CIS 519/419
Applied Machine Learning
www.seas.upenn.edu/~cis519

Dan Roth

danroth@seas.upenn.edu

http://www.cis.upenn.edu/~danroth/

461C, 3401 Walnut

Slides were created by Dan Roth (for CIS519/419 at Penn or CS446 at UIUC), Eric Eaton for CIS519/419 at Penn, or from other authors who have made their ML slides available.

Administration

- Registration ☺
- Hw1 is due next week
  - You should have started working on it already…
  - Recall that this is an Applied Machine Learning class.
  - We are not asking you to simply give us back what you've seen in class.
  - The HW will try to simulate challenges you might face when you want to apply ML.
  - Allow you to experience various ML scenarios and make observations that are best experienced when you play with it yourself.
- Hw2 will be out next week

Questions

Projects

https://www.seas.upenn.edu/~cis519/fall2018/project.html

- CIS 519 students need to do a team project
  - Teams will be of size 2-3
- Projects proposals are due on Friday 10/26/18
  - Details will be available on the website
  - We will give comments and/or requests to modify / augment/ do a different project.
  - There may also be a mechanism for peer comments.
- Please start thinking and working on the project now.
  - Your proposal is limited to 1-2 pages, but needs to include references and, ideally, some preliminary results/ideas.
- Any project with a significant Machine Learning component is good.
  - Experimental work, theoretical work, a combination of both or a critical survey of results in some specialized topic.
  - The work has to include some reading of the literature .
  - Originality is not mandatory but is encouraged.
- Try to make it interesting!

Project Examples

Just going to Kaggle to take a dataset and running algorithms on it is not enough.

- KDD Cup 2013:
  - "Author-Paper Identification": given an author and a small set of papers, we are asked to identify which papers are really written by the author.
    - https://www.kaggle.com/c/kdd-cup-2013-author-paper-identification-challenge
  - "Author Profiling": given a set of document, profile the author: identification, gender, native language, ….
- File my e-mail to folders better than Apple does
- Caption Control: Is it gibberish? Spam? High quality text?
- Adapt an NLP program to a new domain
- Work on making learned hypothesis more comprehensible
  - Explain the prediction
- Develop a (multi-modal) People Identifier
- Identify contradictions in news stories
- Large scale clustering of documents + name the cluster
  - E.g., cluster news documents and give a title to the document

You need to have a "thesis".
Example: classify internal organisms;
Thesis: can generalize across gender, other populations.

A Guide

Today:
Take a more general perspective and think more about learning, learning protocols, quantifying performance, etc.
This will motivate some of the ideas we will see next.

- Learning Algorithms
  - (Stochastic) Gradient Descent (with LMS)
  - Decision Trees
- ✓ Importance of hypothesis space (representation)
- How are we doing?
  - Quantification in terms of cumulative # of mistakes
  - Our algorithms were driven by a different metric than the one we care about.
- Today: Versions of Perceptron
  - How to deal better with large features spaces & sparsity?
  - Variations of Perceptron
    - Dealing with overfitting
  - Closing the loop: Back to Gradient Descent
  - Dual Representations & Kernels
- Multilayer Perceptron
- Beyond Binary Classification?
  - Multi-class classification and Structured Prediction
- More general way to quantify learning performance (PAC)
  - New Algorithms (SVM, Boosting)

Quantifying Performance

- We want to be able to say something rigorous about the performance of our learning algorithm.

- We will concentrate on discussing the number of examples one needs to see before we can say that our learned hypothesis is good.

Learning Conjunctions

- There is a hidden (monotone) conjunction the learner (you) is to learn

$$f(x_1, x_2,\ldots,x_{100}) = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$$

- How many examples are needed to learn it ? How ?
  - Protocol I: The learner proposes instances as queries to the teacher
  - Protocol II: The teacher (who knows f) provides training examples
  - Protocol III: Some random source (e.g., Nature) provides training examples; the Teacher (Nature) provides the labels (f(x))

Learning Conjunctions (I)

- Protocol I: The learner proposes instances as queries to the teacher
- Since we know we are after a monotone conjunction:
- Is x_{100} in? <(1,1,1…,1,0), ?> f(x)=0 (conclusion: Yes)
- Is x_{99} in? <(1,1,…1,0,1), ?> f(x)=1 (conclusion: No)
- Is x_1 in ? <(0,1,…1,1,1), ?> f(x)=1 (conclusion: No)

- A straight forward algorithm requires n=100 queries, and will produce as a result the hidden conjunction (exactly).
  - $h(x_1, x_2,\ldots,x_{100}) = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

What happens here if the conjunction is not known to be monotone?
If we know of a positive example, the same algorithm works.

Learning Conjunctions(II)

- Protocol II: The teacher (who knows f) provides training examples

Learning Conjunctions (II)

- Protocol II: The teacher (who knows f) provides training examples
- <(0,1,1,1,1,0,…,0,1), 1>

Learning Conjunctions (II)

- Protocol II: The teacher (who knows f) provides training examples
- <(0,1,1,1,1,0,…,0,1), 1> (We learned a superset of the good variables)

Learning Conjunctions (II)

- Protocol II: The teacher (who knows f) provides training examples
- <(0,1,1,1,1,0,…,0,1), 1> (We learned a superset of the good variables)
- To show you that all these variables are required…

Learning Conjunctions (II)

- Protocol II: The teacher (who knows f) provides training examples
- <(0,1,1,1,1,0,…,0,1), 1> (We learned a superset of the good variables)
- To show you that all these variables are required…
  - <(0,0,1,1,1,0,…,0,1), 0> need x_2
  - <(0,1,0,1,1,0,…,0,1), 0> need x_3
  - …..
  - <(0,1,1,1,1,0,…,0,0), 0> need x_{100}

Modeling Teaching Is tricky

- A straight forward algorithm requires k = 6 examples to produce the hidden conjunction (exactly).

$$h(x_1, x_2,\ldots,x_{100}) = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$$

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
- Teacher (Nature) provides the labels (f(x))
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0>
  - <(1,1,1,1,1,0,...0,1,1), 1>
  - <(1,0,1,1,1,0,...0,1,1), 0>
  - <(1,1,1,1,1,0,...0,0,1), 1>
  - <(1,0,1,0,0,0,...0,1,1), 0>
  - <(1,1,1,1,1,1,…,0,1), 1>
  - <(0,1,0,1,0,0,...0,1,1), 0>
- How should we learn?
- Skip

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0>

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0> ← learned nothing: $h = x_1 \wedge x_2 \wedge \ldots \wedge x_{100}$
  - <(1,1,1,1,1,0,…0,1,1), 1>

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0> ← learned nothing: $h = x_1 \wedge x_2 \wedge \ldots \wedge x_{100}$
  - <(1,1,1,1,1,0,...0,1,1), 1> ← $h = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{99} \wedge x_{100}$

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0> ← learned nothing
  - <(1,1,1,1,1,0,...0,1,1), 1> ← $h = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{99} \wedge x_{100}$
  - <(1,0,1,1,0,0,...0,0,1), 0> ← learned nothing
  - <(1,1,1,1,1,0,...0,0,1), 1>

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0> ← learned nothing
  - <(1,1,1,1,1,0,...0,1,1), 1> ← $h = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{99} \wedge x_{100}$
  - <(1,0,1,1,0,0,...0,0,1), 0> ← learned nothing
  - <(1,1,1,1,1,0,...0,0,1), 1> ← $h = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0> learned nothing
  - <(1,1,1,1,1,0,...0,1,1), 1>
  - <(1,0,1,1,0,0,...0,0,1), 0> learned nothing
  - <(1,1,1,1,1,0,...0,0,1), 1> ← $h = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$
  - <(1,0,1,0,0,0,...0,1,1), 0>
  - <(1,1,1,1,1,1,…,0,1), 1>
  - <(0,1,0,1,0,0,...0,1,1), 0>

Learning Conjunctions(III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: Elimination
  - Start with the set of all literals as candidates
  - Eliminate a literal that is not active (0) in a positive example
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0> learned nothing
  - <(1,1,1,1,1,0,...0,1,1), 1>
  - <(1,0,1,1,0,0,...0,0,1), 0> learned nothing
  - <(1,1,1,1,1,0,...0,0,1), 1>
  - <(1,0,1,0,0,0,...0,1,1), 0> Final hypothesis:
  - <(1,1,1,1,1,1,…,0,1), 1> $h = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$
  - <(0,1,0,1,0,0,...0,1,1), 0>

- Is it good
- Performance ?
- # of examples ?

Learning Conjunctions (III)

$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

- Protocol III: Some random source (e.g., Nature) provides training examples
  - Teacher (Nature) provides the labels (f(x))
- Algorithm: …….
  - <(1,1,1,1,1,1,…,1,1), 1>
  - <(1,1,1,0,0,0,…,0,0), 0>
  - <(1,1,1,1,1,0,...0,1,1), 1>
  - <(1,0,1,1,0,0,...0,0,1), 0>
  - <(1,1,1,1,1,0,...0,0,1), 1>
  - <(1,0,1,0,0,0,...0,1,1), 0>
  - <(1,1,1,1,1,1,…,0,1), 1>
  - <(0,1,0,1,0,0,...0,1,1), 0>
  - <(0,1,0,1,0,0,...0,1,1), 0>

- Is it good
- Performance ?
- # of examples ?

- With the given data, we only learned an "approximation" to the true concept
- We don't know **how many examples** we need to see to learn **exactly**. (do we care?)
- But we know that we can make a limited **# of mistakes**.

Final hypothesis:

$h = x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$

Two Directions

- Can continue to analyze the probabilistic intuition:
  - Never saw x_1=0 in positive examples, maybe we'll never see it?
  - And if we will, it will be with small probability, so the concepts we learn may be pretty good
  - Good: in terms of performance on future data
  - PAC framework
- Mistake Driven Learning algorithms/On line algorithms
  - Now, we can only reason about #(mistakes), not #(examples)
    - any relations?
  - Update your hypothesis only when you make mistakes
    - Not all on-line algorithms are mistake driven, so performance measure could be different.

On-Line Learning

- New learning algorithms

  (all learn a linear function over the feature space)
  - Perceptron (+ many variations)
  - General Gradient Descent view

- Issues:
  - Importance of Representation
  - Complexity of Learning
  - Idea of Kernel Based Methods
  - More about features

Generic Mistake Bound Algorithms

- Is it clear that we can bound the number of mistakes ?
- Let C be a finite concept class. Learn f $\in$ C
- CON:
  - In the ith stage of the algorithm:
  - C_i all concepts in C consistent with all i-1 previously seen examples
  - Choose randomly f $\in$ C_i and use to predict the next example
  - Clearly, C_{i+1} $\subseteq$ C_i and, if a mistake is made on the ith example, then $|C_{i+1}| < |C_i|$ so progress is made.
- The CON algorithm makes at most |C|-1 mistakes
- Can we do better ?

The goal of the following discussion is to think about hypothesis spaces, and some "optimal" algorithms, as a way to understand what might be possible.
For this reason, here we think about the class of possible target functions and the class of hypothesis as the same.

The Halving Algorithm

- Let C be a concept class. Learn f 2 C
- Algorithm:
- In the ith stage of the algorithm:
  - C_i all concepts in C consistent with all i-1 previously seen examples
- Given an example e_t consider the value $f_j(e_t)$ for all f_j 2 C_i and predict by majority.
- Predict 1 iff

$$|\{f_j \in C_i; f_j(e_i) = 0\}| < |\{f_j \in C_i; f_j(e_i) = 1\}|$$

- Clearly $C_{i+1} \subseteq C_i$ and if a mistake is made in the ith example, then $|C_{i+1}| < 1/2\ |C_i|$
- The Halving algorithm makes at most log(|C|) mistakes
  - Of course, this is a theoretical algorithm; can this ne achieved with an efficient algorithm?

Learning Conjunctions

Can this bound be achieved?

- There is a hidden conjunctions the learner is to learn

$$f(x_1, x_2, \ldots, x_{100}) = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$$

- The number of (all; not monotone) conjunctio
- log(|C|) = n

Can mistakes be bounded in the **non-finite case?**

Earlier:

- Talked about various learning protocols & on algorithms for conjunctions.
- Discussed the performance of the algorithms in terms of bounding the number of mistakes that algorithm makes.
- Gave a "theoretical" algorithm with log|C| mistakes.

Assume that only k << n attributes occur in the disjunction

- The number of k-conjunctions: $\binom{n}{k}2^k$
  - log(|C|) = klog n
  - Can we learn efficiently with this number of mistakes ?

Linear Threshold Functions

$$f(x) = \text{sgn}\{w^T \cdot x - \theta\} = \text{sgn}\{\sum_{i=1}^{n} w_i x_i - \theta\}$$

- Many functions are Linear
  - Conjunctions:

$y = x_1 \wedge x_3 \wedge x_5$

$y = \text{sgn}\{1 \cdot x_1 + 1 \cdot x_3 + 1 \cdot x_5 - 3\}$; $w = (1, 0, 1, 0, 1)$ $\theta = 3$

- In our Elimination Algorithm we started with:

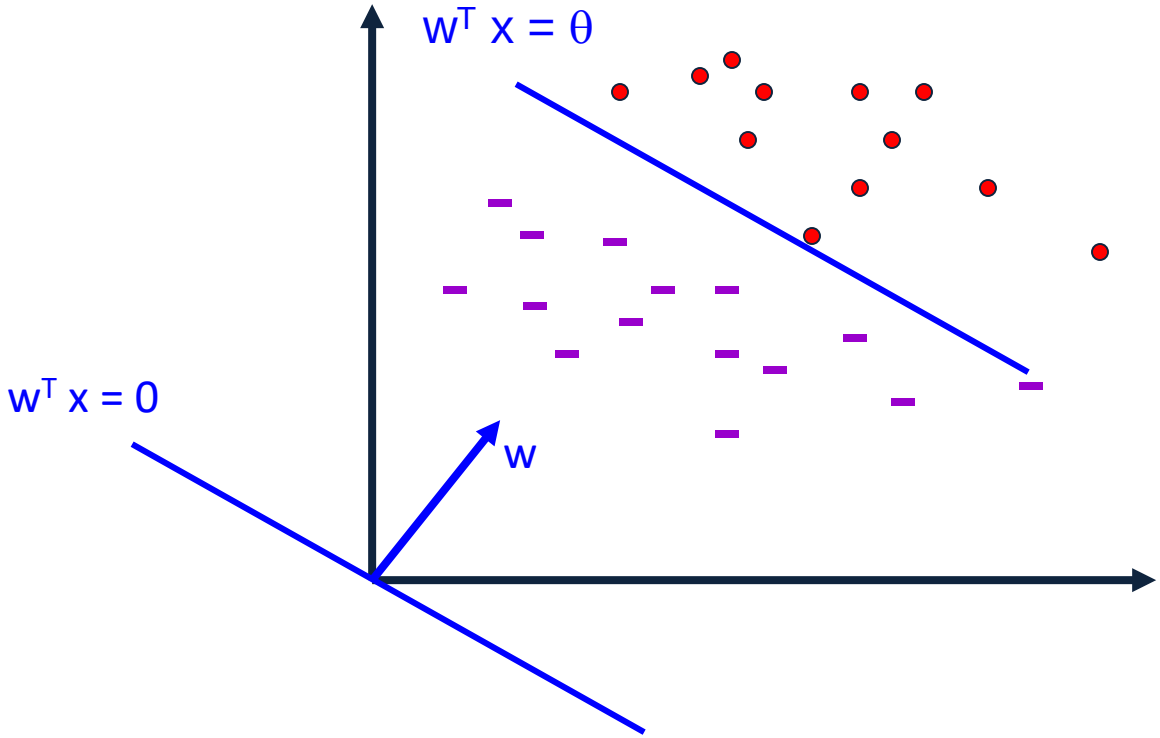
$w = (1, 1, 1, 1, 1, n)$ and changed w_i from $1 \rightarrow 0$ following a mistake

