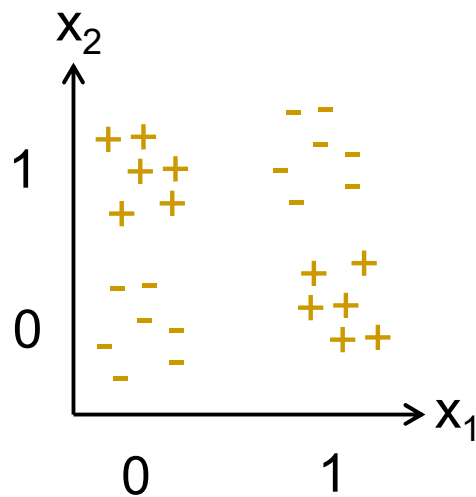
A schematic way of
showing how $w \cdot x + b$
is computed. The
red box outputs
+ or – according to
whether $w \cdot x + b > 0$.

Can't be done with any $w = (w_0, w_1, w_2)$!
Why not?

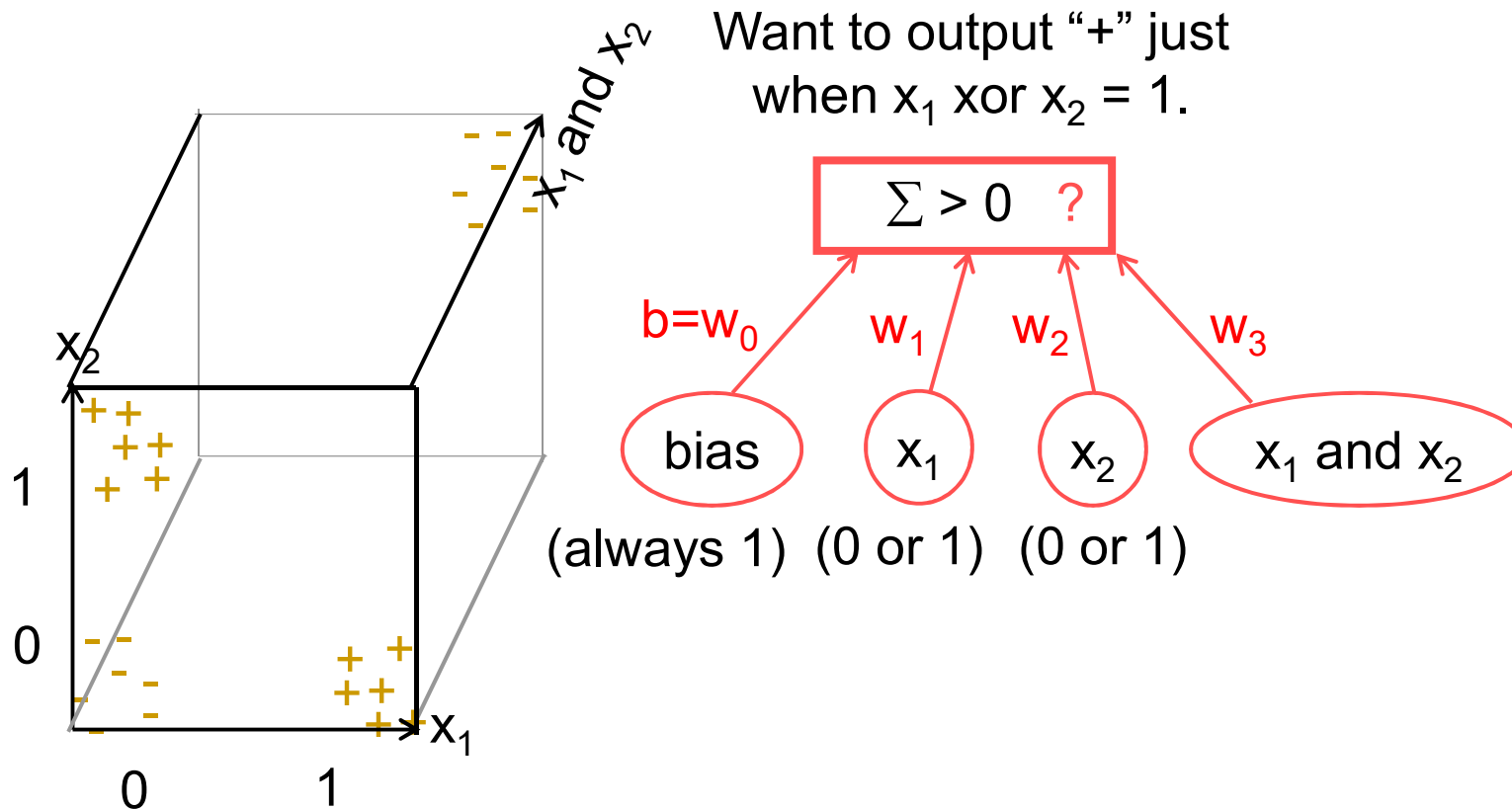
Formal proof: these equations would all have to be true,
but they are inconsistent.

The xor solution: Add features

Want to output “+” just
when $x_1 \text{ xor } x_2 = 1$.

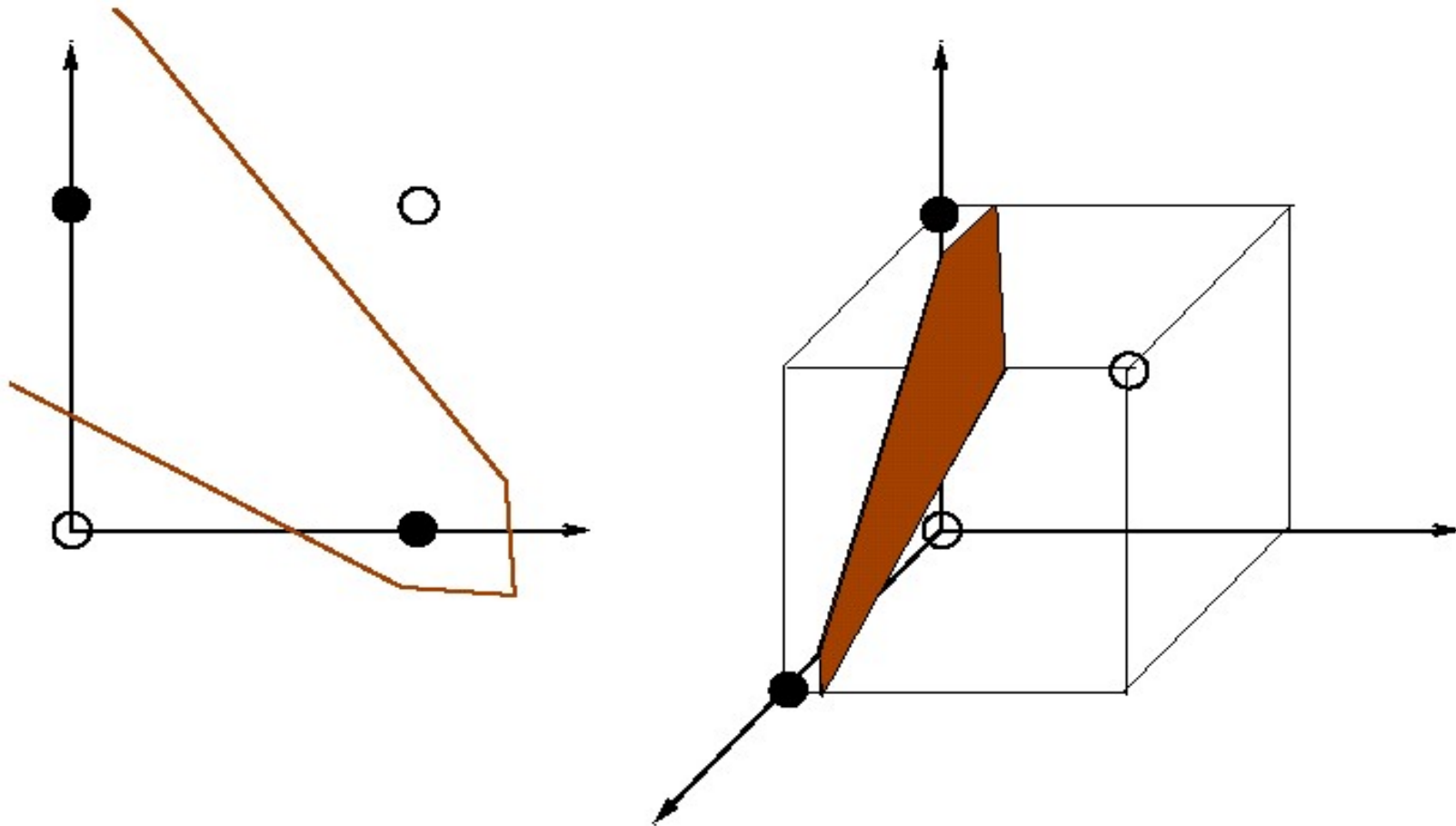


The xor solution: Add features

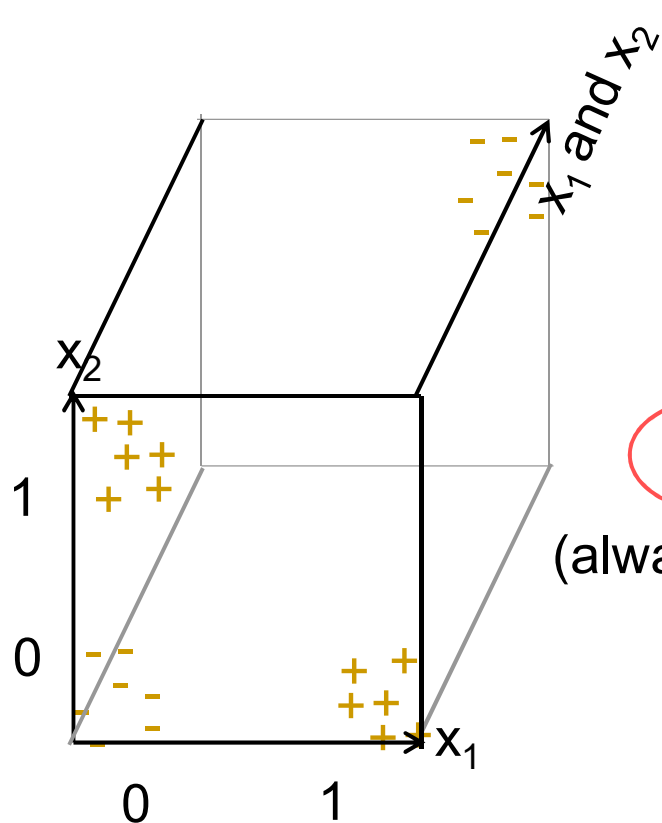


In this new 3D space, it is possible to draw a plane that separates + from -.

