The Perceptron

William Cohen

10-601

Aside: Logistic Regression: Notational Differences

Ziv's notation

Logistic Regression

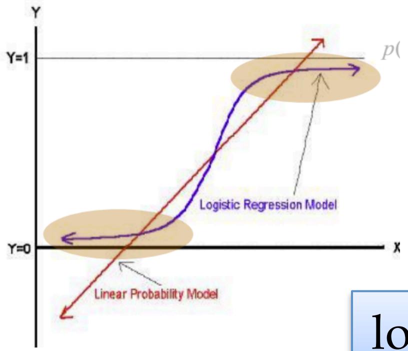
Defining a new function, g

$$p(y=0 \mid X;\theta) = g(X;w) = \frac{1}{1+e^{\mathrm{w}^{\mathrm{T}}X}}$$

$$p(y=1 \mid X;\theta) = 1 - g(X;w) = \frac{e^{\mathrm{w}^{\mathrm{T}}X}}{1+e^{\mathrm{w}^{\mathrm{T}}X}}$$

Data likelihood

$$L(y \mid X;w) = \prod_i (1-g(X_i;w))^{y_i}\, g(X_i;w)^{(1-y_i)}$$



$$LL(y \mid X;w) = \sum_{i=1}^{N} y_i \mathrm{w}^{\mathrm{T}} X_i - \ln(1+e^{w^T X_i})$$

log data likelihood

Logistic regression via gradient ascent: MLE for log likelihood

William's notation

$$\frac{\partial}{\partial w^j} \log P(Y = y|X = \mathbf{x}, \mathbf{w}) = (y - p)x^j$$

1. Chose λ
2. Start with a guess for **w**
3. For all j set

$$w^j \leftarrow w^j + \varepsilon \sum_{i=1}^{N} X_i^j \{y_i - (1 - g(X_i; w))\}$$

4. If no improvement for

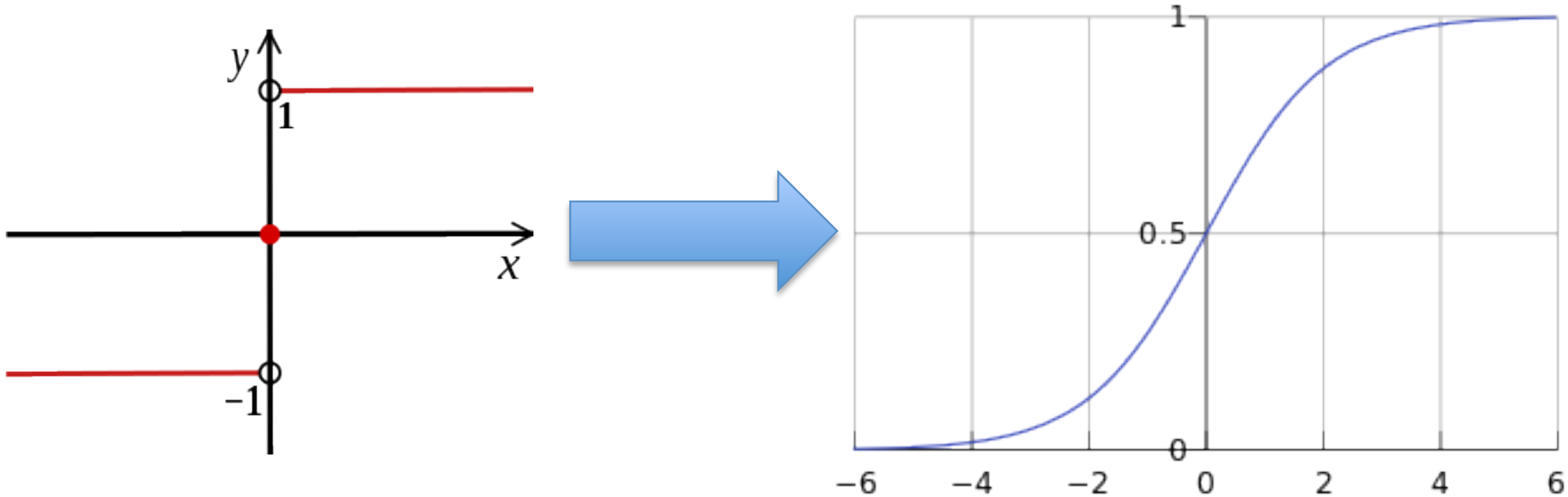
$$LL(y \mid X; w) = \sum_{i=1}^{N} y_i \ln(1 - g(X_i; w)) + (1 - y_i) \ln g(X_i; w)$$