- In general, when learning linear functions, we will change w_i more carefully

$$y = \text{sgn}\{w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + w_4 \cdot x_4 + w_5 \cdot x_5 - \theta\};$$

Representation

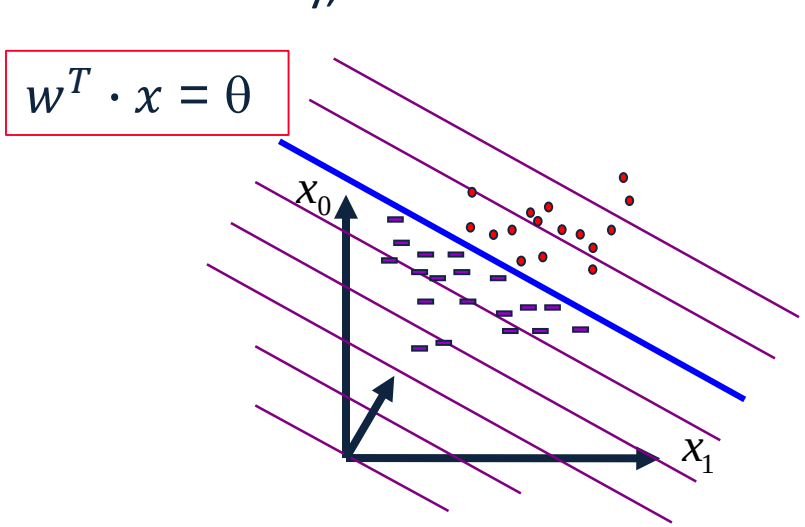
- Assume that you want to learn conjunctions. Should your hypothesis space be the class of conjunctions?
  - **Theorem:** Given a sample on n attributes that is consistent with a conjunctive concept, it is NP-hard to find a pure conjunctive hypothesis that is both consistent with the sample and has the minimum number of attributes.
  - [David Haussler, AIJ'88: "Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework"]
- Same holds for Disjunctions.
- Intuition: Reduction to minimum set cover problem.
  - Given a collection of sets that cover X, define a set of examples so that learning the best (dis/conj)junction implies a minimal cover.
- Consequently, we cannot learn the concept efficiently **as a (dis/con)junction.**
- But, we will see that we can do that, if we are willing to learn the concept as a Linear Threshold function.
- **In a more expressive class, the search for a good hypothesis sometimes becomes combinatorially easier.**

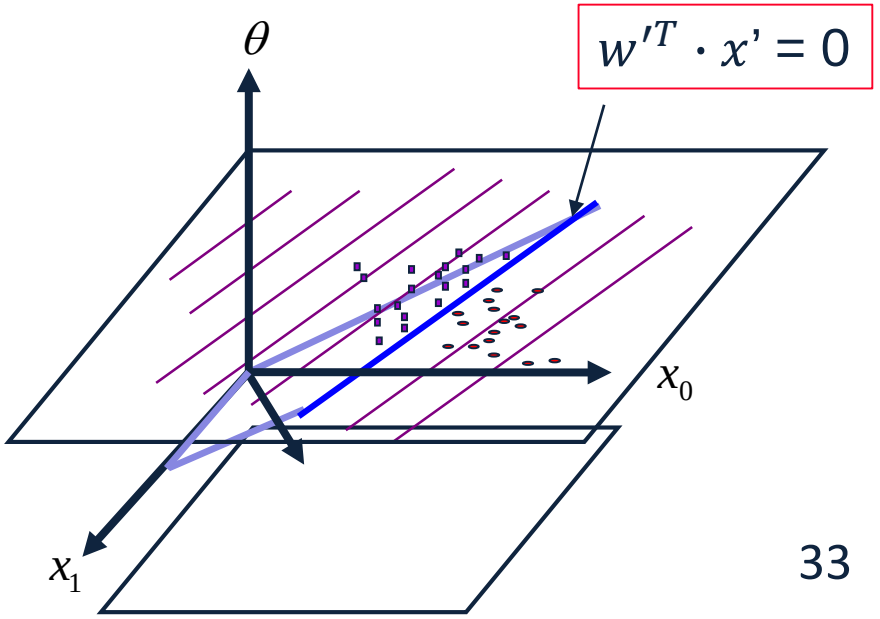

$w^T x = \theta$
$w^T x = 0$
w

Canonical Representation

$$f(x) = \text{sgn}\{w^T \cdot x - \theta\} = \text{sgn}\{\sum_{i=1}^{n} w_i x_i - \theta\}$$

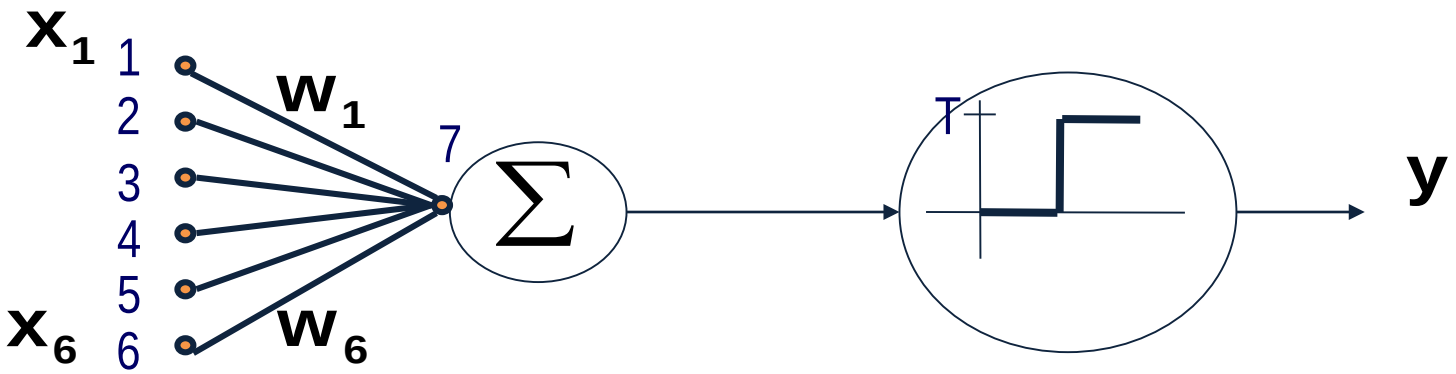
- **Note:** $\text{sgn}\{w^T \cdot x - \theta\} = \text{sgn}\{w'^T \cdot x'\}$
- Where:
  - $x' = (x, -1)$ and $w' = (w, \theta)$
- Moved from an n dimensional representation to an (n+1) dimensional representation, but now can look for hyperplanes that go through the origin.
- Basically, that means that we learn both w and θ





Perceptron learning rule

- On-line, mistake driven algorithm.
- Rosenblatt (1959) suggested that when a target output value is provided for a single neuron with fixed input, it can incrementally change weights and learn to produce the output using the Perceptron learning rule
- (Perceptron == Linear Threshold Unit)



Perceptron learning rule

- We learn f:X$\rightarrow${-1,+1} represented as f =sgn{$w^T \bullet x$)
- Where X= {0,1}^{n} or X= R^n and w$\in$ R^n
- Given Labeled examples: {(x_1, y_1), (x_2, y_2),…(x_m, y_m)}

1. Initialize w=0$\in \mathbf{R}^n$
2. Cycle through all examples [multiple times]

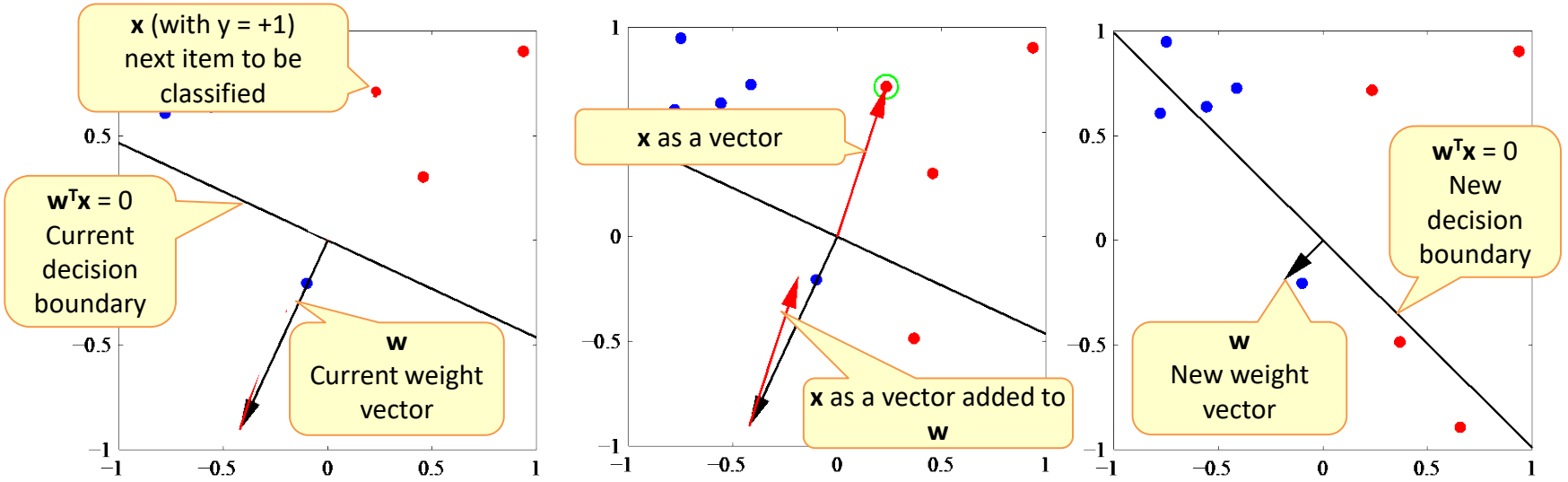
   a. Predict the label of instance x to be y' = sgn{$w^T \bullet x$)

   b. If y'$\neq$y, update the weight vector:

   w = w + r y x (r - a constant, learning rate)

   Otherwise, if y'=y, leave weights unchanged.

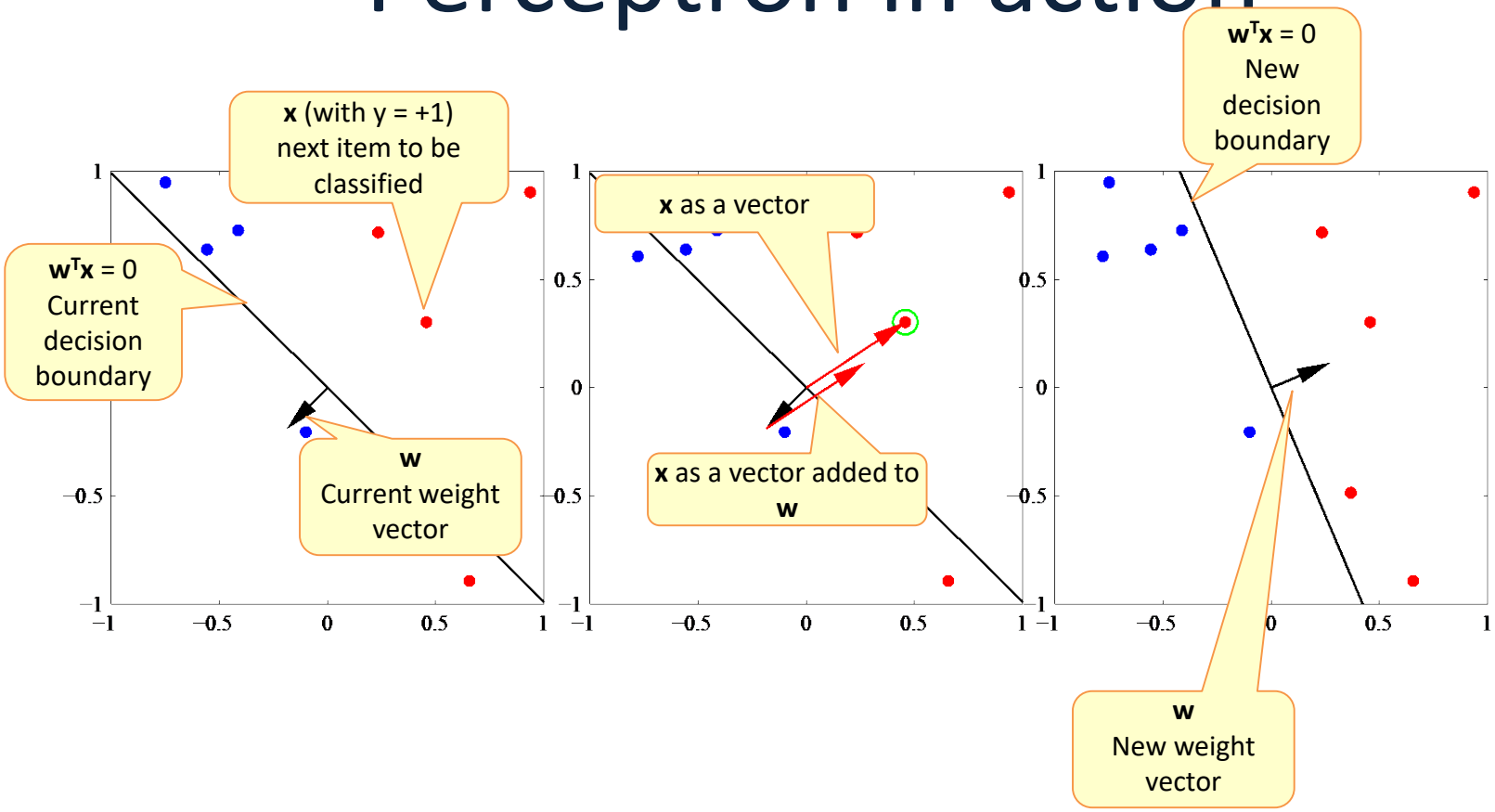
Perceptron in action



(Figures from Bishop 2006)

- Positive
- Negative

Perceptron in action



(Figures from Bishop 2006)

Perceptron learning rule

- If x is Boolean, only weights of active features are updated
- Why is this important?

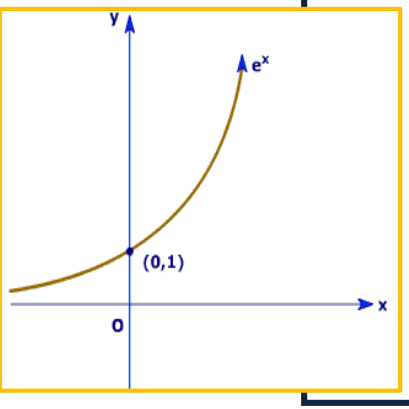
1. Initialize $w=0 \in \mathbf{R}^n$
2. Cycle through all examples

   a. Predict the label of instance x to be y' = sgn{$\mathbf{w}^T \bullet x$)

   b. If y'≠y, update the weight vector to

   w = w + r y x (r - a constant, learning rate)

   Otherwise, if y'=y, leave weights unchanged.