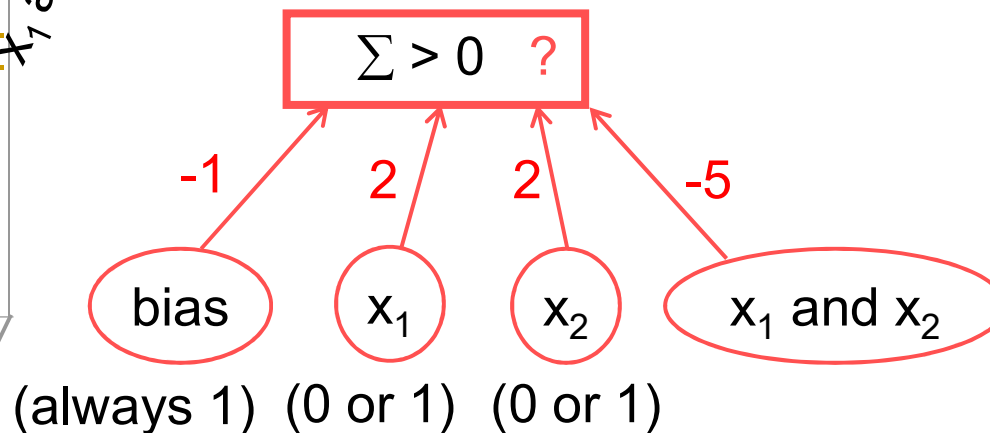
The xor solution: Add features



The xor solution: Add features



Want to output “+” just
when $x_1 \text{ xor } x_2 = 1$.



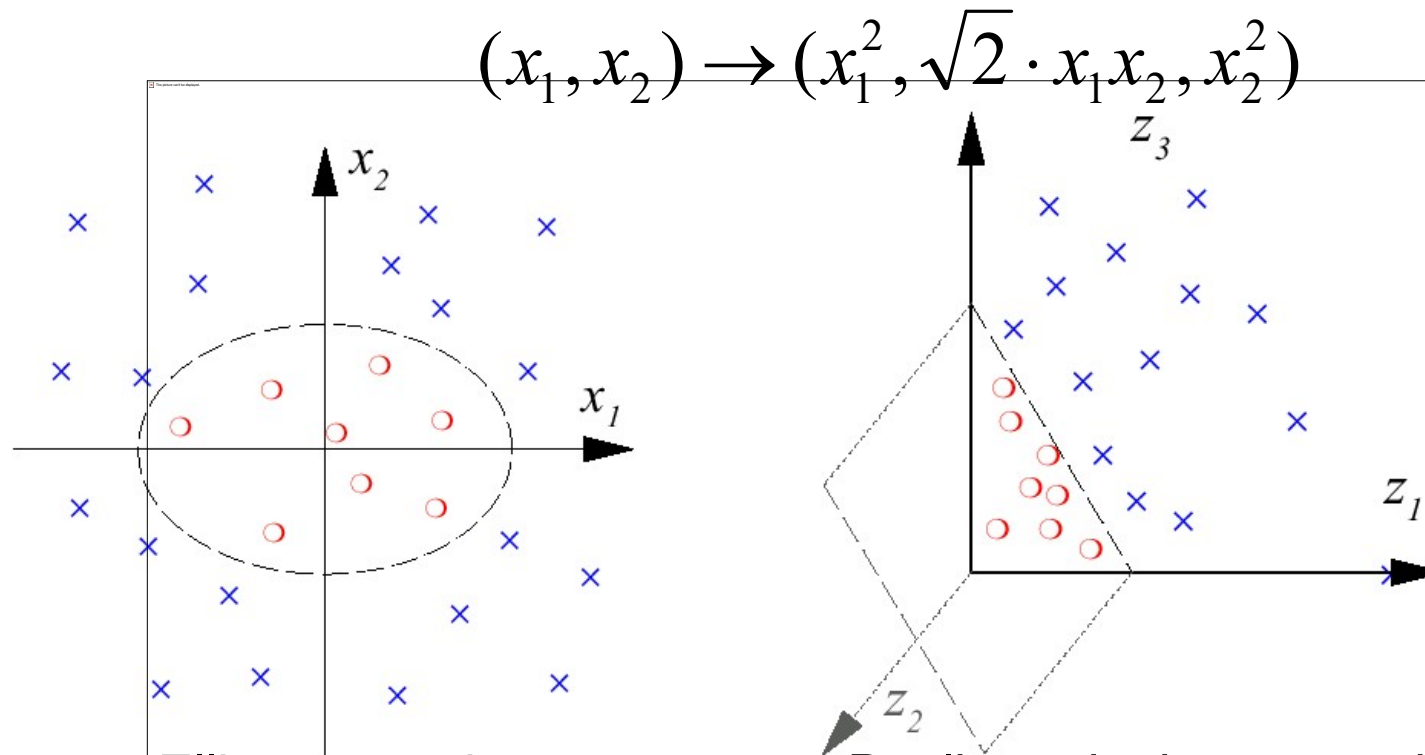
x_1 and x_2 drive the output positive.
But if they **both** fire,
then so does the new feature,
more than canceling out their combined effect.

“Blessing of dimensionality” 😊

- In an n -dimensional space, almost any set of up to $n+1$ labeled points is linearly separable!
- **General approach:** Encode your data in such a way that they become linearly separable.
- Option 1: Choose additional features by hand.
- Option 2: Automatically learn a small number of features that will suffice.
(neural networks)
- Option 3: Throw in a huge set of features like x_1 and x_2 , slicing and dicing the original features in many different but standard ways.
(kernel methods, usually with SVMs)

In fact, possible to use an infinite set of features (!)

Another example: The ellipse problem



Ellipse equation:

$$x_1^2 + 2x_2^2 < 3$$

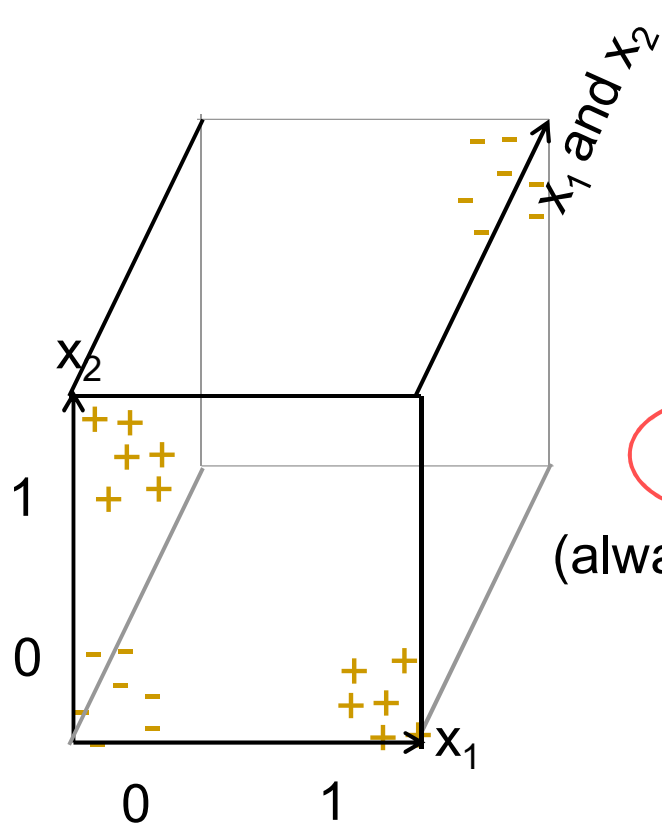
Not linear in the original variables.

But linear in the squared variables!
Boundaries defined by linear combinations of x^2 , y^2 , xy , x , and y are ellipses, parabolas, and hyperbolas in the original space.