stop. Otherwise go to step 3

William's notation

Logistic regression

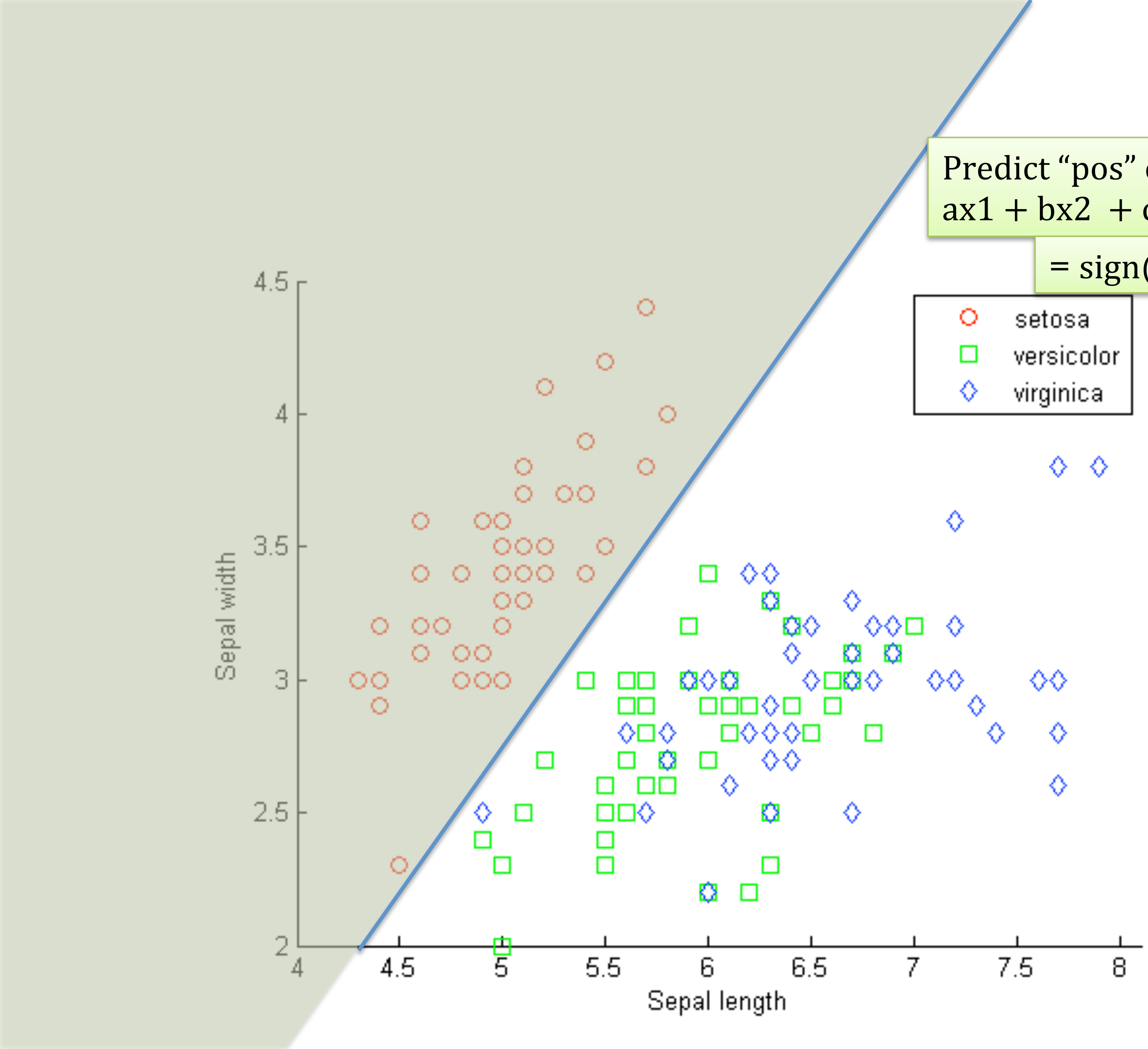
$$P(y_i \mid \mathbf{x}_i, \mathbf{w}) \equiv \begin{cases} \dfrac{1}{1+\exp(-\mathbf{x}_i \cdot \mathbf{w})} & \text{if } y_i = 1 \\ \left(1 - \dfrac{1}{1+\exp(-\mathbf{x}_i \cdot \mathbf{w})}\right) & \text{if } y_i = 0 \end{cases}$$