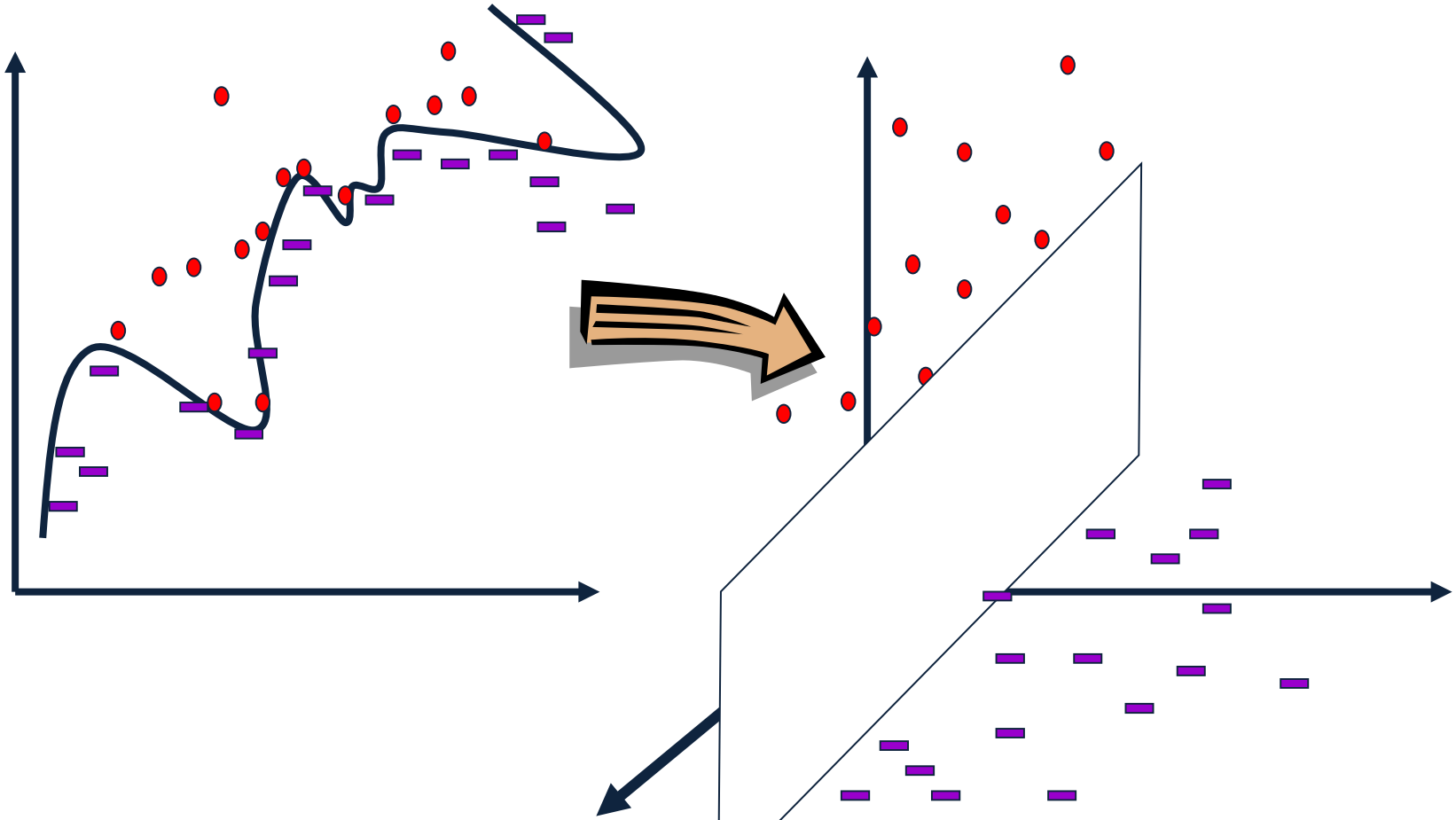
- $w^T x > 0$ is equivalent to: $\frac{1}{1+e^{-w^T x}} > \frac{1}{2}$

Perceptron Learnability

- Obviously can't learn what it can't represent (???)
  - Only linearly separable functions
- Minsky and Papert (1969) wrote an influential book demonstrating Perceptron's representational limitations
  - Parity functions can't be learned (XOR)
  - In vision, if patterns are represented with local features, can't represent symmetry, connectivity
- Research on Neural Networks stopped for years

- Perceptron
  - Rosenblatt himself (1959) asked,

    - *"What pattern recognition problems can be transformed so as to become linearly separable?"*



$(X_1 \wedge X_2) \vee (X_3 \wedge X_4)$

$y_1 \wedge y_2$

Perceptron Convergence

- **Perceptron Convergence Theorem:**
- If there exist a set of weights that are consistent with the data (i.e., the data is linearly separable), the perceptron learning algorithm will converge
  - How long would it take to converge ?
- **Perceptron Cycling Theorem:**
- If the training data is not linearly separable the perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop.
  - How to provide robustness, more expressivity ?

Administration

- Hw1 was due yesterday
- Hw2 will be out tonight
- My office hour today: 1:30 (after class)
- Projects

Questions

Perceptron

Input set of examples and their labels $\mathcal{Z} = ((x_1, y_1), \ldots, (x_m, y_m)) \in (\mathbb{R}^n \times \{-1, 1\})^m$, η, θ_{Init}

- Initialize $\mathbf{w} \leftarrow \mathbf{0}$ and $\theta \leftarrow \theta_{Init}$
- for every training epoch:
- for every $x_j \in \mathcal{X}$:
  - $\hat{y} \leftarrow sign(\langle \mathbf{w}, x_j \rangle + \theta)$
  - if $(\hat{y} \neq y_j)$
    - $\mathbf{w} \leftarrow \mathbf{w} + \eta y_j x_j$
    - $\theta \leftarrow \theta + \eta y_j$

Just to make sure we understand that we learn both w and μ

Perceptron: Mistake Bound Theorem

- Maintains a weight vector $w \in \mathcal{R}^N$, $w_0=(0,\ldots,0)$.
- Upon receiving an example $x \in \mathcal{R}^N$
- Predicts according to the linear threshold function $w^T \bullet x \geq 0$.

- **Theorem [Novikoff,1963]** *Let* $(x_1; y_1),\ldots,: (x_t; y_t)$, *be a sequence of labeled examples with* $x_i \in R^N$, $||x_i|| \leq R$ *and* $y_i \in \{-1,1\}$ *for all i. Let* $u \in R^N$, $\gamma > 0$ *be such that,* $||u|| = 1$ *and* $y_i u^T \bullet x_i \geq \gamma$ *for all i.*

  Complexity Parameter

  Then Perceptron makes at most R^2 / γ^2 *mistakes on this example sequence.*

(see additional notes)

Perceptron-Mistake Bound

Proof: Let v_k be the hypothesis before the k-th mistake. Assume that the k-th mistake occurs on the input example (x_i, y_i).

Assumptions

$v_1 = 0$

$\|u\| = 1$

$y_i\, u^T \bullet x_i \geq \gamma$

$$\therefore \; y_i(\vec{v_k} \cdot \vec{x_i}) \leq 0.$$

$$\begin{aligned} \vec{v_{k+1}} &= \vec{v_k} + y_i \vec{x_i} \\ \vec{v_{k+1}} \cdot \vec{u} &= \vec{v_k} \cdot \vec{u} + y_i(\vec{u} \cdot \vec{x_i}) \\ &\geq \vec{v_k} \cdot \vec{u} + \gamma \\ \therefore \; \vec{v_{k+1}} \cdot \vec{u} &\geq k\gamma \end{aligned}$$

1. Note that the bound does not depend on the dimensionality nor on the number of examples.
2. Note that we place weight vectors and examples in the same space.
3. Interpretation of the theorem

$$\begin{aligned} \|\vec{v_{k+1}}\|^2 &= \|\vec{v_k}\|^2 + 2y_i(\vec{v_k} \cdot \vec{x_i}) + \|\vec{x_i}\|^2 \\ &\leq \|\vec{v_k}\|^2 + R^2 \\ \therefore \; \|\vec{v_{k+1}}\|^2 &\leq kR^2 \end{aligned}$$

Therefore,

$$\sqrt{k}R \geq \|\vec{v_{k+1}}\| \geq \vec{v_{k+1}} \cdot \vec{u} \geq k\gamma.$$

$k < R^2 / \gamma^2$

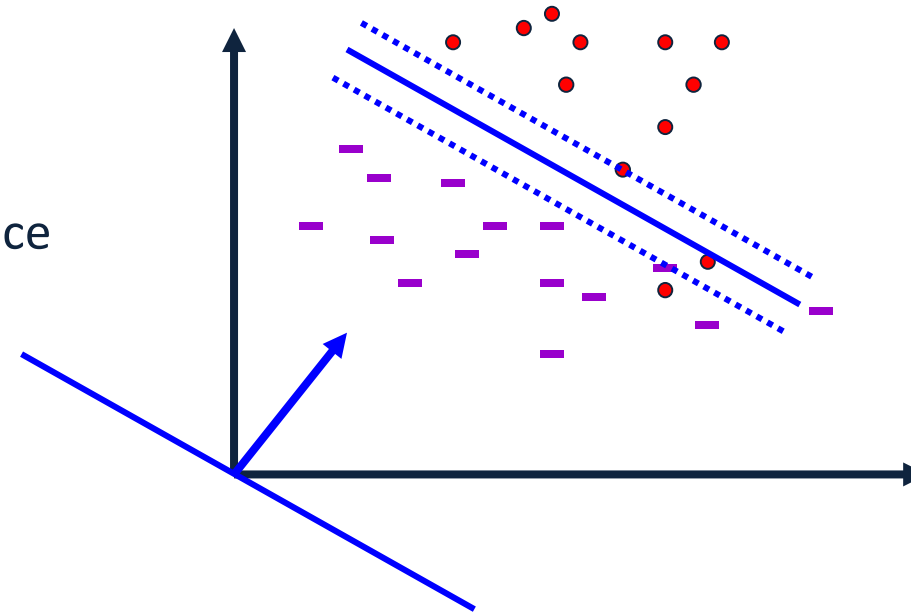
The second inequality follows because $\|\vec{u}\| \leq 1$.

Robustness to Noise

- In the case of non-separable data , the extent to which a data point fails to have margin γ via the hyperplane w can be quantified by a slack variable

$$\xi_i = \max(0, \gamma - y_i w^T x_i).$$

- Observe that when $\xi_i = 0$, the example x_i has margin at least γ. Otherwise, it grows linearly with $- y_i w^T x_i$
- Denote: $D_2 = [\sum \{\xi_i^2\}]^{1/2}$
- Theorem: The perceptron is guaranteed to make no more than $((R+D_2)/\gamma)^2$ mistakes on any sequence of examples satisfying $||x_i||^2 < R$



- Perceptron is expected to have some robustness to noise.

Perceptron for Boolean Functions

- How many mistakes will the Perceptron algorithms make when learning a k-disjunction?
- Try to figure out the bound
- Find a sequence of examples that will cause Perceptron to make O(n) mistakes on k-disjunction on n attributes.
- (Where is n coming from?)
- Recall that halving suggested the possibility of a better bound – klog(n).

- This can be achieved by Winnow
  - A multiplicative update algorithm [Littlestone'88]
  - See HW2

Practical Issues and Extensions

- There are many extensions that can be made to these basic algorithms.
- Some are necessary for them to perform well
  - Regularization (next; will be motivated in the next section, COLT)
- Some are for ease of use and tuning
  - Converting the output of a Perceptron/Winnow to a conditional probability

$$P(y = +1|x) = \frac{1}{1 + e^{-Aw^T x}}$$

  - The parameter A can be tuned on a development set
- Multiclass classification (later)
- Key efficiency issue: Infinite attribute domain
  - Sparse representation on the input

Regularization Via Averaged Perceptron

- An Averaged Perceptron Algorithm is motivated by the following considerations:
  - In real life, we want more guarantees from our learning algorithm
  - In the mistake bound model:
    - We don't know when we will make the mistakes.
    - In a sequential/on-line scenario, which hypothesis will you choose…??
    - Being consistent with more examples is better
  - Ideally, we want the expected performance to be on the number of examples seen and not number of mistakes. (The PAC model does that)
  - Every Mistake-Bound Algorithm can be converted efficiently to a PAC algorithm – to yield global guarantees on performance.
- To convert a given Mistake Bound algorithm (into a global guarantee algorithm):
  - Wait for a long stretch w/o mistakes (there must be one)
  - Use the hypothesis at the end of this stretch.
  - Its PAC behavior is relative to the length of the stretch.
- Averaged Perceptron returns a weighted average of a number of earlier hypotheses; the weights are a function of the length of no-mistakes stretch.

Regularization Via Averaged Perceptron

- **Training:**

[m: #(examples); k: #(mistakes) = #(hypotheses); c_i: consistency count for hypothesis v_i]

- Input: a labeled training set $\{(x_1, y_1),\ldots(x_m, y_m)\}$
- Number of epochs T
- Output: a list of weighted perceptrons $\{(v_1, c_1),\ldots,(v_k, c_k)\}$
- Initialize: k=0; $v_1 = 0$, $c_1 = 0$
- Repeat T times:
  - For i =1,…m:
  - Compute prediction y' = sgn(v_k^T x_i)
  - If y' = y, then $c_k = c_k + 1$

    else: $v_{k+1} = v_k + y_i x$; $c_{k+1} = 1$; k = k+1
- **Prediction:**
- Given: a list of weighted perceptrons $\{(v_1, c_1),\ldots(v_k, c_k)\}$; a new example x

  Predict the label(x) as follows:

  $y(x) = \text{sgn}\ [\ \sum_{1,k} c_i\ (v_i^T\ x)\ ]$

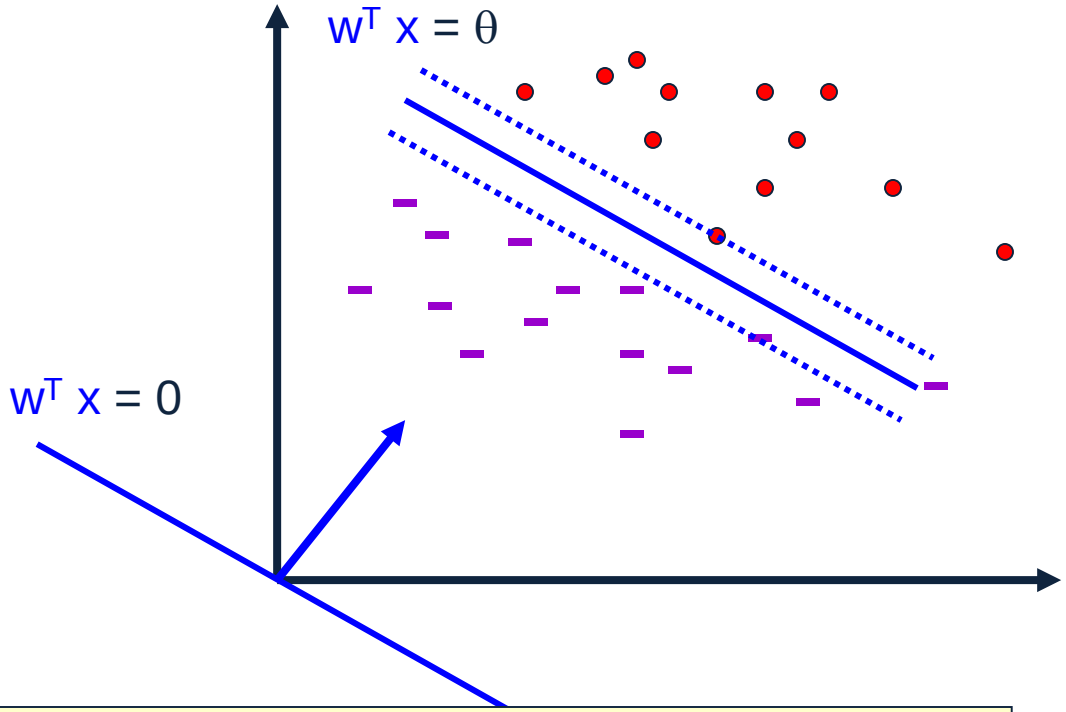
- This can be done on top of any online mistake driven algorithm.
- In HW 2 you will run it over three different algorithms.
- The implementation requires thinking.