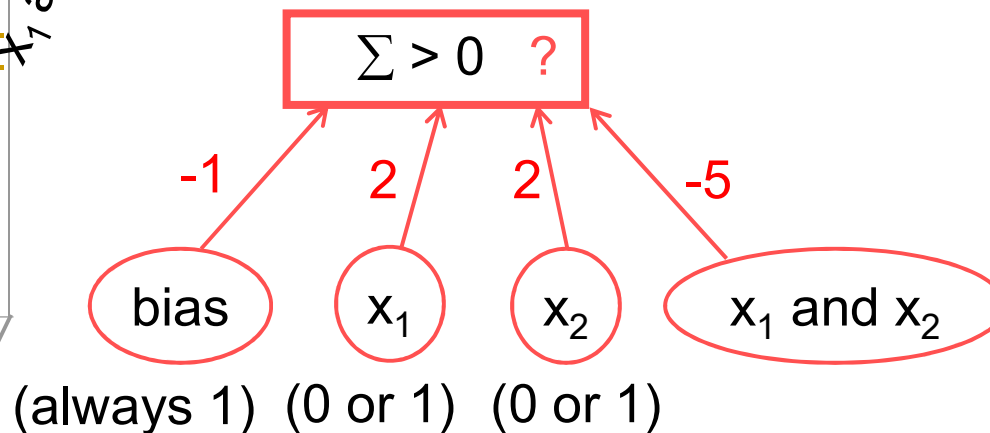
Adding new features

- Instead of a classifier w such that
 - $w \cdot x > 0$ for positive examples
 - $w \cdot x < 0$ for negative examples,
- pick some function Φ that turns x into a longer example vector, and learn a longer weight vector w such that
 - $w \cdot \Phi(x) > 0$ for positive examples
 - $w \cdot \Phi(x) < 0$ for negative examples
- So where does Φ come from?
 - Are there good standard Φ functions to try?
 - Or could we learn Φ too?

What kind of features to consider?

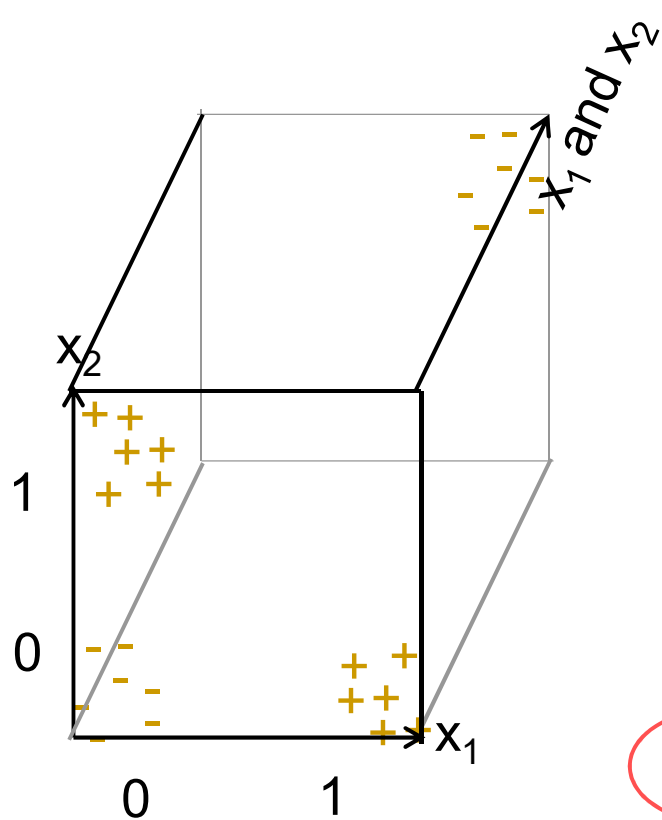


Want to output “+” just
when $x_1 \text{ xor } x_2 = 1$.

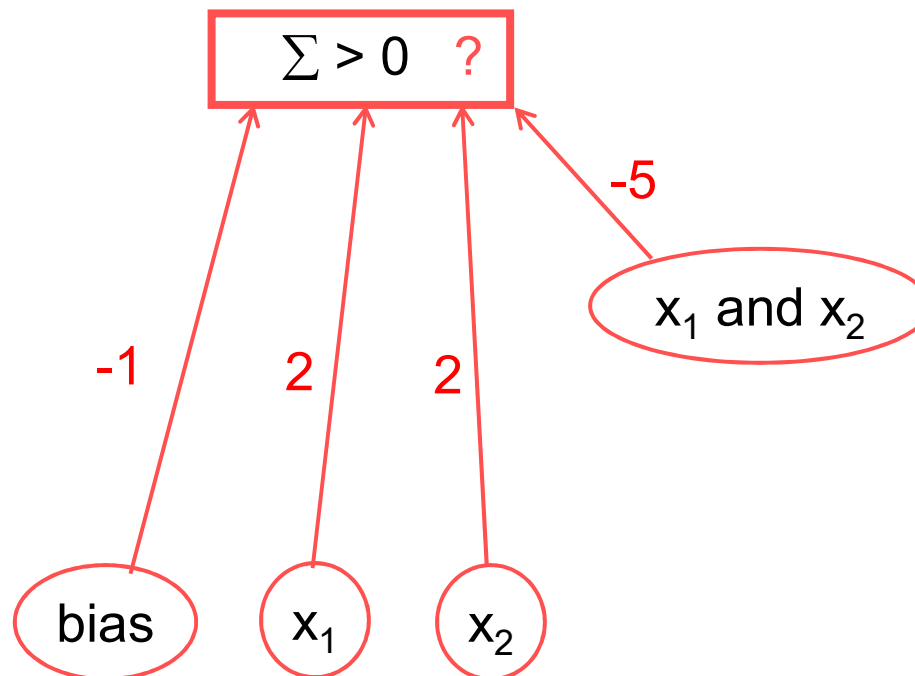


x_1 and x_2 drive the output positive.
But if they **both** fire,
then so does the new feature,
more than canceling out their combined effect.

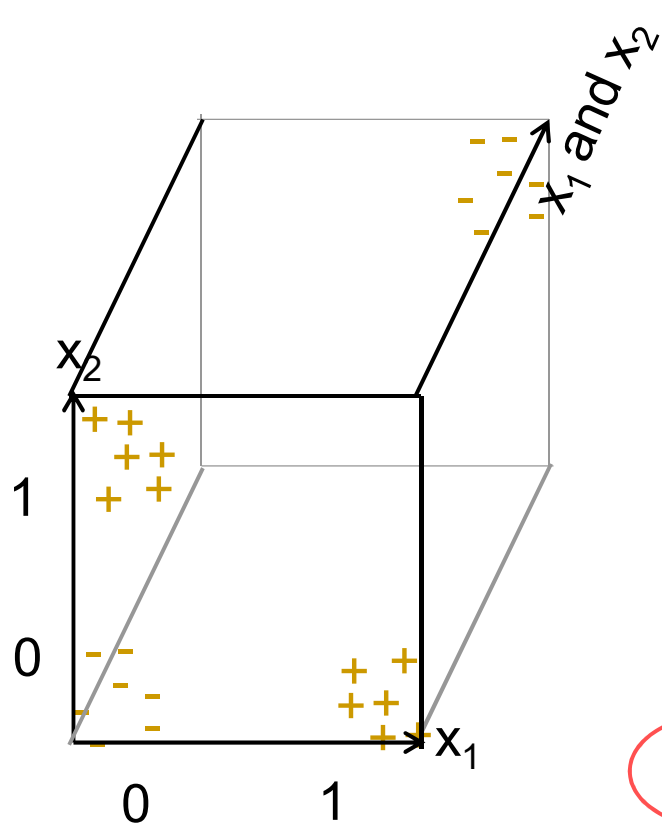
What kind of features to consider?



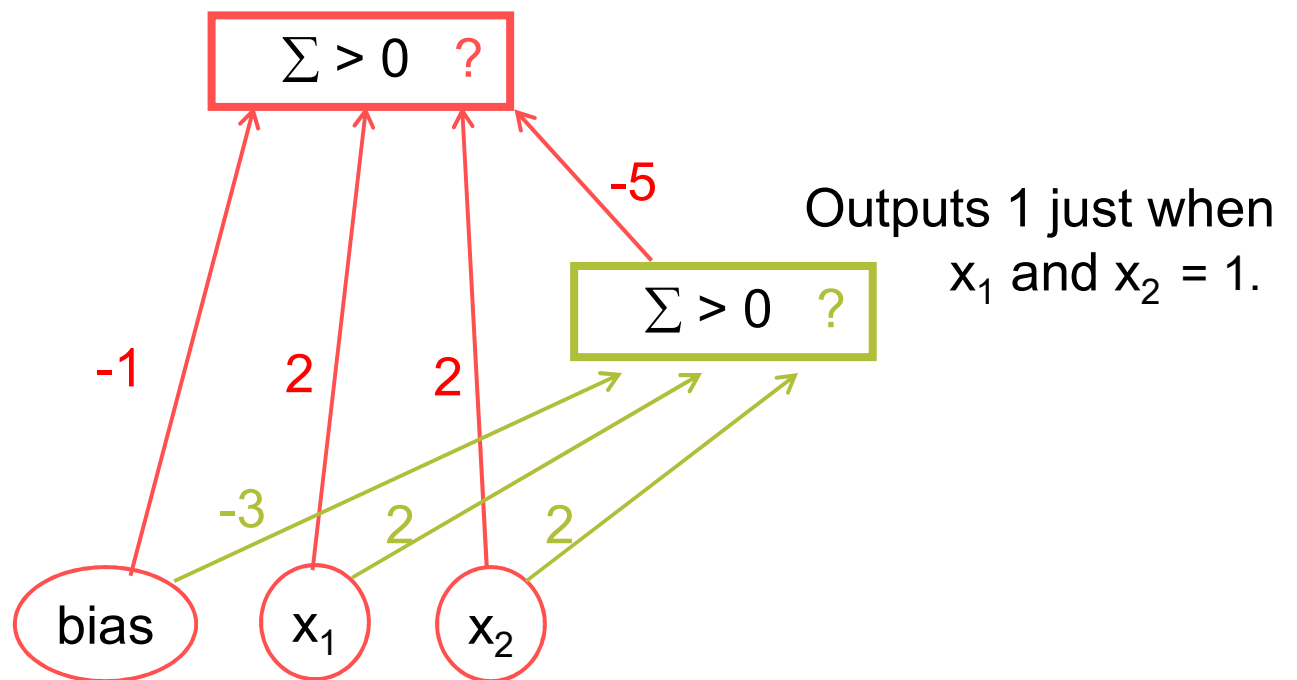
Want to output “+” just
when $x_1 \text{ xor } x_2 = 1$.