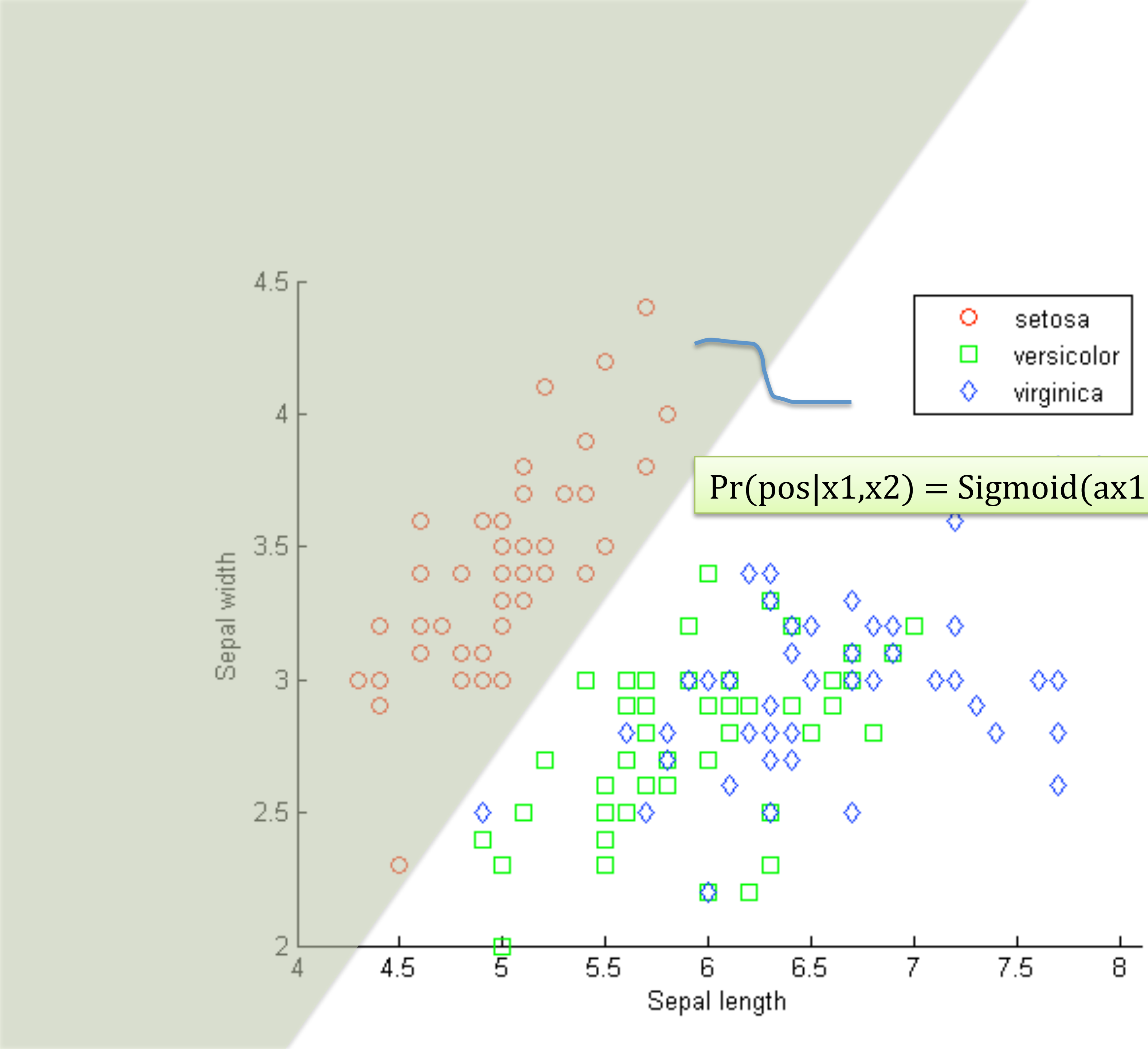
$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

Predict “pos” on (x1,x2) iff ax1 + bx2 + c >0

= sign(ax1 + bx2 + c)