Averaged version of Perceptron /Winnow is as good as any other linear learning algorithm, if not better.

Perceptron with Margin

- Thick Separator (aka as Perceptron with Margin)
  (Applies both for Perceptron and Winnow)

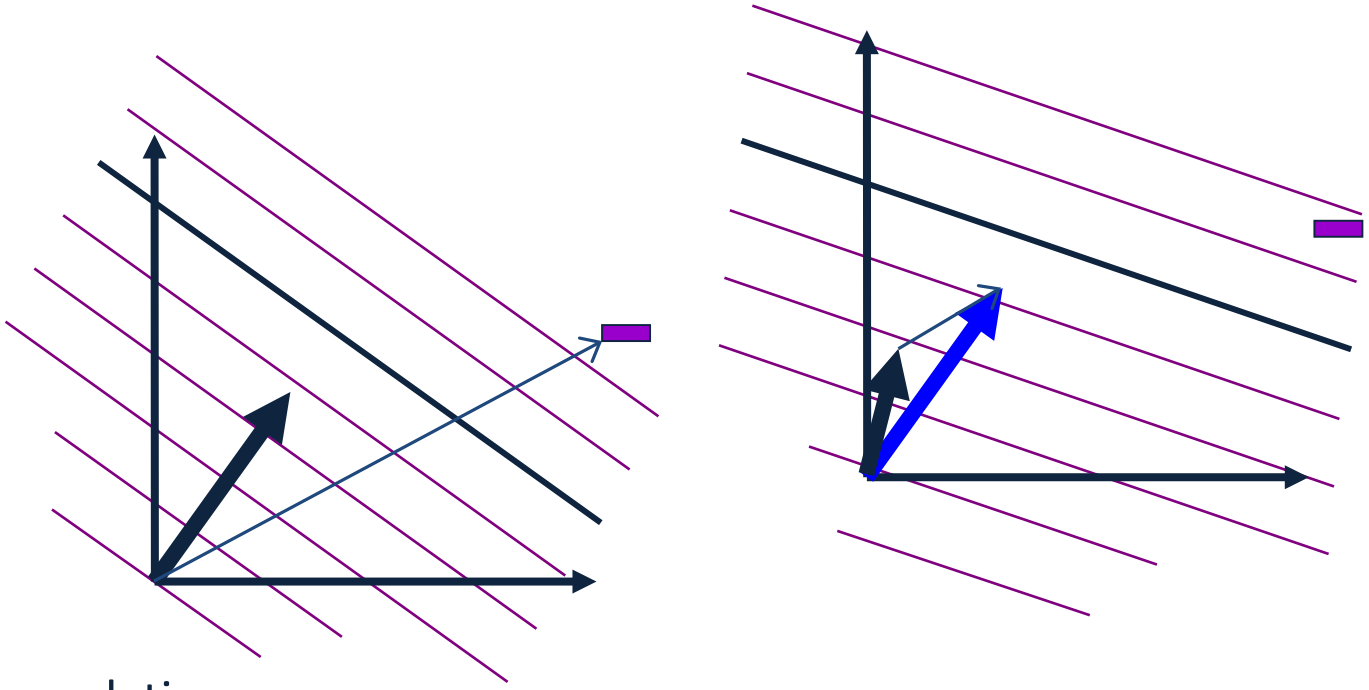
- Promote if:
  - $w^T x - \theta < \gamma$
- Demote if:
  - $w^T x - \theta > \gamma$



Note: γ is a functional margin. Its effect could disappear as w grows.
Nevertheless, this has been shown to be a very effective algorithmic addition.
(Grove & Roth 98,01; Karov et. al 97)

Other Extensions

- Assume you made a mistake on example x.
- You then see example x again; will you make a mistake on it?
- Threshold relative updating (Aggressive Perceptron)
- w ← w + rx
- $r = \frac{\theta - w^T x}{||x||^2}$



- Equivalent to updating on the same example multiple times

LBJava

- Several of these extensions (and a couple more) are implemented in the LBJava learning architecture that supports several linear update rules (Winnow, Perceptron, naïve Bayes)
- Supports
  - Regularization(averaged Winnow/Perceptron; Thick Separator)
  - Conversion to probabilities
  - Automatic parameter tuning
  - True multi-class classification
  - Feature Extraction and Pruning
  - Variable size examples
  - Good support for large scale domains in terms of number of examples and number of features.
  - Very efficient
  - Many other options
- [Download from: http://cogcomp.org/page/software/]

General Stochastic Gradient Algorithms

The loss Q: a function of x, w and y

- Given examples $\{z=(x,y)\}_{1,m}$ from a distribution over $X \times Y$, we are trying to learn a linear function, parameterized by a weight vector w, so that we minimize the expected risk function

$$J(w) = E_z\, Q(z,w) \sim= 1/m \sum_{1,m} Q(z_i, w_i)$$

- In Stochastic Gradient Descent Algorithms we approximate this minimization by incrementally updating the weight vector w as follows:

$$w_{t+1} = w_t - r_t\, g_w\, Q(z_t, w_t) = w_t - r_t\, g_t$$

- Where $g_t = g_w\, Q(z_t, w_t)$ is the gradient with respect to w at time t.

- The difference between algorithms now amounts to choosing a different loss function Q(z, w)

General Stochastic Gradient Algorithms

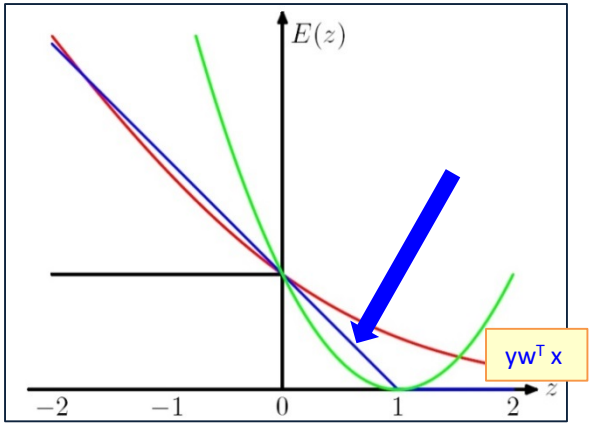
Learning rate | gradient | **The loss Q:** a function of x, w and y

$w_{t+1} = w_t - r_t\, g_w\, Q(x_t, y_t, w_t) = w_t - r_t\, g_t$

- LMS: $Q((x, y), w) = 1/2\ (y - w^T x)^2$
- Computing the gradient leads to the update rule (Also called Widrow's Adaline): $w_{t+1} = w_t + r\ (y_t - w_t^T\ x_t)\ x_t$
- Here, even though we make binary predictions based on $sgn\ (w^T x)$ we do not take the sign of the dot-product into account in the loss.

- Another common loss function is:
- Hinge loss:

  $Q((x, y), w) = \max(0, 1 - y\ w^T x)$
- Computing the gradient leads to the perceptron update rule:



- If $y_i\ w_i^T \cdot x_i > 1$ (No mistake, by a margin): No update
- Otherwise (Mistake, relative to margin): $w_{t+1} = w_t + r\ y_t\ x_t$

Here g = -yx
Good to think about the case of Boolean examples

New Stochastic Gradient Algorithms

$$w_{t+1} = w_t - r_t\, g_w\, Q(z_t, w_t) = w_t - r_t\, g_t$$

(notice that this is a vector, each coordinate (feature) has its own $w_{t,j}$ and $g_{t,j}$)

- So far, we used fixed learning rates $r = r_t$, but this can change.
- AdaGrad alters the update to adapt based on historical information
  - Frequently occurring features in the gradients get small learning rates and infrequent features get higher ones.
  - The idea is to "learn slowly" from frequent features but "pay attention" to rare but informative features.
- Define a "per feature" learning rate for the feature j, as:

$$r_{t,j} = r/(G_{t,j})^{1/2}$$

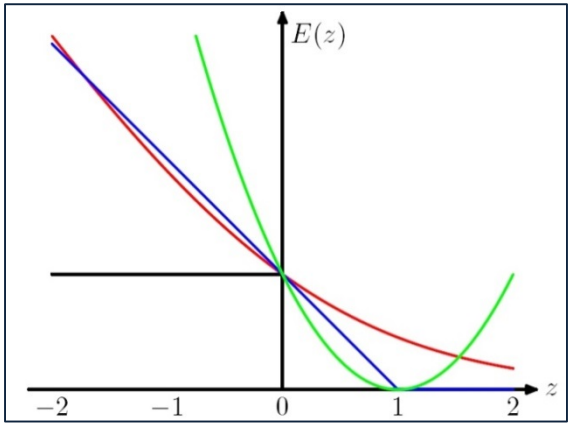
- where $G_{t,j} = \sum_{k=1,\, t} g^2_{k,j}$ the sum of squares of gradients at feature j until time t.
- Overall, the update rule for Adagrad is:

$$w_{t+1,j} = w_{t,j} - g_{t,j}\, r/(G_{t,j})^{1/2}$$

Easy to think about the case of Perceptron, and on Boolean examples.

- This algorithm is supposed to update weights faster than Perceptron or LMS when needed.

Regularization



- The more general formalism adds a regularization term to the risk function, and minimize:

$$J(w) = 1/m \sum_{1,\,m} Q(z_i, w_i) + \lambda R_i(w_i)$$

- Where R is used to enforce "simplicity" of the learned functions.

- LMS case: $Q((x, y), w) = (y - w^T x)^2$
  - $R(w) = ||w||^2_2$ gives the optimization problem called **Ridge Regression**.
  - $R(w) = ||w||_1$ gives a problem called the **LASSO problem**

- Hinge Loss case: $Q((x, y), w) = \max(0, 1 - y\, w^T x)$
  - $R(w) = ||w||^2_2$ gives the problem called **Support Vector Machines**

- Logistics Loss case: $Q((x,y),w) = \log(1+\exp\{-y\, w^T x\})$
  - $R(w) = ||w||^2_2$ gives the problem called **Logistics Regression**

- These are convex optimization problems and, in principle, the same gradient descent mechanism can be used in all cases.
- We will see later why it makes sense to use the "size" of w as a way to control "simplicity".

Algorithmic Approaches

- Focus: Two families of algorithms (one of the on-line representative)
  - Additive update algorithms: Perceptron
    - SVM is a close relative of Perceptron
  - Multiplicative update algorithms: Winnow
    - Close relatives: Boosting, Max entropy/Logistic Regression

Summary of Algorithms

- Examples: x 2 {0,1}^{n}; or x 2 R^n (indexed by k) ; Hypothesis: w 2 R^n
- Prediction: y 2 {-1,+1}: Predict: y = 1 iff w¢x >μ
- Update: Mistake Driven
- Additive weight update algorithm: w ← w +r y_k x_k
  - (Perceptron, Rosenblatt, 1958. Variations exist)
  - In the case of Boolean features:

$$\text{If Class} = 1 \text{ but } \mathbf{w} \bullet \mathbf{x} \leq \theta \text{ , } w_i \leftarrow w_i + 1 \text{ (if } x_i = 1) \text{ (promotion)}$$
$$\text{If Class} = 0 \text{ but } \mathbf{w} \bullet \mathbf{x} \geq \theta \text{ , } w_i \leftarrow w_i - 1 \text{ (if } x_i = 1) \text{ (demotion)}$$

- Multiplicative weight update algorithm $w_i \leftarrow w_i \exp\{y_k x_i\}$
- (Winnow, Littlestone, 1988. Variations exist)
  - Boolean features:

$$\text{If Class} = 1 \text{ but } \mathbf{w} \bullet \mathbf{x} \leq \theta \text{ , } w_i \leftarrow 2w_i \text{ (if } x_i = 1) \text{ (promotion)}$$
$$\text{If Class} = 0 \text{ but } \mathbf{w} \bullet \mathbf{x} \geq \theta \text{ , } w_i \leftarrow w_i/2 \text{ (if } x_i = 1) \text{ (demotion)}$$

Which algorithm is better?
How to Compare?