Some new features can themselves be computed by linear classifiers



Want to output “+” just
when $x_1 \text{ xor } x_2 = 1$.



A 3D plot illustrating a unit cube with axes x_1 and x_2 . The cube is divided into two regions: a front face (labeled 0) and a back face (labeled 1). The front face contains several yellow '+' markers, and the back face contains several yellow '-' markers. The axes are labeled x_1 and x_2 , and the faces are labeled 0 and 1.

Diagram illustrating a neuron model. Inputs x_1 and x_2 are combined with a bias to calculate the output. The diagram shows the following values and connections:

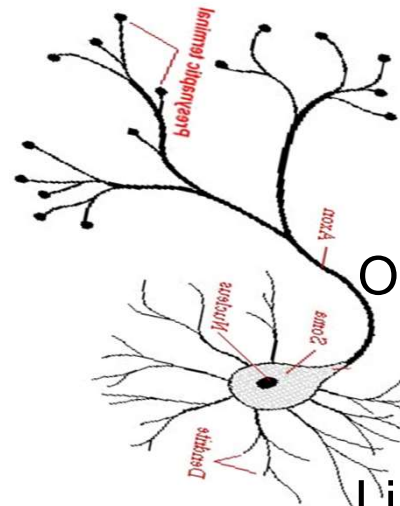
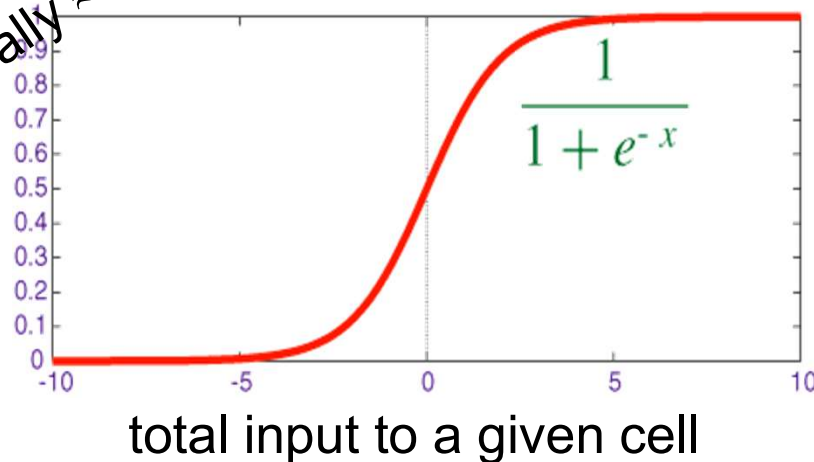
- Inputs: x_1 and x_2 (circled in red).
- Weights: x_1 is weighted by 2 (green arrow), and x_2 is weighted by 2 (green arrow).
- Bias: The bias is -1 (red arrow).
- Summation: The weighted inputs and bias are summed to produce the output. The diagram shows a red box containing $\Sigma > 0$ and a red question mark.
- Output: The output is labeled "Output" and is shown as a neuron with a dendrite and an axon. The output is also labeled "Output" and "Likely neuron other tends".

Outputs 1 just when x_1 and $x_2 = 1$.

Like a biological neuron (brain cell or other nerve cell), which tends to “fire” (spike in electrical output) only if it gets enough total electrical input.

Some new features can themselves be computed by linear classifiers

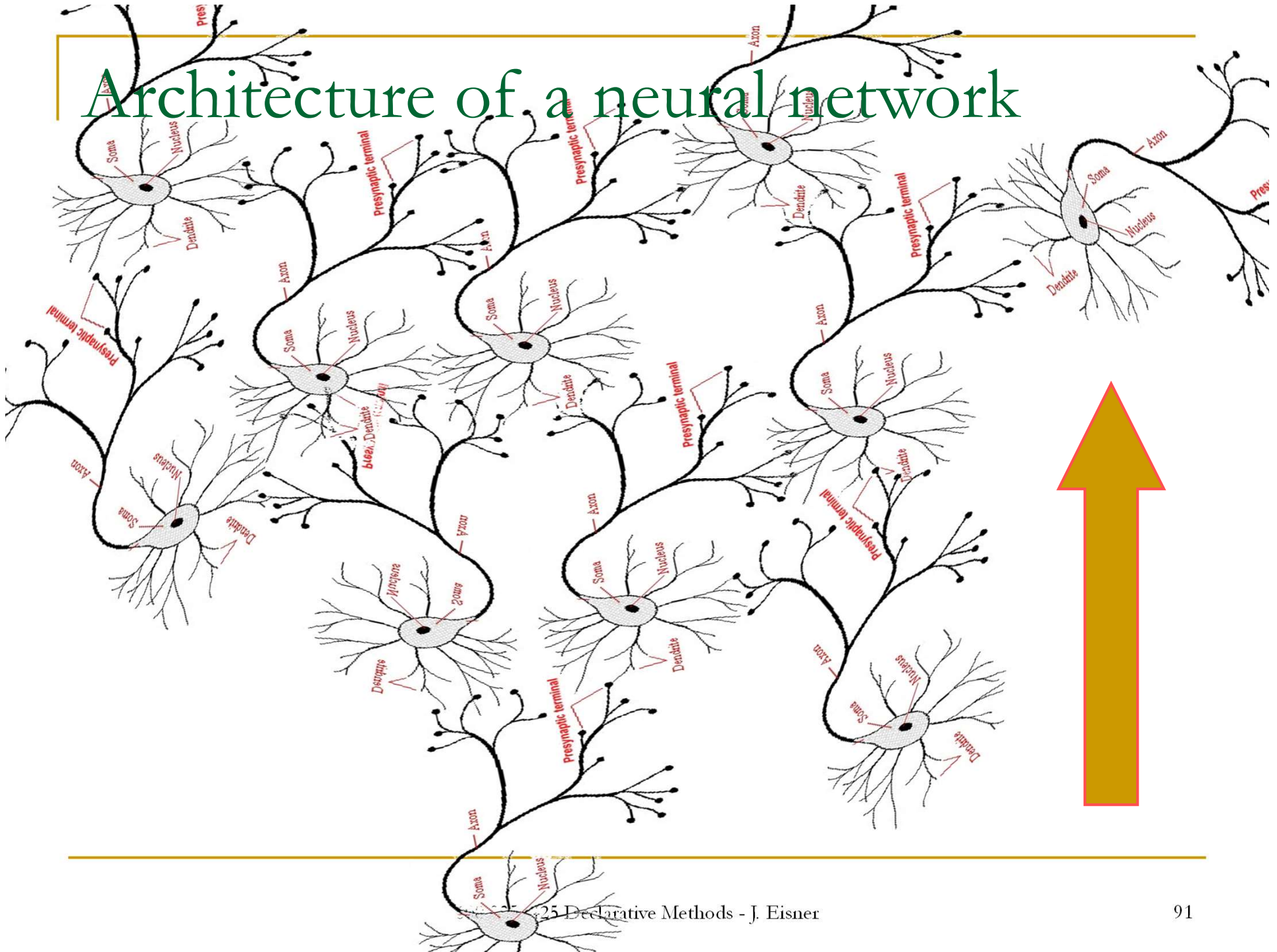
cell's output
(usually ≈ 0 or 1)



Outputs 1 just when x_1 and $x_2 = 1$.

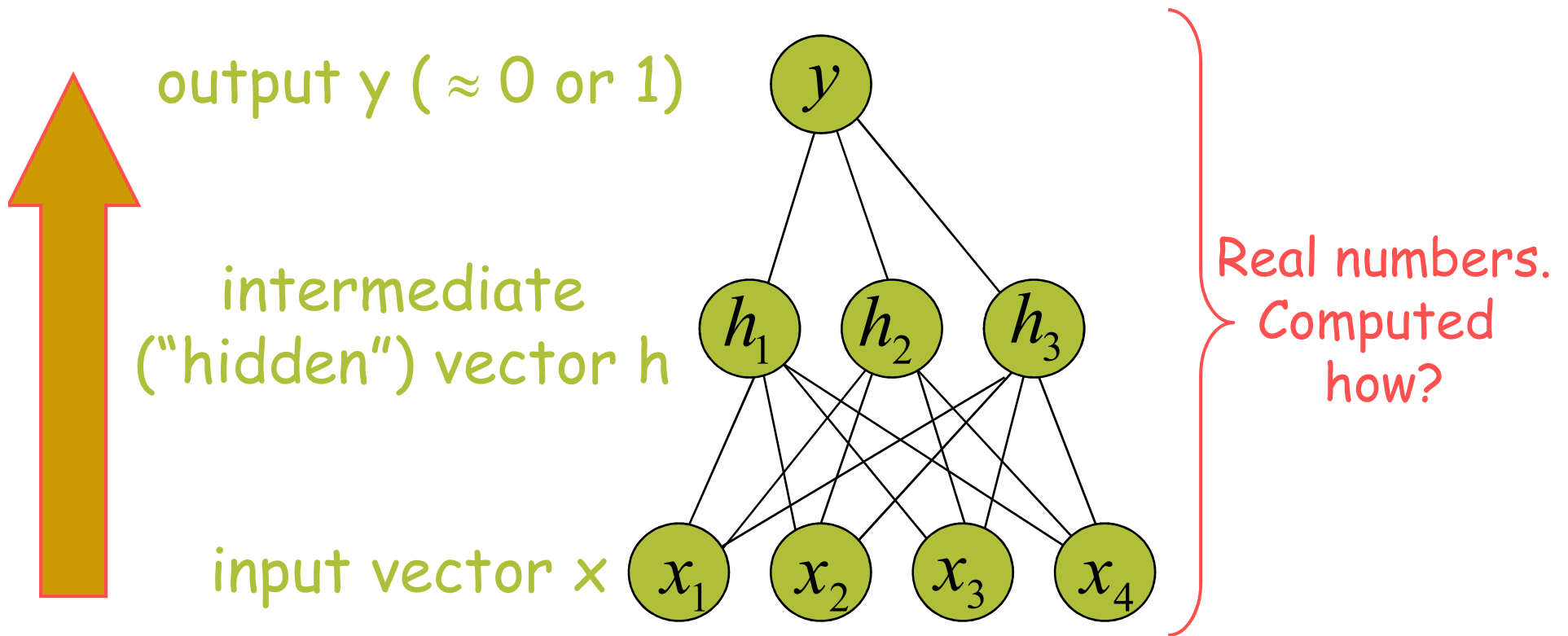
Like a biological neuron (brain cell or other nerve cell), which tends to “fire” (spike in electrical output) only if it gets enough total electrical input.