Logistic regression as “soft” linear classifier



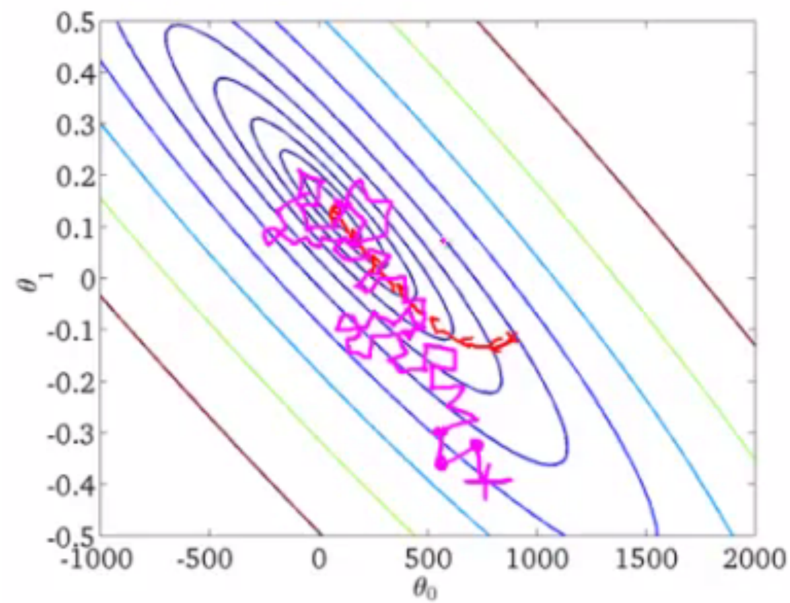
$$\Pr(pos|x1,x2) = \text{Sigmoid}(ax1+bx2+c)$$

Aside: Logistic Regression Stochastic vs “Batch” Gradient

Stochastic gradients (SGD) for logistic regression

1. P(y|**x**)=logistic(**x . w**)
2. Log conditional likelihood: $$\mathrm{LCL}_D(\mathbf{w}) \equiv \sum_i \log P(y_i \mid \mathbf{x}_i, \mathbf{w})$$
3. Differentiate the LCL function and use gradient descent to minimize
   - Start with $\mathbf{w}_0$
   - For t=1,…,T - *until convergence*
     - For each example ***x**,y* in D:
     - $\mathbf{w}_{t+1} = \mathbf{w}_t + \lambda\, L_{x,y}(\mathbf{w}_t)$

       where λ is small



More steps, noisier path toward the minimum, but each step is cheaper

Breaking it down: SGD for logistic regression

1. P(y|**x**)=logistic*(***x . w***)*
2. Define a function

$$\mathrm{LCL}_D(\mathbf{w}) \equiv \sum_i \log P(y_i \mid \mathbf{x}_i, \mathbf{w})$$

3. Differentiate the function and use gradient descent
   - Start with $\mathbf{w}_0$
   - For t=1,…,T - *until convergence*
     - For each example ***x,****y* in D:

       $$p_i = \left(1 + \exp(-\mathbf{x} \cdot \mathbf{w})\right)^{-1}$$

     - $\mathbf{w}_{t+1} = \mathbf{w}_t + \lambda L_{x,y}(\mathbf{w}_t)$ $= \mathbf{w}_t + \lambda(y - p_i)\mathbf{x}$

       where λ is small

Aside: Logistic Regression and Regularization

Non-stochastic gradient descent

$$\frac{\partial}{\partial w^j} \log P(Y = y|X = \mathbf{x}, \mathbf{w}) = (y - p)x^j$$

- In batch gradient descent, average the gradient over all the examples $D=\{(x_1,y_1)\ldots,(x_n , y_{n)}\}$

$$\frac{\partial}{\partial w^j} \log P(D|\mathbf{w}) = \frac{1}{n}\sum_i (y_i - p_i)x_i^j =$$

$$= \boxed{\frac{1}{n}\sum_{i:x_i^j=1} y_i} - \boxed{\frac{1}{n}\sum_{i:x_i^j=1} p_i}$$

Non-stochastic gradient descent

- This can be interpreted as a difference between the expected value of $y \mid x^j\!=\!1$ *in the data* and the expected value of $y \mid x^j\!=\!1$ *as predicted by the model*
- Gradient ascent tries to make those equal

$$\frac{\partial}{\partial w^j} \log P(D|\mathbf{w}) = \frac{1}{n}\sum_i (y_i - p_i) x_i^j =$$

$$= \boxed{\frac{1}{n}\sum_{i:x_i^j=1} y_i} - \boxed{\frac{1}{n}\sum_{i:x_i^j=1} p_i}$$