- Generalization
  - Since we deal with linear learning algorithms, we know (???) that they will all converge eventually to a perfect representation.
    - All can represent the data
- So, how do we compare:
  1. How many examples are needed to get to a given level of accuracy?
  2. Efficiency: How long does it take to learn a hypothesis and evaluate it (per-example)?
  3. Robustness (to noise);
  4. Adaptation to a new domain, ….
- With (1) being the most fundamental question:
  - Compare as a function of what?
    - One key issue is the characteristics of the data

Sentence Representation

S= I don't know whether to laugh or cry

- Define a set of features:
  - features are relations that hold in the sentence
- Map a sentence to its feature-based representation
  - The feature-based representation will give some of the information in the sentence
- Use this feature-based representation as an example to your algorithm

Sentence Representation

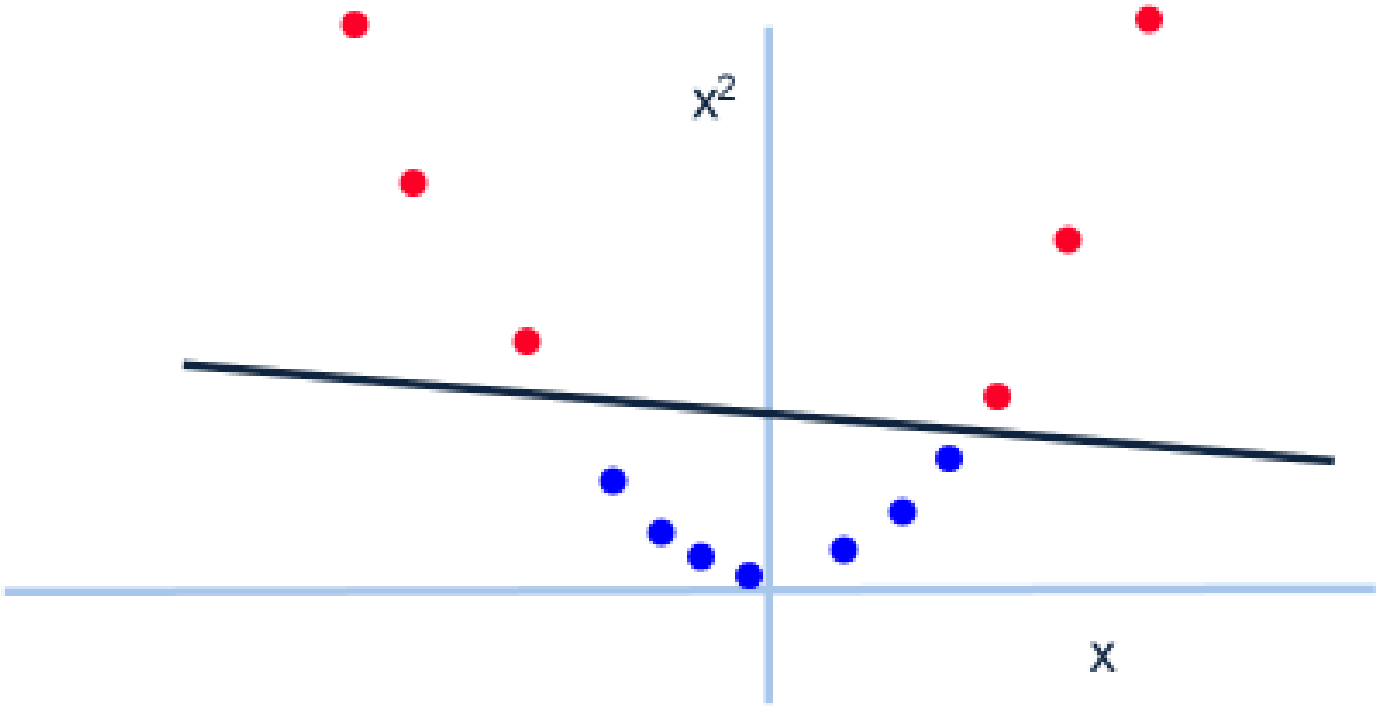
S= I don't know whether to laugh or cry

- Define a set of features:
  - features are properties that hold in the sentence
- Conceptually, there are two steps in coming up with a feature-based representation
  - What are the information sources available?
    - Sensors: words, order of words, properties (?) of words
  - What features to construct based on these?

Why is this distinction needed?

Blown Up Feature Space

- Data are separable in <x, x^2> space



er

Domain Characteristics

- The number of potential features is very large
- The instance space is sparse
- Decisions depend on a small set of features: the function space is sparse
- Want to learn from a number of examples that is small relative to the dimensionality

Generalization

- Dominated by the sparseness of the function space
  - Most features are irrelevant

- # of examples required by multiplicative algorithms depends mostly on # of relevant features
  - (Generalization bounds depend on the target ||u||)

- # of examples required by additive algorithms depends heavily on sparseness of features space:
  - Advantage to additive. Generalization depend on input ||x||
    - (Kivinen/Warmuth 95).
- Nevertheless, today most people use additive algorithms.

Which Algorithm to Choose?

- Generalization (in terms of # of mistakes made)

The l_1 norm: $\|x\|_1 = \sum_i \|x_i\|$	The l_2 norm: $\|x\|_2 = (\sum_1^n \|x_i\|^2)^{1/2}$
The l_p norm: $\|x\|_p = (\sum_1^n \|x_i\|^P)^{1/p}$	The l_1 norm: $\|x\|_1 = \max_i \|x_i\|$

  - Multiplicative algorithms:
    - Bounds depend on $\|\mathbf{u}\|$, the separating hyperplane; i: example #)
    - $M_w = 2 \ln n \, \|u\|_1^2 \, \max_i \|x^{(i)}\|_1^2 / \min_i (u \, x^{(i)})^2$
    - Do not care much about data; advantage with sparse target u
  - Additive algorithms:
    - Bounds depend on $\|\mathbf{x}\|$ (Kivinen / Warmuth, '95)
    - $M_p = \|u\|_2^2 \, \max_i \|x^{(i)}\|_2^2 / \min_i (u \, x^{(i)})^2$
    - Advantage with few active features per example

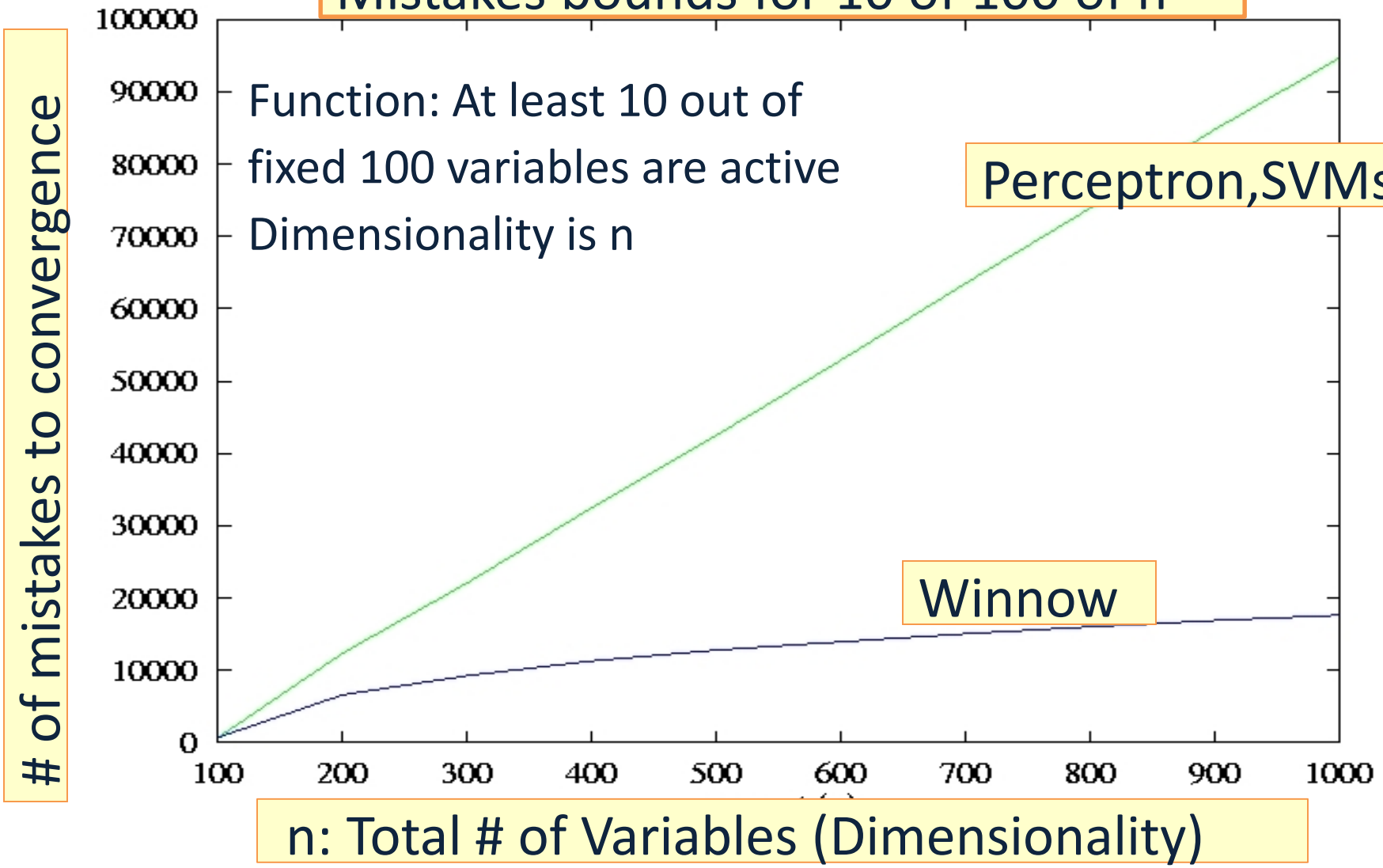
Examples

$M_w = 2\ln n \, \|u\|_1^2 \, \max_i \|x^{(i)}\|_1^2 / \min_i (u \, x^{(i)})^2$
$M_p = \|u\|_2^2 \, \max_i \|x^{(i)}\|_2^2 / \min_i (u \, x^{(i)})^2$

- Extreme Scenario 1: Assume the u has exactly k active features, and the other n-k are 0. That is, only k input features are relevant to the prediction. Then:
  - $\|\mathbf{u}\|_2 = k^{1/2}$; $\|\mathbf{u}\|_1 = k$; $\max \|x\|_2 = n^{1/2}$; $\max \|x\|_1 = 1$
  - We get that: $M_p = kn$; $M_w = 2k^2 \ln n$
  - Therefore, if k<<n, Winnow behaves much better.
- Extreme Scenario 2: Now assume that u=(1, 1,….1) and the instances are very sparse, the rows of an nxn unit matrix. Then:
  - $\|\mathbf{u}\|_2 = n^{1/2}$; $\|\mathbf{u}\|_1 = n$; $\max \|x\|_2 = 1$; $\max \|x\|_1 = 1$
  - We get that: $M_p = n$; $M_w = 2n^2 \ln n$
  - Therefore, Perceptron has a better bound.

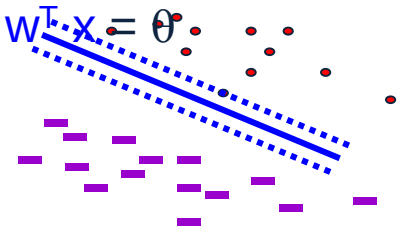
HW2

Mistakes bounds for 10 of 100 of n



- Midterm exam: Moved to October 31st. In class.
- Projects!

Summary



- Introduced multiple versions of on-line algorithms
- Most turned out to be Stochastic Gradient Algorithms
  - For different loss functions
- Some turned out to be mistake driven
- We suggested generic improvements via:
  - Regularization via adding a term that forces a "simple hypothesis"

A term that minimizes error on the training data

A term that forces simple hypothesis

$$J(w) = \sum_{1,m} Q(z_i, w_i) + \lambda R_i(w_i)$$

  - Regularization via the Averaged Trick
    - "Stability" of a hypothesis is related to its ability to generalize
  - An improved, adaptive, learning rate (Adagrad)
- Dependence on function space and the instance space properties.
- Today:
  - A way to deal with non-linear target functions (Kernels)
  - Beginning of Learning Theory.

Efficiency

- Dominated by the size of the feature space
- Most features are functions (e.g. conjunctions) of raw attributes

$$X(x_1, x_2, x_3, \ldots x_k) \rightarrow X(\chi_1(\mathbf{x}), \chi_2(\mathbf{x}), \chi_3(\mathbf{x}) \ldots \chi_n(\mathbf{x})) \qquad \mathbf{n >> k}$$

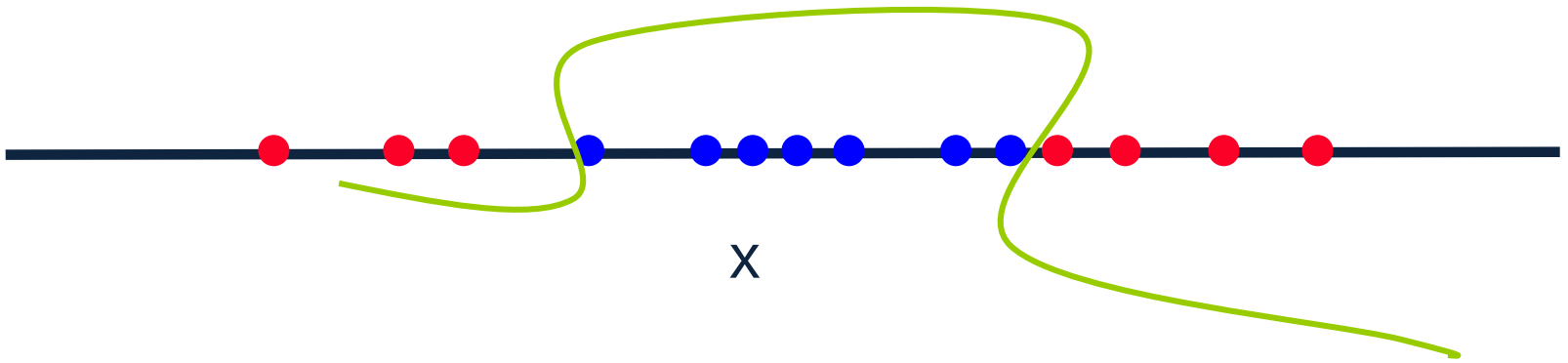
- Additive algorithms allow the use of Kernels
  - No need to explicitly generate complex features

$$f(x) = \sum_i c_i \, K(x, x_i)$$

- Could be more efficient since work is done in the original feature space, but expressivity is a function of the kernel expressivity.