Architecture of a neural network



Architecture of a neural network

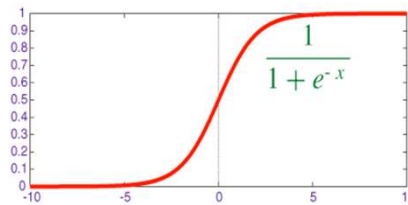
(a basic “multi-layer perceptron” – there are other kinds)



Small example ... often x and h are much longer vectors

Architecture of a neural network

(a basic “multi-layer perceptron” – there are other kinds)



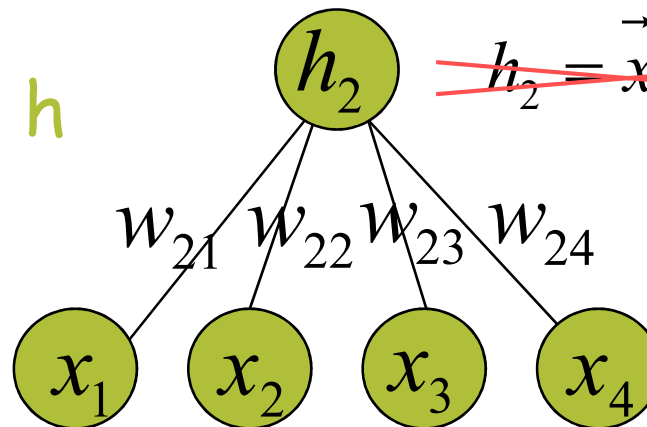
$$h_2 = 1 / (1 + \exp(-\vec{x} \cdot \vec{w}_2))$$

~~$$h_2 = \begin{cases} 0 & \text{if } \vec{x} \cdot \vec{w}_2 < 0 \\ 1 & \text{if } \vec{x} \cdot \vec{w}_2 > 0 \end{cases}$$~~ not differentiable

~~$$h_2 = \vec{x} \cdot \vec{w}_2$$~~ only linear

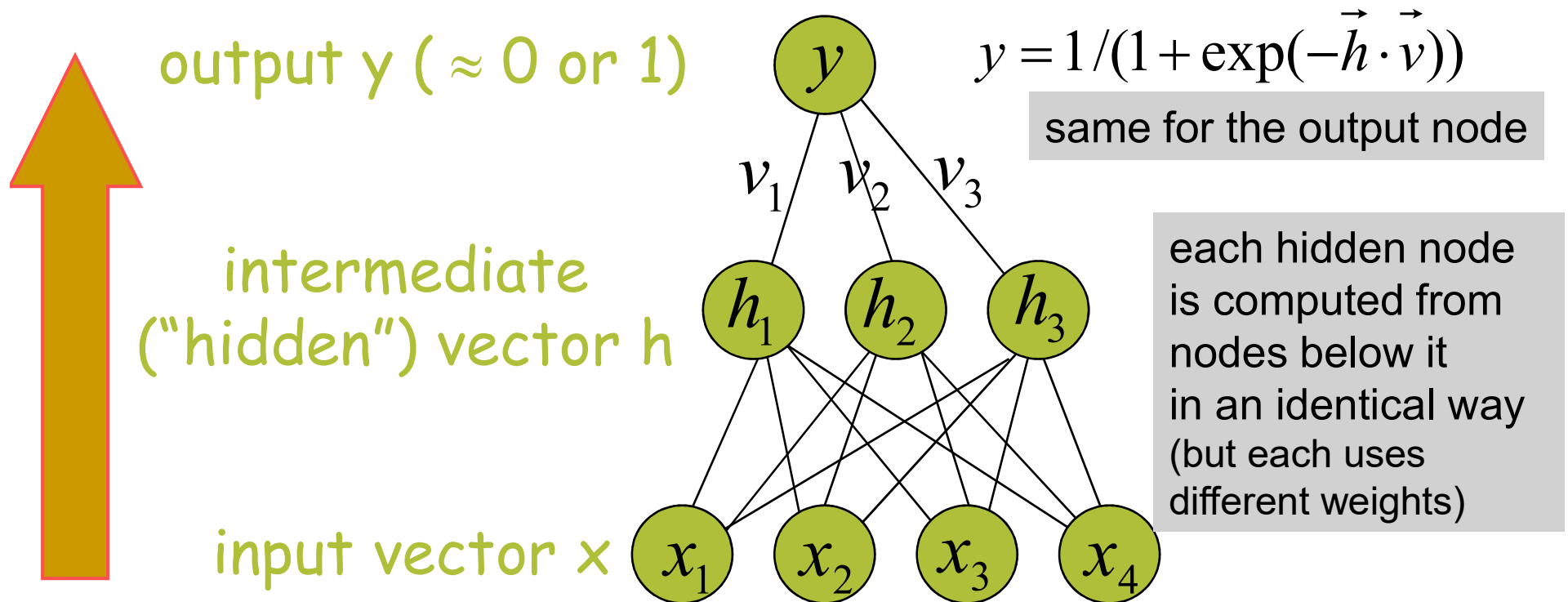
intermediate
("hidden") vector h

input vector x



Architecture of a neural network

(a basic “multi-layer perceptron” – there are other kinds)



Architecture of a neural network

(a basic “multi-layer perceptron” – there are other kinds)

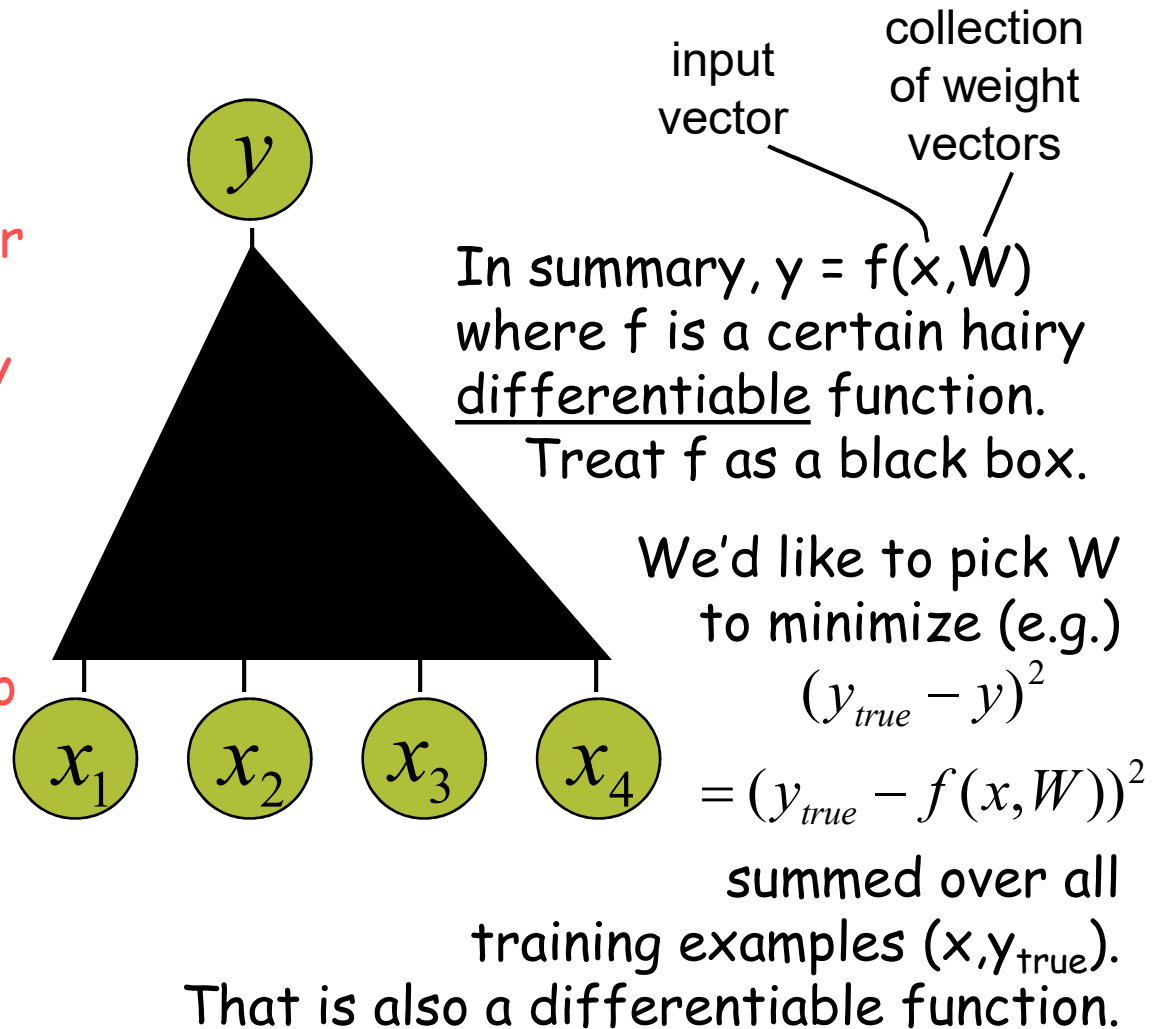
Basic question in differentiating f :

For each weight (w_{23} or v_3), if we increased it by ε , how much would y increase or decrease?