This LCL function "overfits"

- This can be interpreted as a difference between the expected value of $y \mid x^j=1$ *in the data* and the expected value of $y \mid x^j=1$ *as predicted by the model*
- Gradient ascent tries to make those equal

$$\frac{\partial}{\partial w^j} \log P(D|\mathbf{w}) = \frac{1}{n}\sum_i (y_i - p_i)x_i^j = \frac{1}{n}\sum_{i:x_i^j=1} y_i - \frac{1}{n}\sum_{i:x_i^j=1} p_i$$

- That's impossible for some w^j !
  - e.g., if $w^j = 1$ only in positive examples, the gradient is always positive

Regularization

Ziv's notation

- For example, lets assume that w^j comes from a Gaussian distribution with mean 0 and variance σ^2 (where σ^2 is a user defined parameter): $w^j \sim N(0, \sigma^2)$
- In that case we have **a prior** on the parameters and so:

$$p(y=1,\theta \mid X) \propto p(y=1 \mid X;\theta)\,p(\theta)$$

- Here we use a Gaussian model for the prior.
- Thus, the log likelihood changes to :

$$LL(y;w \mid X) = \sum_{i=1}^{N} y_i \mathrm{w}^{\mathrm{T}} X_i - \ln(1+e^{w^T X_i}) - \sum_j \frac{(w^j)^2}{2\sigma^2}$$

Assuming mean of 0 and removing terms that are not dependent on w

Regularization

Ziv's notation

$$p(y=1,\theta \mid X) \propto p(y=1 \mid X;\theta)\,p(\theta)$$

- Here we use a Gaussian model for the prior.
- Thus, the log likelihood changes to :

$$LL(y;w \mid X) = \underline{\sum_{i=1}^{N} y_i \mathrm{w}^{\mathrm{T}} X_i - \ln(1+e^{w^T X_i})} - \sum_j \frac{(w^j)^2}{2\sigma^2}$$

Assuming mean of 0 and removing terms that are not dependent on w

If we differentiate to get the new gradient, we get the <u>old MLE gradient</u> plus a new term:

$$w^j \leftarrow w^j + \varepsilon \sum_{i=1}^{N} X_i^j \{y_i - (1 - g(X_i;w))\} - \varepsilon \frac{w^j}{\sigma^2}$$

Also known as the MAP estimate

The variance of our prior model

Naïve Bayes is also linear

Naïve Bayes

- Given a new instance with $X_i = x_{ij}$ compute the following for each possible value y of Y

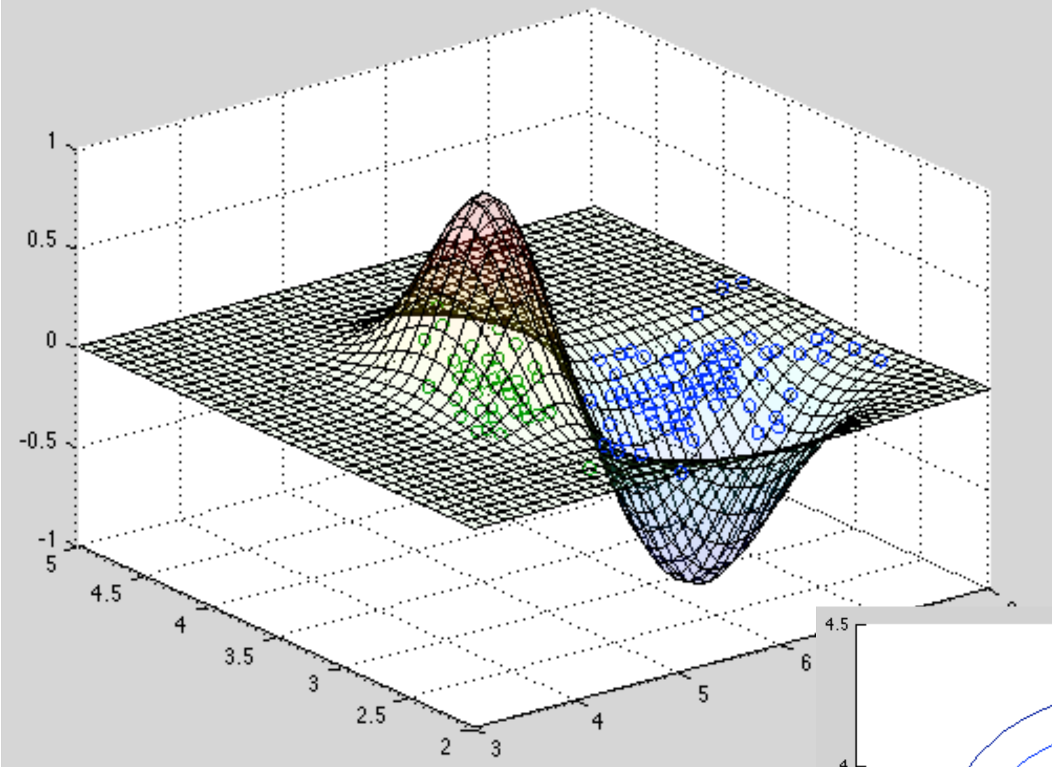
$$= \arg\max_{y_k} P(X_1 = x_{j,1} \mid Y = y) * \ldots * P(X_n = x_{j,n} \mid Y = y_k) P(Y = y_k)$$