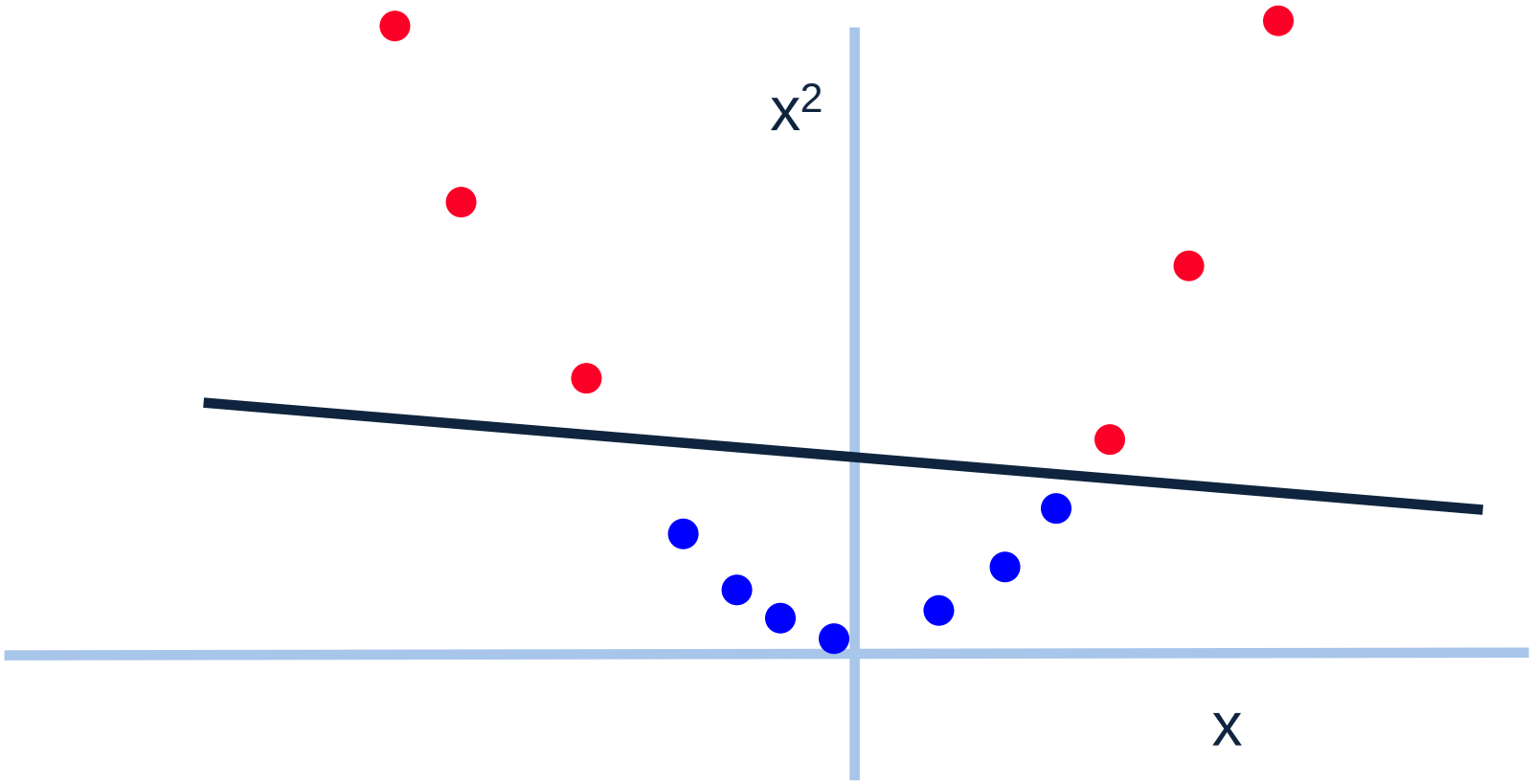
Functions Can be Made Linear

- Data are not linearly separable in one dimension
- Not separable if you insist on using a specific class of functions

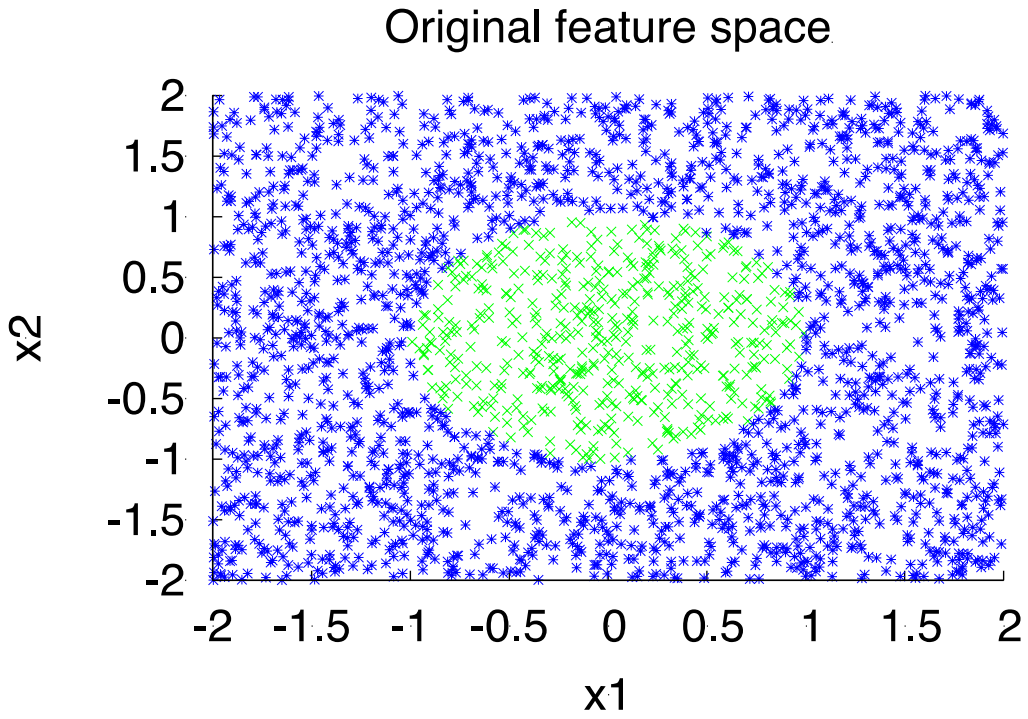


Blown Up Feature Space

- Data are separable in <x, x^2> space



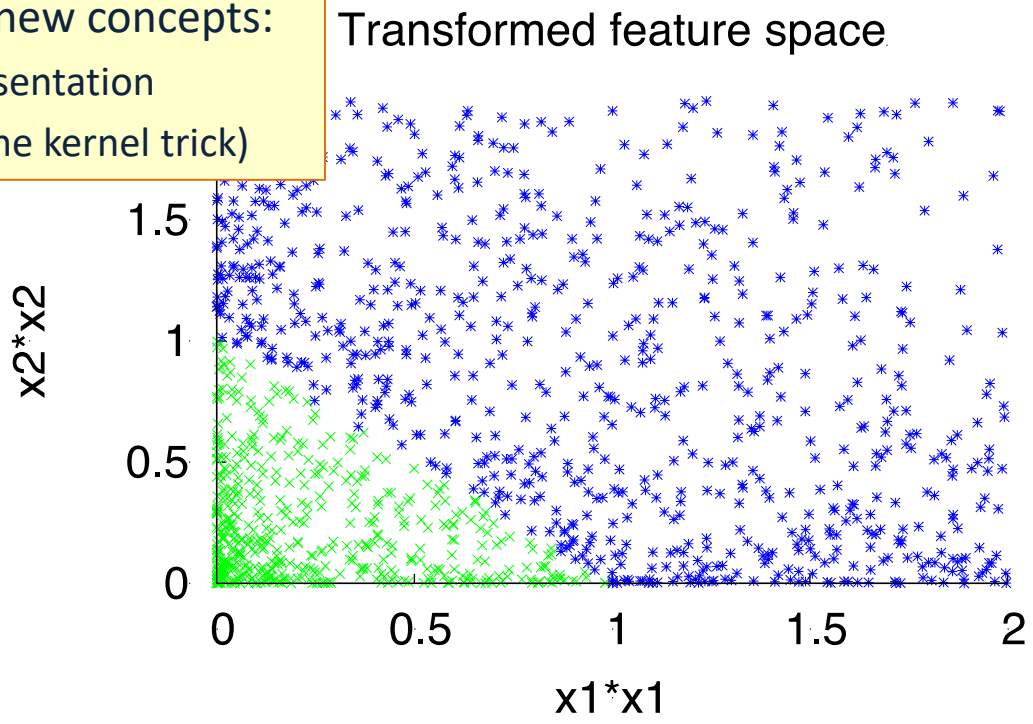
Making data linearly separable



$$f(\mathbf{x}) = 1 \text{ iff } x_1^2 + x_2^2 \leq 1$$

Making data linearly separable

- In order to deal with this, we introduce two new concepts:
  - Dual Representation
  - Kernel (& the kernel trick)



Transform data: $\mathbf{x} = (x_1, x_2)$ => $\mathbf{x'} = (x_1^2, x_2^2)$

$f(\mathbf{x'}) = 1$ iff $x'_1 + x'_2 \leq 1$

Dual Representation

Examples $x \in \{0,1\}^N$; Learned hypothesis $w \in \mathcal{R}^N$

$$f(x) = \text{sgn}\{w^T \cdot x\} = \text{sgn}\{\sum_{i=1}^{n} w_i x_i\}$$

Perceptron Update:

If y'≠y, update: **w = w + ry x**

- Let w be an initial weight vector for perceptron. Let $(x^1,+)$, $(x^2,+)$, $(x^3,-)$, $(x^4,-)$ be examples and assume mistakes are made on x^1, x^2 and x^4.
- What is the resulting weight vector?

$$w = w + x^1 + x^2 - x^4$$

- (here r=1)
- In general, the weight vector w can be written as a linear combination of examples:

$$w = \sum_{1,m} r\, \alpha_i\, y_i\, x^i$$

- Where α_i is the number of mistakes made on x^i.

Note: We care about the dot product: $f(x) = w^T x =$

$$= (\sum_{1,m} r\, \alpha_i\, y_i\, x^i)^T x$$
$$= \sum_{1,m} r\, \alpha_i\, y_i\, (x^{iT} x)$$

Kernel Based Methods

$$f(x) = \text{sgn}\{w^T \cdot x\} = \text{sgn}\{\sum_{i=1}^{n} w_i x_i\}$$

- A method to run Perceptron on a very large feature set, without incurring the cost of keeping a very large weight vector.
- Computing the dot product can be done in the original feature space.
- Notice: this pertains only to efficiency: The classifier is identical to the one you get by blowing up the feature space.
- Generalization is still relative to the real dimensionality (or, related properties).
- Kernels were popularized by SVMs, but many other algorithms can make use of them (== run in the dual).
  - Linear Kernels: no kernels; stay in the original space. A lot of applications actually use linear kernels.

Kernel Base Methods

Examples $x \in \{0,1\}^N$; Learned hypothesis $w \in \mathcal{R}^N$

$$f(x) = \text{sgn}\{w^T \cdot x\} = \text{sgn}\{\sum_{i=1}^{n} w_i x_i\,(x)\}$$

- Let I be the set t_1, t_2, t_3 …of monomials (conjunctions) over the feature space $x_1, x_2 \ldots x_n$.
- Then we can write a linear function over this new feature space.

$$f(x) = \text{sgn}\{w^T \cdot x\} = \text{sgn}\{\sum_{I} w_i t_i\,(x)\}$$

Example : $x_1x_2x_4(11010) = 1 \quad x_3x_4(11010) = 0 \quad x_1x_2x_4(11011) = 1$

Kernel Based Methods

Examples $x \in \{0,1\}^N$; Learned hypothesis $w \in \mathcal{R}^N$

$$f(x) = \text{sgn}\{w^T \cdot x\} = \text{sgn}\{\sum_I w_i t_i(x)\}$$

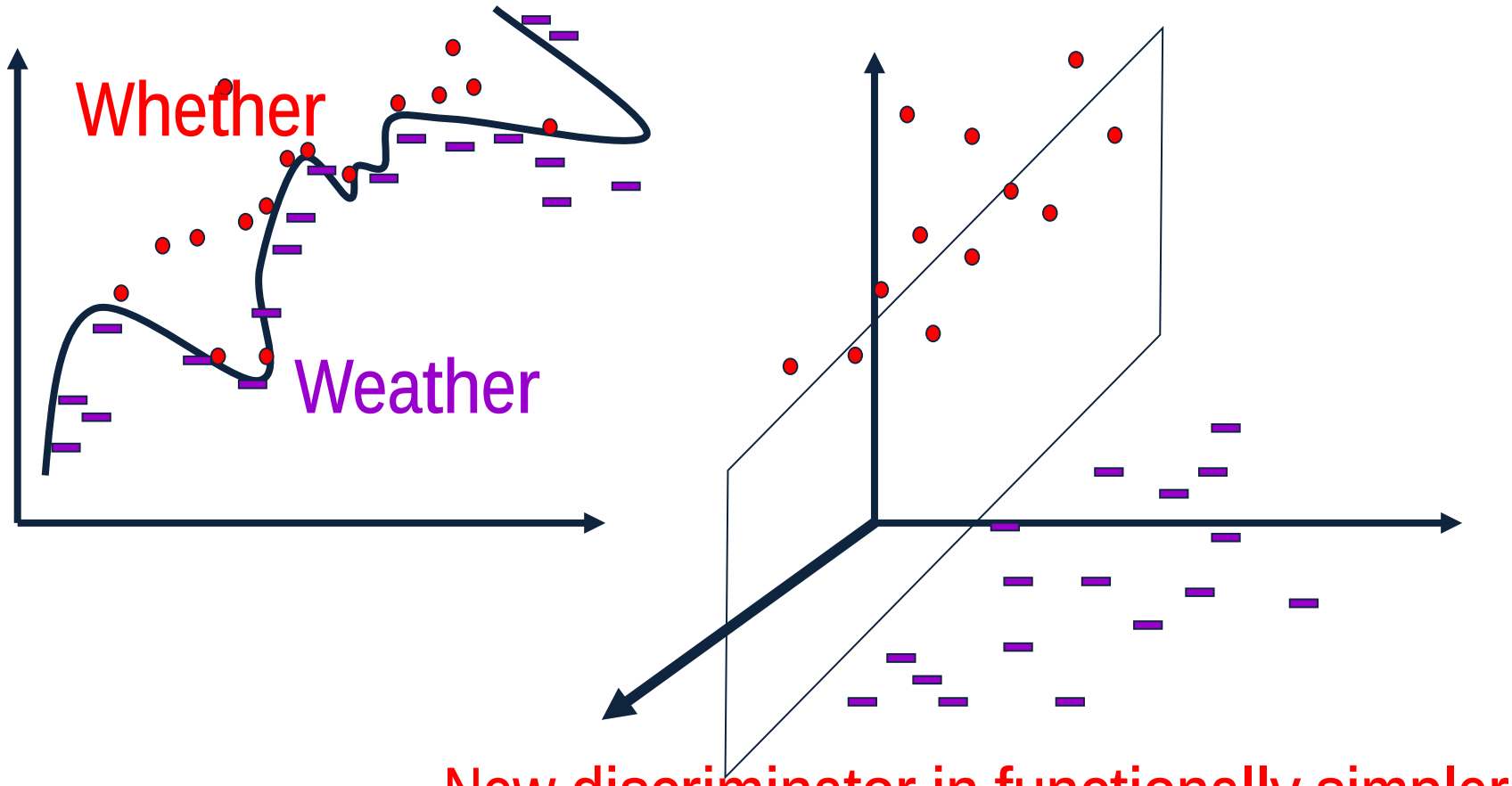
Perceptron Update:

If y'≠y, update: **w = w + ry x**

- Great Increase in expressivity
- Can run Perceptron (and Winnow) but the convergence bound may suffer exponential growth.

- Exponential number of monomials are true in each example.
- Also, will have to keep many weights.

Embedding



New discriminator in functionally simpler

$x_1 x_2 \bar{x}_3 \vee \bar{x}_1 x_4 \bar{x}_3 \vee x_3 \bar{x}_2 x_5$ $\qquad$ $y_1 \vee y_4 \vee y_5$

The Kernel Trick(1)

Examples x $\in \{0,1\}^N$; Learned hypothesis w $\in \mathcal{R}^N$

$$f(x) = \text{sgn}\{w^T \cdot x\} = \text{sgn}\{\sum_I w_i t_i(x)\}$$

Perceptron Update:

If y'≠y, update: **w = w + ry x**

- Consider the value of w used in the prediction.
- Each previous mistake, on example z, makes an additive contribution of +/-1 to some of the coordinates of w.
  - Note: examples are Boolean, so only coordinates of w that correspond to ON terms in the example z ($t_i(z) = 1$) are being updated.
- The value of w is determined by the number and type of mistakes.