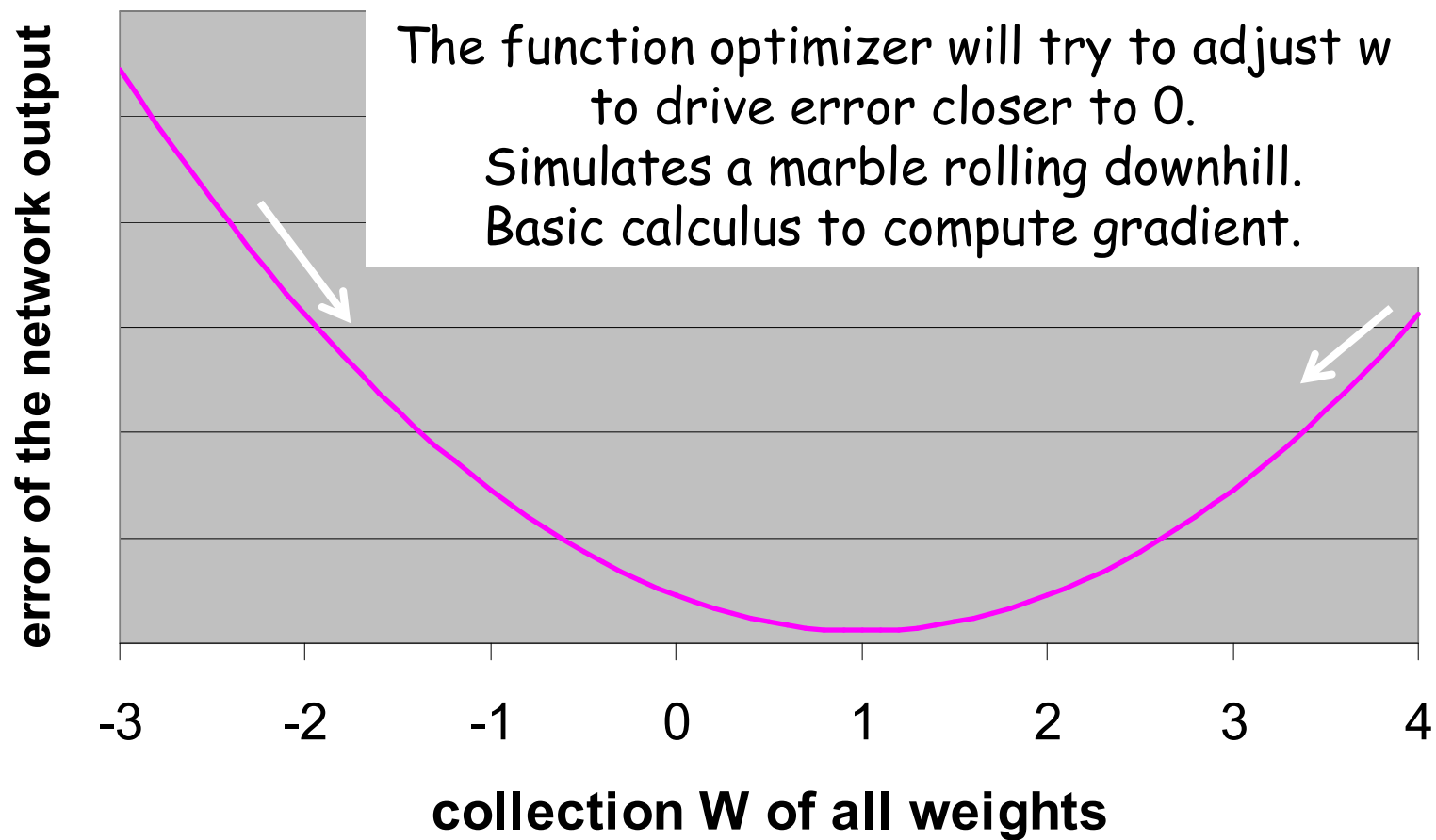
This is exactly a Calc II question!

(What would happen to h_1, h_2, h_3 ? And how would those changes affect y ?

Can easily compute all relevant #s top-down: “back-propagation.”)

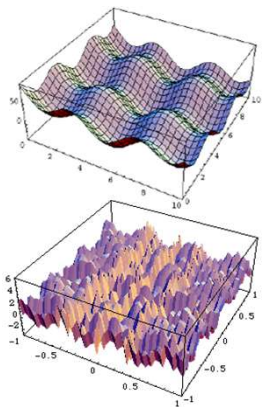


How do you train a neural network?



How do you train a neural network?

- Minimize the loss function, just as before ...
- Use your favorite function minimization algorithm.
 - gradient descent, conjugate gradient, variable metric, etc.



nasty non-differentiable cost
function with local minima

nice smooth and convex cost
function: pick one of these

Uh-oh ...

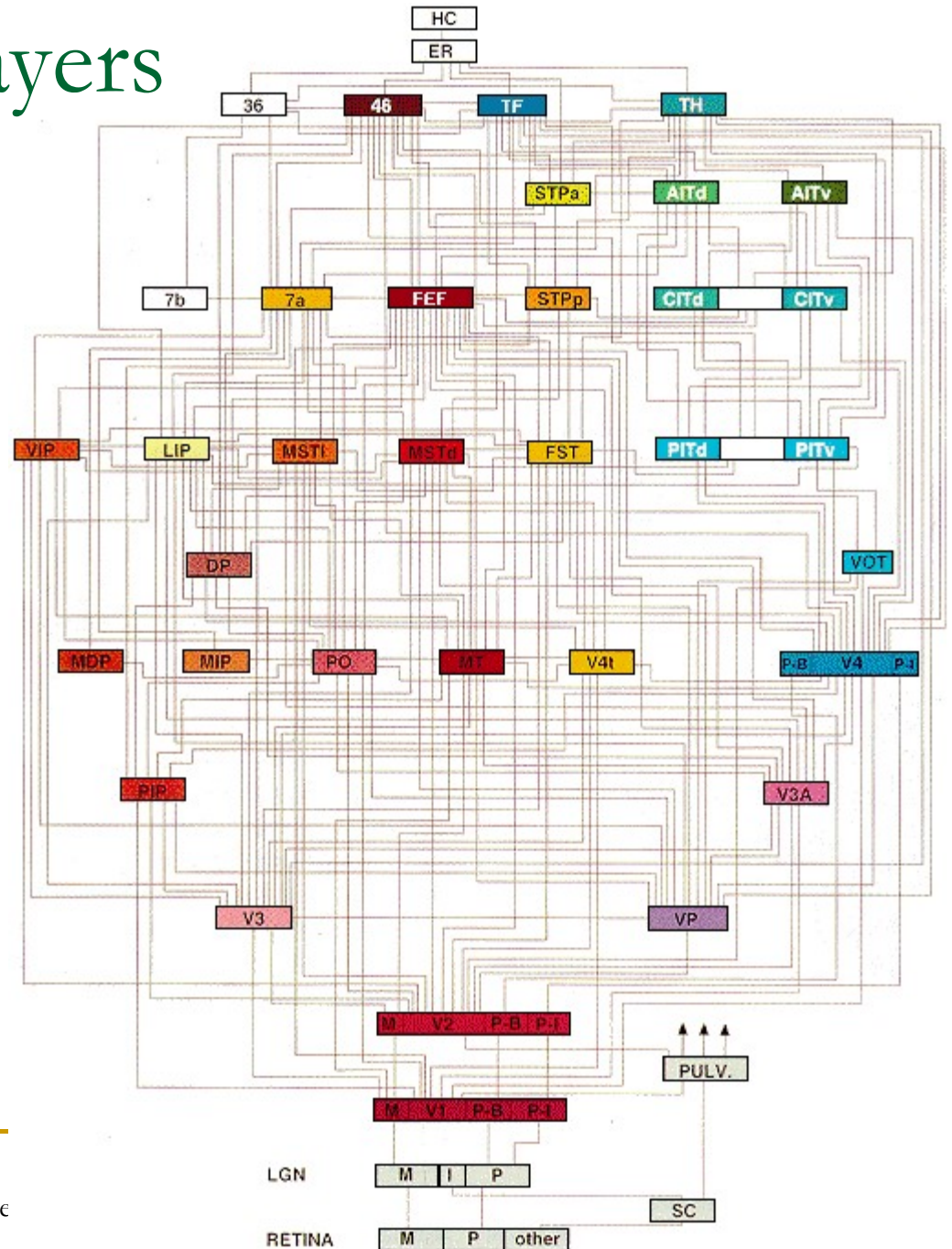
We made the cost function differentiable by using sigmoids
But it's still very bumpy (*sigh*)