$$= \arg\max_{y_k} \prod_i \underline{P(X_i = x_{ji} \mid Y = y_k)} \underline{P(Y = y_k)}$$

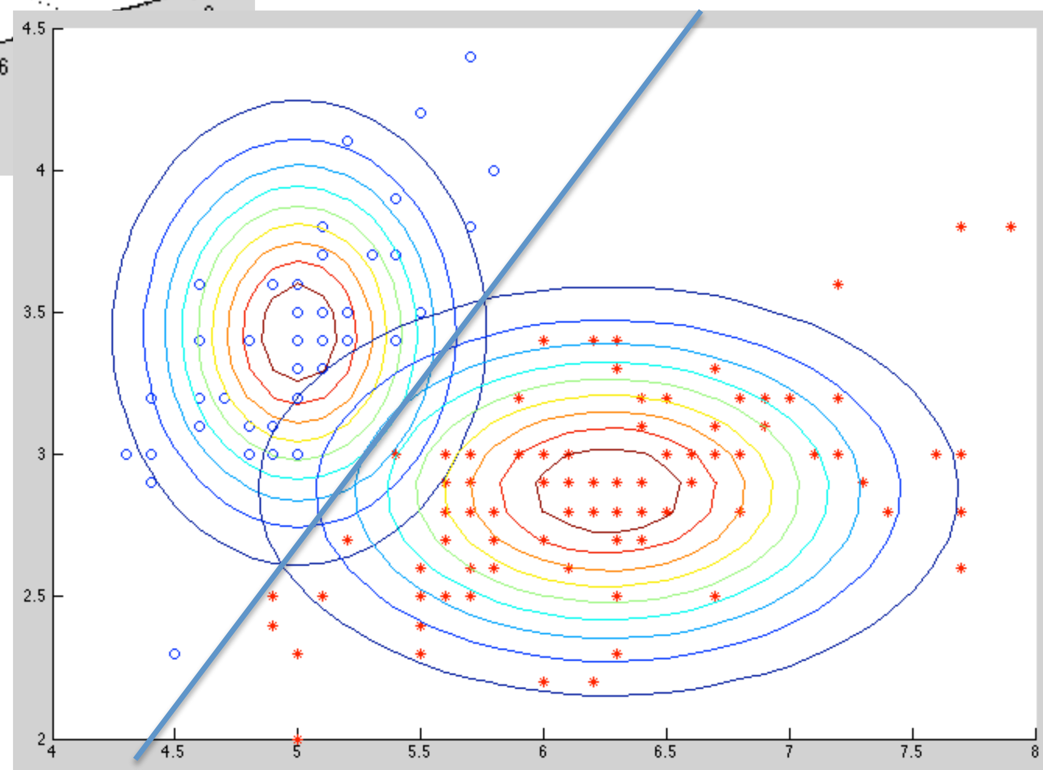
$$= \arg\max_k \sum_{i=1}^{n} \underline{\log q(i, j_i, k)} + \underline{\log p(k)}$$

$$= sign\left[\left(\sum_{i=1}^{n} \log q(i, j_i, \mathrm{pos}) + \log p(\mathrm{pos})\right) - \left(\sum_{i=1}^{n} \log q(i, j_i, \mathrm{neg}) + \log p(neg)\right)\right]$$