The Kernel Trick(2)

Examples $x \in \{0,1\}^N$; Learned hypothesis $w \in \mathcal{R}^N$

$$f(x) = \text{sgn}\{w^T \cdot x\} = \text{sgn}\{\sum_I w_i t_i(x)\}$$

Perceptron Update:

If y'≠y, update: **w = w + ry x**

- P – set of examples on which we Promoted
- D – set of examples on which we Demoted
- M = P [D

$$f(x) = \text{sgn}\sum_I w_i t_i(x) = \sum_I [\sum_{z \in P, t_i(z)=1} 1 - \sum_{z \in D, t_i(z)=1} 1] t_i(x) =$$

$$= \sum_I [\sum_{z \in M} S(z) t_i(z) t_i(x)]$$

The Kernel Trick(3)

$$f(x) = \text{sgn}\{w^T \cdot x\} = \text{sgn}\{\sum_I w_i t_i(x)\}$$

- P – set of examples on which we Promoted
- D – set of examples on which we Demoted
- M = P [D

$$f(x) = \text{sgn}\sum_I w_i t_i(x) = \sum_I [\sum_{z \in P, t_i(z)=1} 1 - \sum_{z \in D, t_i(z)=1} 1] ti(x)$$

$$= \text{sgn}\{\sum_I [\sum_{z \in M} S(z) t_i(z) t_{i(x)}]\}$$

- Where S(z)=1 if z $\in$ P and S(z) = -1 if z $\in$ D.
- Reordering:

$$f(x) = \text{sgn}\{\sum_{z \in M} S(z) \sum_I t_i(z) t_i(x)\}$$

The Kernel Trick(4)

$$\mathbf{f(x) = Th_\theta(\sum_{i\in I} w_i t_i(x))}$$

- S(y)=1 if y $\in$P and S(y) = -1 if y $\in$D.

$$f(x) = \mathbf{Th_\theta(\sum_{z\in M} S(z)\sum_{i\in I} t_i(z)t_i(x))}$$

- A mistake on z contributes the value +/-1 to all monomials satisfied by z. The total contribution of z to the sum is equal to the number of monomials that satisfy both x and z.
- Define a dot product in the t-space:

$$\mathbf{K(x,z) = \sum_{i\in I} t_i(z)t_i(x)}$$

- We get the standard notation:

$$f(x) = \mathbf{Th_\theta(\sum_{z\in M} S(z)K(x,z))}$$

Kernel Based Methods

$$f(x) = \mathbf{Th}_{\theta}\left(\sum_{\mathbf{z}\in M} \mathbf{S(z)K(x,z)}\right)$$

- What does this representation give us?

$$\mathbf{K(x,z)} = \sum_{\mathbf{i}\in I} \mathbf{t_i(z)t_i(x)}$$

- We can view this Kernel as the distance between x,z in the t-space.

- So far, all we did is algebra

- But, K(x,z) can be measured in the original space, without explicitly writing the t-representation of x, z

x_1x_3 (001) = x_1x_3 (011) = 1
x_1 (001) = x_1 (011) = 1 ; x_3 (001) = x_3 (011) = 1
Φ (001) = Φ (011) = 1
If any other variables appears in the monomial, it's evaluation on x, z will be different.

rnel Trick

$$f(x) = Th_\theta\left(\sum_{z \in M} S(z)K(x,z)\right) \quad K(x,z) = \sum_{i \in I} t_i(z)t_i(x)$$

- sider the space of all 3^n monomials (allowing both tive and negative literals). Then,
- :
$$K(x,z) = \sum_{i \in I} t_i(z)t_i(x) = 2^{same(x,z)}$$
- W n same(x,z) is the number of features that have the sa e value for both x and z.
- We get:
$$f(x) = Th_\theta\left(\sum_{z \in M} S(z)(2^{same(x,z)})\right)$$
- Example: Take n=3; x=(001), z=(011), monomials of size 0,1,2,3
- **Proof:** let k=same(x,z); construct a "surviving" monomials by: (1) choosing to include one of these k literals with the right polarity in the monomial, or (2) choosing to not include it at all. Monomials with literals outside this set disappear.

Example

$$f(x) = Th_{\theta}\left(\sum_{z \in M} S(z)K(x,z)\right) \quad K(x,z) = \sum_{i \in I} t_i(z)t_i(x)$$

- Take X={x_1, x_2, x_3, x_4}
- I = The space of all 3^n monomials; | I |= 81
- Consider x=(1100), z=(1101)
- Write down I(x), I(z), the representation of x, z in the I space.
- Compute I(x) ·I(z).
- Show that
- K(x,z) =I(x) · I(z) = $\sum_I t_i(z)\, t_i(x) = 2^{same(x,z)} = 8$
- Try to develop another kernel, e.g., where I is the space of all conjunctions of size 3 (exactly).

Implementation: Dual Perceptron

$$f(x) = \mathbf{Th}_{\theta}\left(\sum_{\mathbf{z}\in M} \mathbf{S(z)K(x,z)}\right)$$

$$\mathbf{K(x,z)} = \sum_{\mathbf{i}\in I} \mathbf{t_i(z)t_i(x)}$$

- Simply run Perceptron in an on-line mode, but keep track of the set M.
- Keeping the set M allows us to keep track of S(z).
- Rather than remembering the weight vector w, remember the set M (P and D) – all those examples on which we made mistakes.

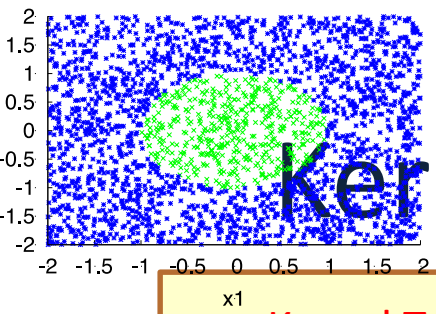
- Dual Representation

Example: Polynomial Kernel

- Prediction with respect to a separating hyper planes (produced by Perceptron, SVM) can be computed as a function of dot products of feature based representation of examples.
- We want to define a dot product in a high dimensional space.
- Given two examples $x = (x_1, x_2, \ldots x_n)$ and $y = (y_1, y_2, \ldots y_n)$ we want to map them to a high dimensional space [example- quadratic]:
  - $\Phi(x_1, x_2, \ldots, x_n) = (1, x_1, \ldots, x_n, x_1^2, \ldots, x_n^2, x_1x_2, \ldots, x_{n-1}x_n)$
  - $\Phi(y_1, y_2, \ldots, y_n) = (1, y_1, \ldots, y_n, y_1^2, \ldots, y_n^2, y_1y_2, \ldots, y_{n-1}y_n)$

  and compute the dot product $A = \Phi(x)^T\Phi(y)$ [takes time]
- Instead, in the original space, compute
  - $B = k(x, y) = [1 + (x_1, x_2, \ldots x_n)^T (y_1, y_2, \ldots y_n)]^2$
- Theorem: A = B (Coefficients do not really matter)

Sq(2)



Kernels – General Conditions

- Kernel Trick: You want to work with degree 2 polynomial features, **Φ**(x). Then, your dot product will be in a space of dimensionality n(n+1)/2. The kernel trick allows you to save and compute dot products in an n dimensional space.

- Can we use any K(.,.)? $f(x) = Th_\theta(\sum_{z \in M} S(z)K(x,z))$
  - A function K(x,z) is a valid kernel if it corresponds to a dot (an inner) product in some (perhaps infinite dimensional) feature space. $K(x,z) = \sum_{i \in I} t_i(z)t_i(x)$
- Take the quadratic kernel: $k(x,z) = (x^T z)^2$
- Example: Direct construction (2 dimensional, for simplicity):
- **$K(x,z) = (x_1 z_1 + x_2 z_2)^2 = x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2$**
- **$= (x_1^2, \text{sqrt}\{2\} x_1x_2, x_2^2)\ (z_1^2, \text{sqrt}\{2\} z_1z_2, z_2^2)^T$**
- **$= \Phi(x)^T \Phi(z)$ → A dot product in an expanded space.**
- It is not necessary to explicitly show the feature function **Φ**.
- General condition: construct the kernel matrix $\{k(x_i, z_j)\}$ (indices are over the examples); check that it's positive semi definite.

We proved that K is a valid kernel by explicitly showing that it corresponds to a dot product.

Polynomial kernels

- Linear kernel: $k(\mathbf{x}, \mathbf{z}) = \mathbf{xz}$
- Polynomial kernel of degree d: $k(\mathbf{x}, \mathbf{z}) = (\mathbf{xz})^d$
  (only dth-order interactions)
- Polynomial kernel up to degree d: $k(\mathbf{x}, \mathbf{z}) = (\mathbf{xz} + c)^d$ (c>0)
  (all interactions of order d or lower)

Constructing New Kernels

- You can construct new kernels $k'(\mathbf{x}, \mathbf{x'})$ from existing ones:
  - Multiplying $k(\mathbf{x}, \mathbf{x'})$ by a constant c:
    $k'(\mathbf{x}, \mathbf{x'}) = ck(\mathbf{x}, \mathbf{x'})$
  - Multiplying $k(\mathbf{x}, \mathbf{x'})$ by a function f applied to $\mathbf{x}$ and $\mathbf{x'}$:
    $k'(\mathbf{x}, \mathbf{x'}) = f(\mathbf{x})k(\mathbf{x}, \mathbf{x'})f(\mathbf{x'})$
  - Applying a polynomial (with non-negative coefficients) to $k(\mathbf{x}, \mathbf{x'})$:
    $k'(\mathbf{x}, \mathbf{x'}) = P(\ k(\mathbf{x}, \mathbf{x'})\ )$ with $P(z) = \sum_i a_i z^i$ and $a_i \geq 0$
  - Exponentiating $k(\mathbf{x}, \mathbf{x'})$:
    $k'(\mathbf{x}, \mathbf{x'}) = \exp(k(\mathbf{x}, \mathbf{x'}))$

Constructing New Kernels (2)

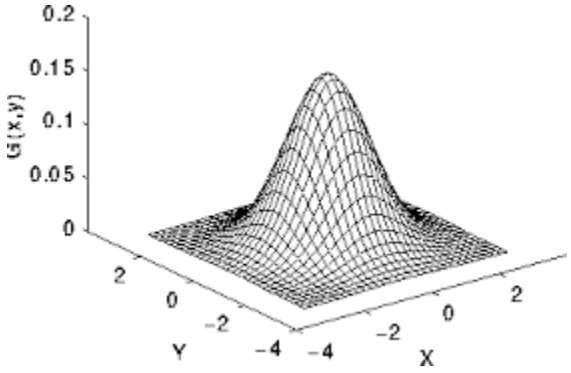
- You can construct $k'(\mathbf{x}, \mathbf{x'})$ from $k_1(\mathbf{x}, \mathbf{x'})$, $k_2(\mathbf{x}, \mathbf{x'})$ by:
  - Adding $k_1(\mathbf{x}, \mathbf{x'})$ and $k_2(\mathbf{x}, \mathbf{x'})$:
    $k'(\mathbf{x}, \mathbf{x'}) = k_1(\mathbf{x}, \mathbf{x'}) + k_2(\mathbf{x}, \mathbf{x'})$
  - Multiplying $k_1(\mathbf{x}, \mathbf{x'})$ and $k_2(\mathbf{x}, \mathbf{x'})$:
    $k'(\mathbf{x}, \mathbf{x'}) = k_1(\mathbf{x}, \mathbf{x'})k_2(\mathbf{x}, \mathbf{x'})$
- Also:
  - If $\phi(\mathbf{x}) \in R^m$ and $k_m(\mathbf{z}, \mathbf{z'})$ a valid kernel in R^m,
    $k(\mathbf{x}, \mathbf{x'}) = k_m(\phi(\mathbf{x}), \phi(\mathbf{x'}))$ is also a valid kernel
  - If $\mathbf{A}$ is a symmetric positive semi-definite matrix,
    $k(\mathbf{x}, \mathbf{x'}) = \mathbf{x}\mathbf{A}\mathbf{x'}$ is also a valid kernel
- In all cases, it is easy to prove these directly by construction.

Gaussian Kernel

(aka Radial Basis Function kernel)

- $k(x, z)\ \exp(-(x - z)^2/c)$
  - $(x - z)^2$: squared Euclidean distance between **x** and **z**
  - $c = 2\sigma^2$: a free parameter
  - very small c: K ≈ identity matrix (every item is different)
  - very large c: K ≈ unit matrix (all items are the same)

  - $k(\mathbf{x}, \mathbf{z}) \approx 1$ when **x**, **z** close
  - $k(\mathbf{x}, \mathbf{z}) \approx 0$ when **x**, **z** dissimilar



Gaussian Kernel

- $k(x, z) = \exp(-(x - z)^2/c)$
- Is this a kernel?
- $k(\mathbf{x}, \mathbf{z}) = \exp(-(\mathbf{x} - \mathbf{z})^2/2\sigma^2)$

  $= \exp(-(\mathbf{xx} + \mathbf{zz} - 2\mathbf{xz})/2\sigma^2)$

  $= \exp(-\mathbf{xx}/2\sigma^2)\ \exp(\mathbf{xz}/\sigma^2)\ \exp(-\mathbf{zz}/2\sigma^2)$

  $= f(\mathbf{x})\ \exp(\mathbf{xz}/\sigma^2)\ f(\mathbf{z})$
- $\exp(\mathbf{xz}/\sigma^2)$ is a valid kernel:
  - $\mathbf{xz}$ is the linear kernel;
  - we can multiply kernels by constants ($1/\sigma^2$)
  - we can exponentiate kernels

Unlike the discrete kernels discussed earlier, here you cannot easily explicitly blow up the feature space to get an identical representation.

Summary – Kernel Based Methods

$$f(x) = \mathbf{Th}_{\theta}\left(\sum_{\mathbf{z}\in M} \mathbf{S(z)K(x,z)}\right)$$

- A method to run Perceptron on a very large feature set, without incurring the cost of keeping a very large weight vector.
- Computing the weight vector can be done in the original feature space.
- Notice: this pertains only to efficiency: the classifier is identical to the one you get by blowing up the feature space.
- Generalization is still relative to the real dimensionality (or, related properties).
- Kernels were popularized by SVMs but apply to a range of models, Perceptron, Gaussian Models, PCAs, etc.

Explicit & Implicit Kernels: Complexity

- Is it always worthwhile to define kernels and work in the dual space?
- Computationally:
  - Let m be # of examples, t_1, t_2 be the sizes of the (Dual, Primal) feature spaces, respectively.
  - Then, computational cost is:
    - Dual space – $t_1 m^2$ vs, Primal Space – $t_2 m$
  - Typically, $t_1 << t_2$, so it boils down to the number of examples one needs to consider relative to the growth in dimensionality.
- Rule of thumb: a lot of examples → use Primal space
- Most applications today: People use explicit kernels. That is, they blow up the feature space explicitly.

Kernels: Generalization

- Do we want to use the most expressive kernels we can?
  - (e.g., when you want to add quadratic terms, do you really want to add all of them?)
- No; this is equivalent to working in a larger feature space, and will lead to overfitting.