You can use neural nets to solve SAT, so they must be hard

Multiple hidden layers

The human visual system is a feedforward hierarchy of neural modules

Roughly, each module is responsible for a certain function

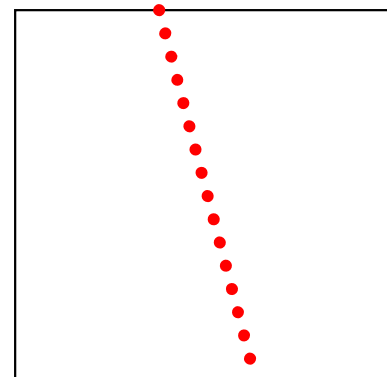
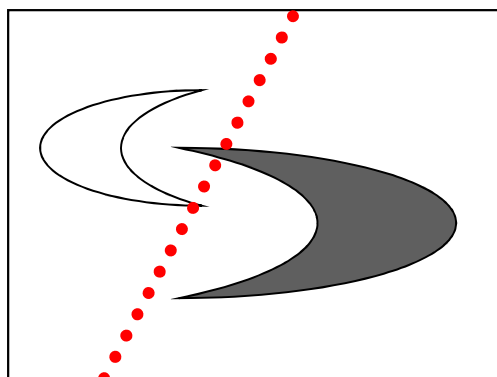
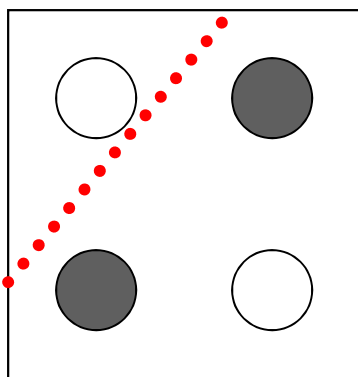
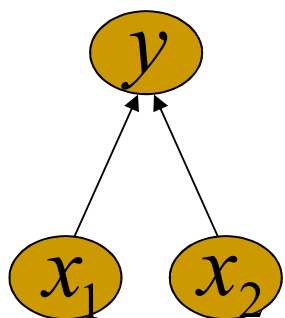


600.325/425 De

slide thanks to Eric Postma (modified)

Decision boundaries of neural nets

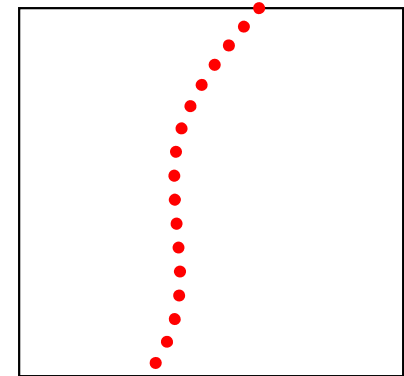
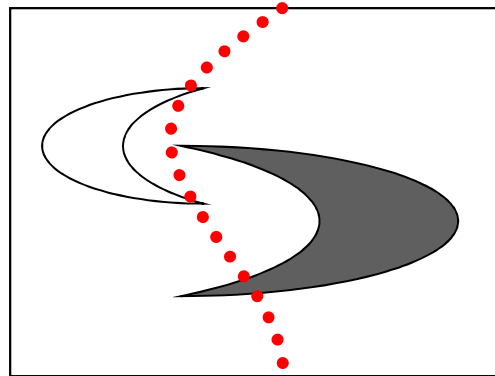
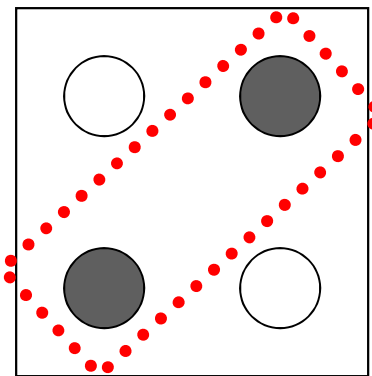
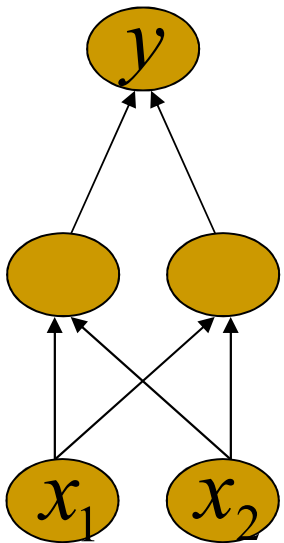
0 hidden layers: straight lines (hyperplanes)



Decision boundaries of neural nets

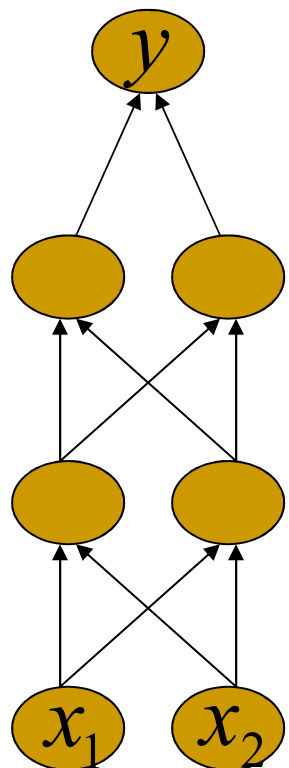
(found this on the web, haven't checked it)

1 hidden layer: boundary of convex region
(open or closed)

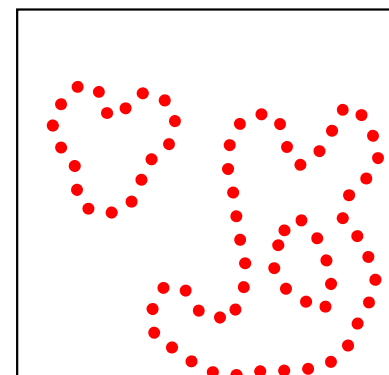
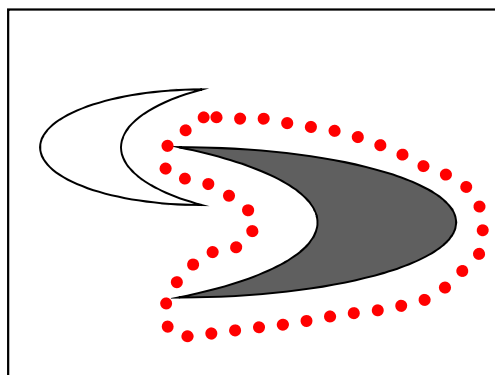
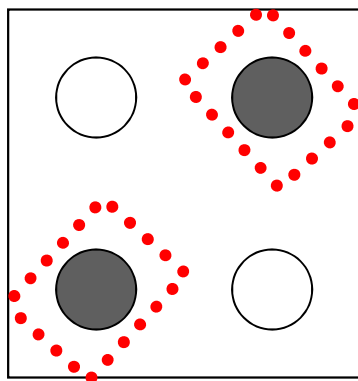


Decision boundaries of neural nets

(found this on the web, haven't checked it)