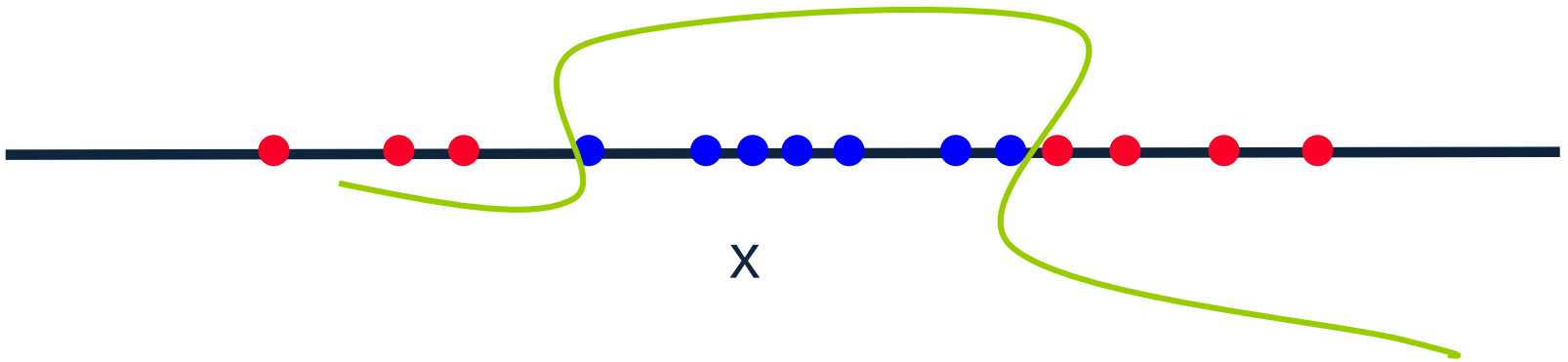
- It's possible to give simple arguments that show that simply adding irrelevant features does not help.

Conclusion- Kernels

- The use of Kernels to learn in the dual space is an important idea
  - Different kernels may expand/restrict the hypothesis space in useful ways.
  - Need to know the benefits and hazards
- To justify these methods we must embed in a space much larger than the training set size.
  - Can affect generalization
- Expressive structures in the input data could give rise to specific kernels, designed to exploit these structures.
  - E.g., people have developed kernels over parse trees: corresponds to features that are sub-trees.
  - It is always possible to trade these with explicitly generated features, but it might help one's thinking about appropriate features.

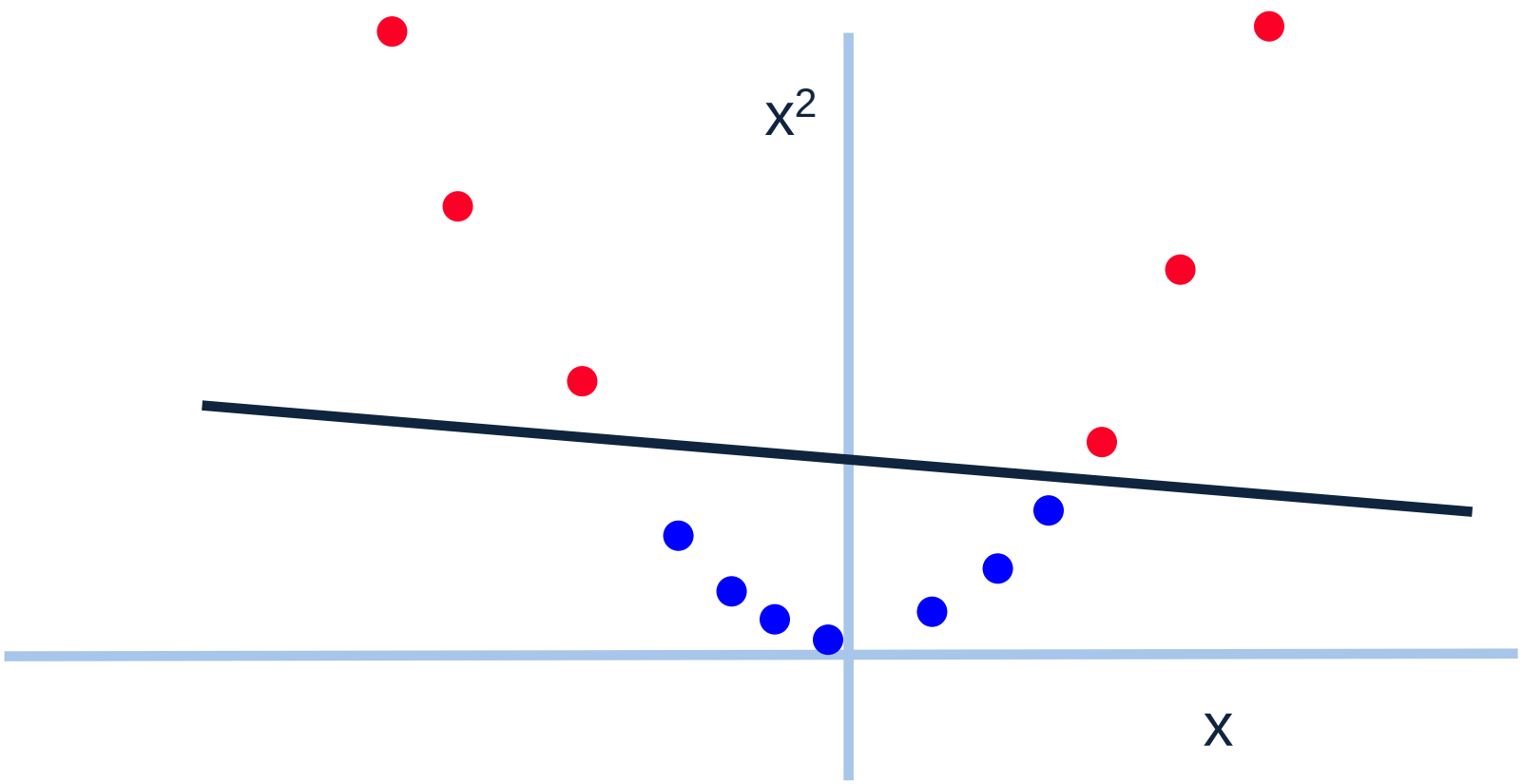
Functions Can be Made Linear

- Data are not linearly separable in one dimension
- Not separable if you insist on using a specific class of functions



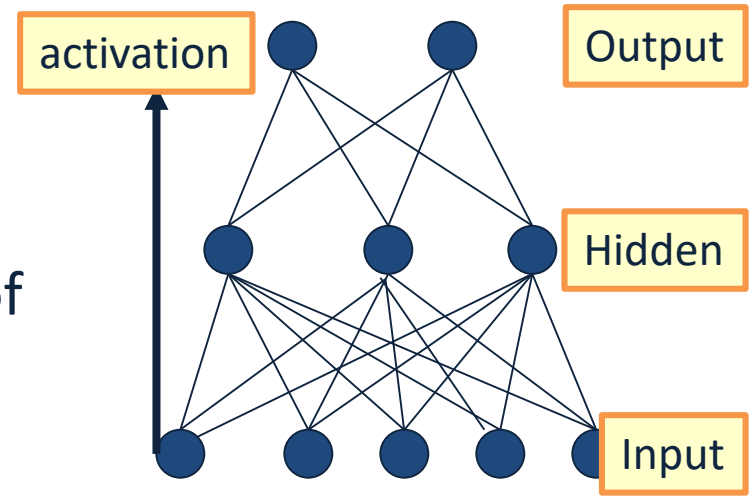
Blown Up Feature Space

- Data are separable in <x, x^2> space



Multi-Layer Neural Network

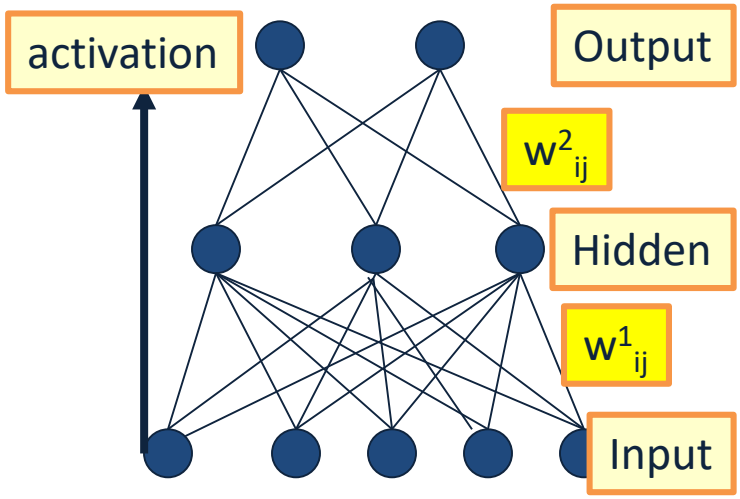
- Multi-layer network were designed to overcome the computational (expressivity) limitation of a single threshold element.
- The idea is to stack several layers of threshold elements, each layer using the output of the previous layer as input.



- Multi-layer networks can represent arbitrary functions, but building effective learning methods for such network was [thought to be] difficult.

Basic Units

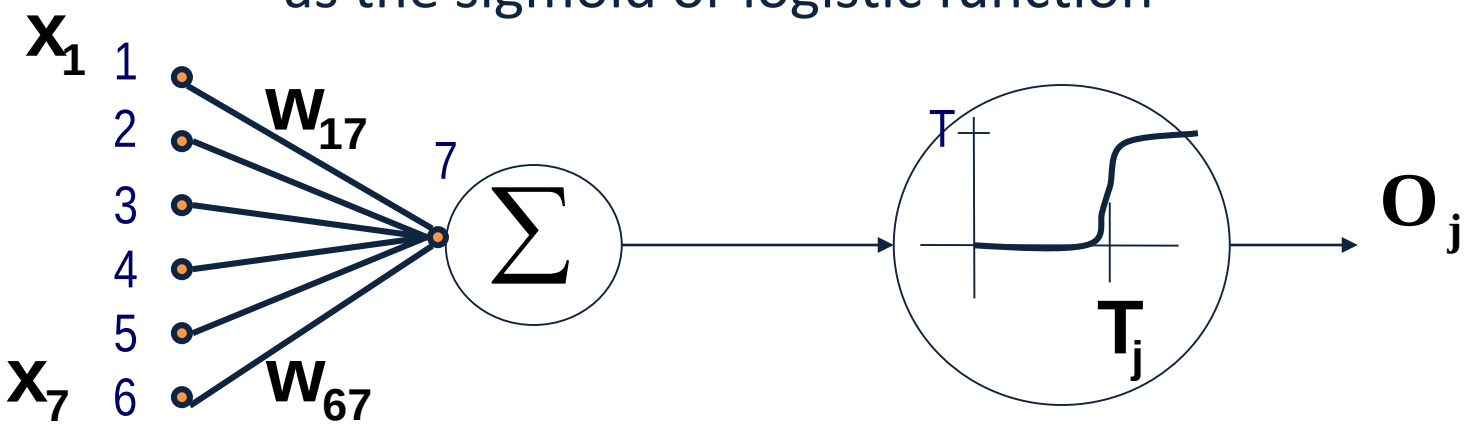
- Linear Unit: Multiple layers of linear functions $o_j = w \cdot x$ produce linear functions. We want to represent nonlinear functions.
- Need to do it in a way that facilitates learning
- Threshold units: $o_j = \text{sgn}(w \cdot x)$ are not differentiable, hence unsuitable for gradient descent.



- The key idea was to notice that the discontinuity of the threshold element can be represents by a smooth non-linear approximation: $o_j = [1+ \exp\{-w \cdot x\}]^{-1}$
  - (Rumelhart, Hinton, Williiam, 1986), (Linnainmaa, 1970) , see: http://people.idsia.ch/~juergen/who-invented-backpropagation.html)

Model Neuron (Logistic)

- Us a non-linear, differentiable output function such as the sigmoid or logistic function

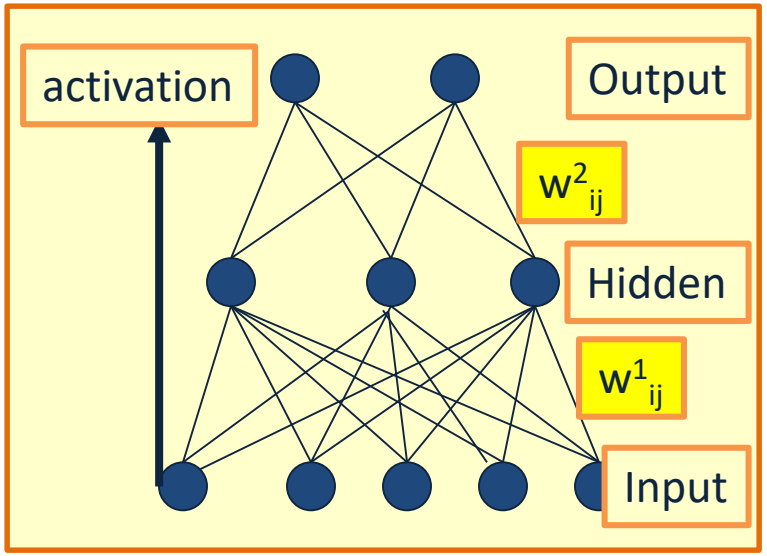


- Net input to a unit is defined as: $\mathbf{net_j} = \sum \mathbf{w_{ij}} \bullet \mathbf{x_i}$
- Output of a unit is defined as:

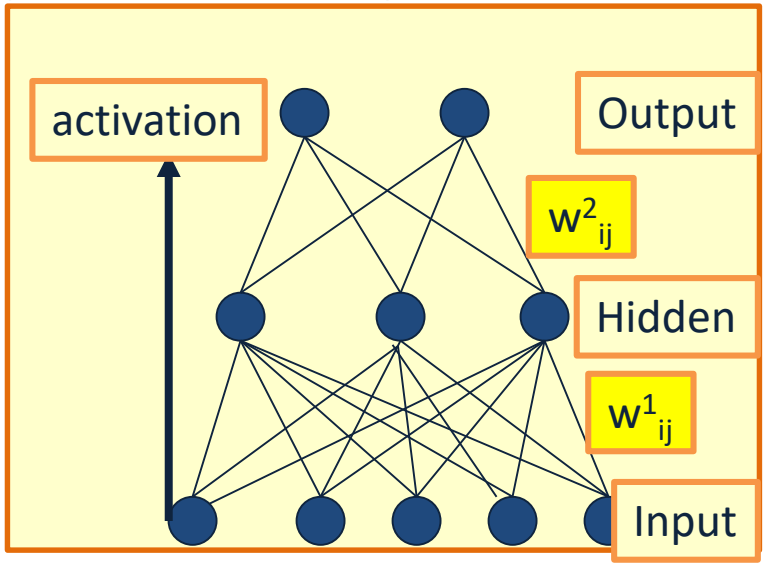
$$\mathbf{O_j} = \frac{\mathbf{1}}{\mathbf{1} + \mathbf{e}^{-(\mathbf{net_j} - \mathbf{T_j})}}$$

Learning with a Multi-Layer Perceptron

- It's easy to learn the top layer – it's just a linear unit.
- Given feedback (truth) at the top layer, and the activation at the layer below it, you can use the Perceptron update rule (more generally, gradient descent) to updated these weights.
- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).



Learning with a Multi-Layer Perceptron



- The problem is what to do with the other set of weights – we do not get feedback in the intermediate layer(s).
- Solution: If all the activation functions are differentiable, then the output of the network is also a differentiable function of the input and weights in the network.
- Define an error function (multiple options) that is a differentiable function of the output, that this error function is also a differentiable function of the weights.
- We can then evaluate the derivatives of the error with respect to the weights, and use these derivatives to find weight values that minimize this error function. This can be done, for example, using gradient descent .
- This results in an algorithm called back-propagation.