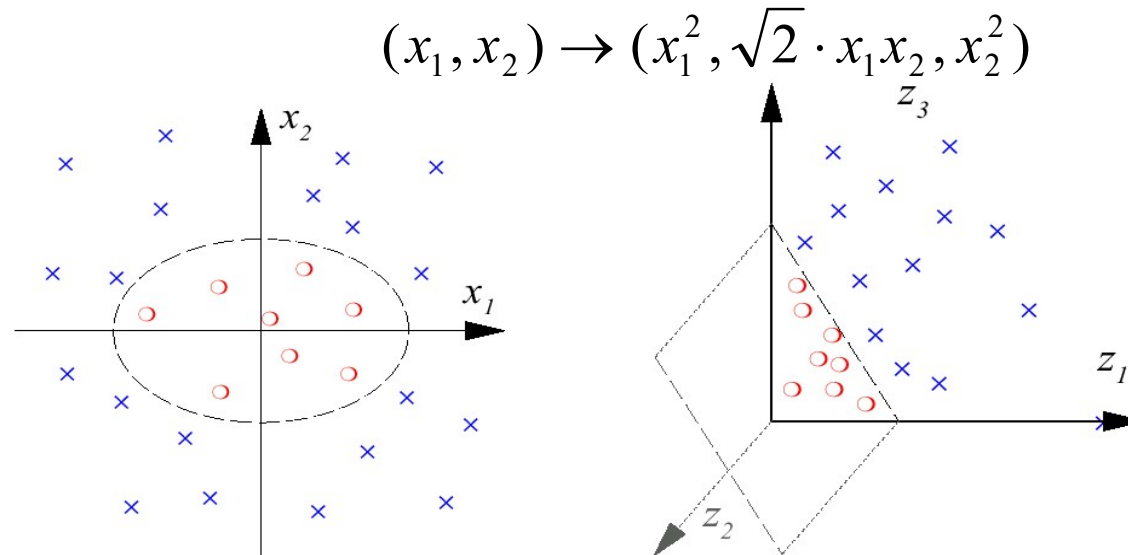


2 hidden layers: combinations of convex regions



Kernel methods

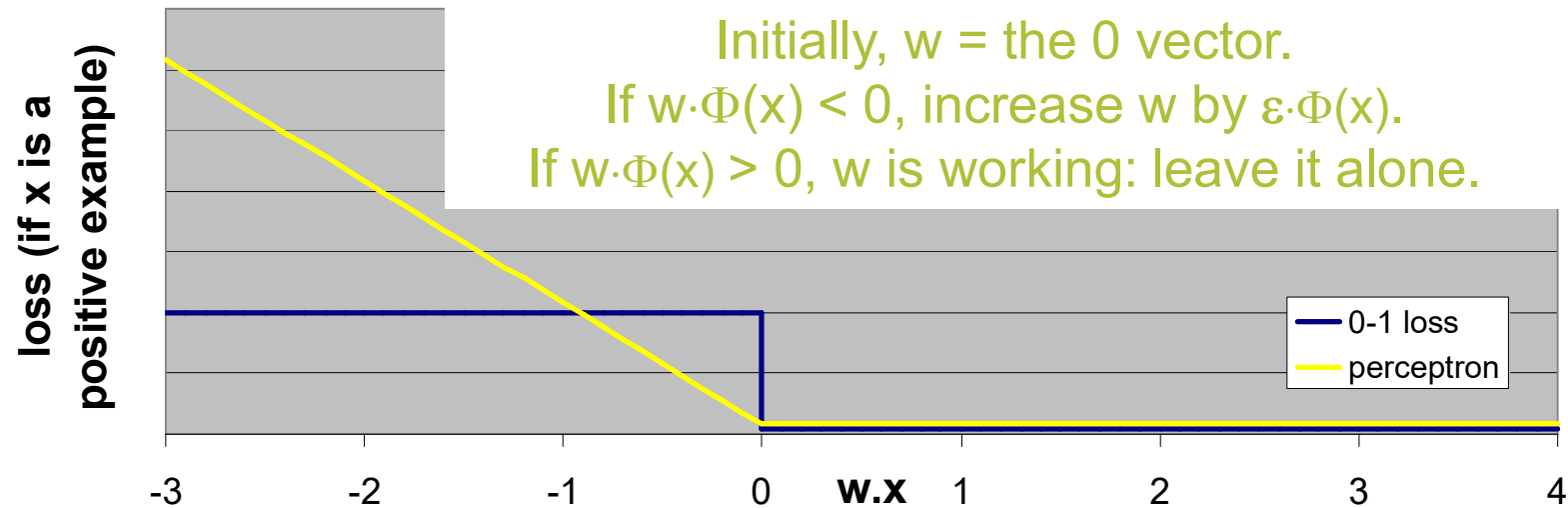
- Neural network uses a smallish number of **learned** features (the hidden nodes).
- An alternative is to throw in a large number of **standard** features: e.g., products of pairs and triples of the original features.



Kernel methods

- Neural network uses a smallish number of **learned** features (the hidden nodes).
- An alternative is to throw in a large number of **standard** features: e.g., products of pairs and triples of the original features. (Or quadruples, or quintuples ...)
- **But this seems to lead to a big problem:**
 - With quintuples, 256 features \rightarrow about 10^{10} features.
 - Won't this make the algorithm really, really slow?
 - And how the heck will we accurately learn 10^{10} coefficients (the length of the weight vector w) from a small training set? Won't this lead to horrible overfitting?

Why don't we overfit when learning 10^{10} coefficients from a small training set?



- Remember the perceptron algorithm? How does it change w ?
- Only ever by adding a multiple of some training example $\Phi(x_i)$.
- So w ends up being a **linear combination of training examples!**
- Thus, are we really free to choose any 10^{10} numbers to describe w ?
- Not for a normal-size training set ... we could represent w much more concisely by storing each x_i and a coefficient α_i . Then $w = \sum_i \alpha_i \Phi(x_i)$.

Why don't we overfit when learning 10^{10} coefficients from a small training set?

- Remember the perceptron algorithm? How does it change w ?
 - Only ever by adding a multiple of some training example $\Phi(x_i)$.
 - **So w ends up being a linear combination of training examples!**
 - Thus, are we really free to choose any 10^{10} numbers to describe w ?
 - Not for a normal-size training set ... we could represent w much more concisely by storing each x_i and a coefficient α_i . Then $w = \sum_i \alpha_i \Phi(x_i)$.
-
- Small training set \rightarrow less complex hypothesis space. (Just as for nearest neighbor.) Good!
 - Better yet, α_i is often 0 (if we got x_i right all along).
 - All of this also holds for SVMs. (If you looked at the SVM learning algorithm, you'd see w was a linear combination, with $\alpha_i \neq 0$ only for the support vectors.)

How about speed with 10^{10} coefficients?

- What computations needed for perceptron, SVM, etc?
- Testing: Evaluate $w \cdot \Phi(x)$ for test example x .
- Training: Evaluate $w \cdot \Phi(x_i)$ for training examples x_i .
- We are storing w as $\sum_i \alpha_i \Phi(x_i)$.
- Can we compute with it that way too?
- Rewrite $w \cdot \Phi(x) = (\sum_i \alpha_i \Phi(x_i)) \cdot \Phi(x)$
 $= \sum_i \alpha_i (\Phi(x_i) \cdot \Phi(x))$
- So all we need is a fast indirect way to get $(\Phi(x_i) \cdot \Phi(x))$ **without** computing huge vectors like $\Phi(x)$...

The kernel trick

- Define a “kernel function” $k(a,b)$ that computes $(\Phi(a) \cdot \Phi(b))$
- Polynomial kernel: $k(a,b) = (a \cdot b)^2$

$$\begin{aligned} k(a,b) &= (a \cdot b)^2 = (a_1 b_1 + a_2 b_2)^2 \\ &= (a_1^2 b_1^2 + a_2^2 b_2^2 + 2a_1 a_2 b_1 b_2) \\ &= (a_1^2, a_2^2, \sqrt{2}a_1 a_2) \cdot (b_1^2, b_2^2, \sqrt{2}b_1 b_2) \\ &= \Phi(a) \cdot \Phi(b) \text{ for the } \Phi \text{ we used for ellipse example} \end{aligned}$$

- How about $(a \cdot b)^3$, $(a \cdot b)^4$? What Φ do these correspond to?
- How about $(a \cdot b + 1)^2$?
- Whoa. Does **every** random function $k(a,b)$ have a Φ ?

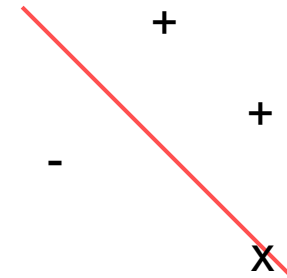
What kernels can I use?

- Given an arbitrary function $k(a,b)$, does it correspond to $\Phi(a) \cdot \Phi(b)$ for some Φ ?
 - Yes, **if** $k(a,b)$ is symmetric and meets the so-called Mercer condition. Then $k(a,b)$ is called a “kernel” and we can use it.
 - Sums and products of kernels with one another, and with constants, are also kernels.
 - Some kernels correspond to weird Φ such that $\Phi(a)$ is infinite-dimensional. That is okay – we never compute it! We just use the kernel to get $\Phi(a) \cdot \Phi(b)$ directly.
 - Example: Gaussian Radial Basis Function (RBF) kernel:

$$k(\vec{a}, \vec{b}) = \exp(\|\vec{a} - \vec{b}\|^2 / 2\sigma^2)$$

How does picking a kernel influence the decision boundaries that can be learned?

- What do the decision boundaries look like in the original space?
- Curve defined by x such that $w \cdot \Phi(x) = 0$
 - for quadratic kernel, a quadratic function of $(x_1, x_2, \dots) = 0$
- Equivalently, x such that $\sum_i \alpha_i k(x_i, x) = 0$
 - where $\alpha_i > 0$ for positive support vectors,
 $\alpha_i < 0$ for negative support vectors
 - so x is on the decision boundary
if its “similarity” to positive support vectors
balances its “similarity” to negative support vectors,
weighted by fixed coefficients α_i .



How does picking a kernel influence the decision boundaries that can be learned?

- x is on the decision boundary if $\sum_i \alpha_i k(x_i, x) = 0$

- i.e., if its “similarity” to positive support vectors balances its “similarity” to negative support vectors.

- Is $\alpha_i k(x_i, x)$ really a “similarity” between x_i and x ?

- Kind of: remember $\alpha_i k(x_i, x) = \alpha_i \Phi(x_i) \cdot \Phi(x)$
 $= \alpha_i \underbrace{\|\Phi(x_i)\|}_{\text{who cares? } \alpha_i \text{ could be such that these cancel out (if not, different support vectors have different weights)}} \underbrace{\|\Phi(x)\|}_{\text{constant factor for a given } x; \text{ (dropping it doesn't affect whether } \sum_i \alpha_i k(x_i, x) = 0\text{)}} \underbrace{\cos(\text{angle between vectors } \Phi(x_i), \Phi(x))}_{\text{a measure of similarity!}}$

who cares? α_i could be such that these cancel out (if not, different support vectors have different weights)

constant factor for a given x ; (dropping it doesn't affect whether $\sum_i \alpha_i k(x_i, x) = 0$)

a measure of similarity!
1 if high-dim vectors point in same direction,
-1 if they point in opposite directions.
The other factors don't matter so much...

Visualizing SVM decision boundaries ...

- <http://www.cs.jhu.edu/~jason/tutorials/SVMApplet/> (by Guillaume Caron)
- Use the mouse to plot your own data, or try a standard dataset
- Try a bunch of different kernels
- The original data here are only in 2-dimensional space, so the decision boundaries are easy to draw.
- If the original data were in 3-dimensional or 256-dimensional space, you'd get some wild and wonderful curved hypersurfaces.