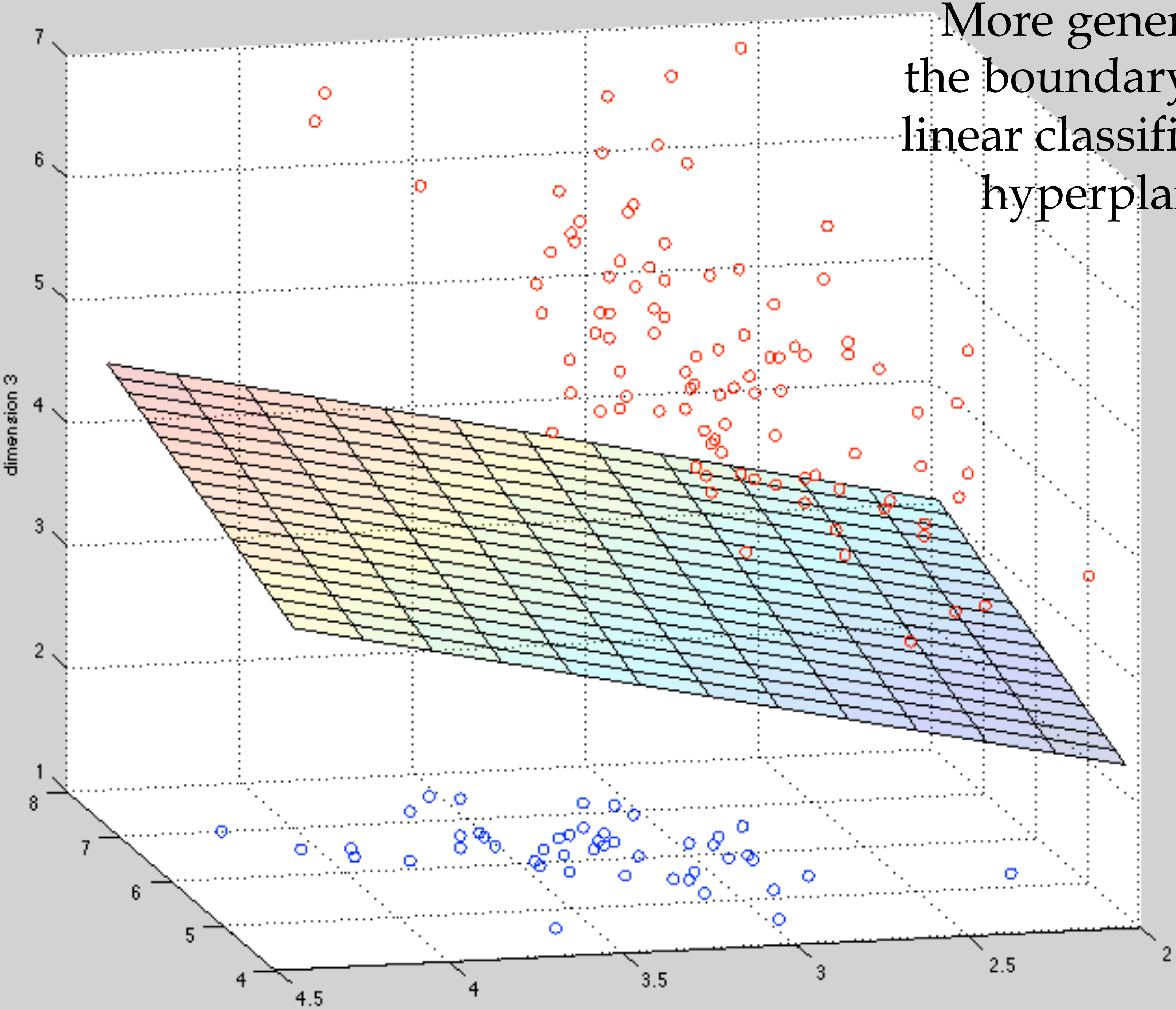
for two classes y_1=pos, y_2=neg



For two classes Naïve Bayes is linear

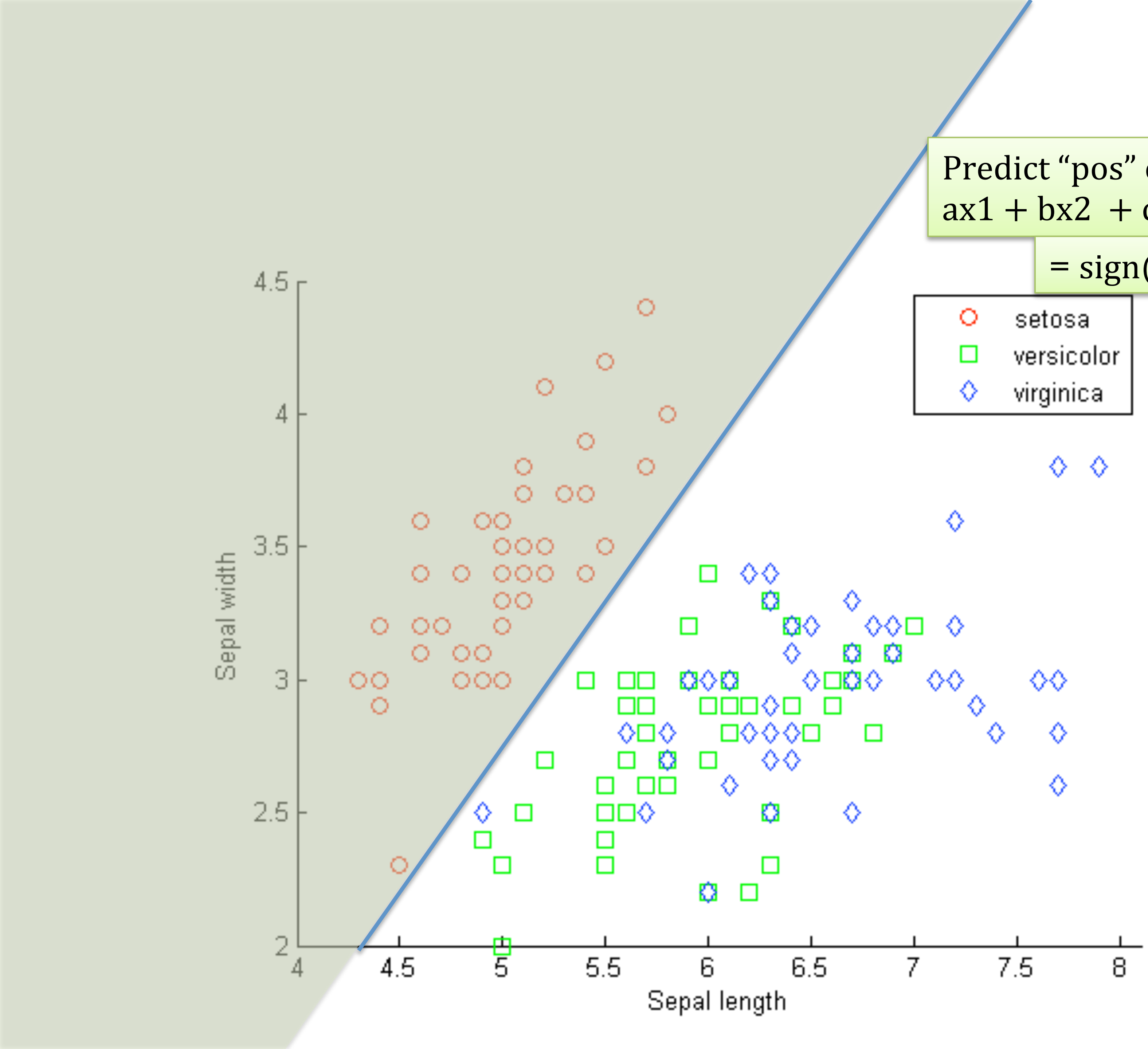


More generally the boundary for a linear classifier is a hyperplane



Predict “pos” on (x1,x2) iff $ax1 + bx2 + c > 0$

$= \text{sign}(ax1 + bx2 + c)$



LOGISTIC REGRESSION AND LINEAR CLASSIFIERS

Linear classifiers we've seen so far

- Naïve Bayes:
  - a generative linear classifier
  - can show the decision boundary is linear
- Logistic regression:
  - a discriminative linear classifier
  - same functional form (linear) but optimize the LCL $\log Pr(y \mid \boldsymbol{x})$, not the joint likelihood $\log Pr(\boldsymbol{x}, y)$
- Do we need anything else?

Questions:

- Why optimize LCL if we want to reduce errors on the test data?
- Assume there is a linear classifier: does that always make learning "easy"? can we quantify how "easy" a learning problem is?

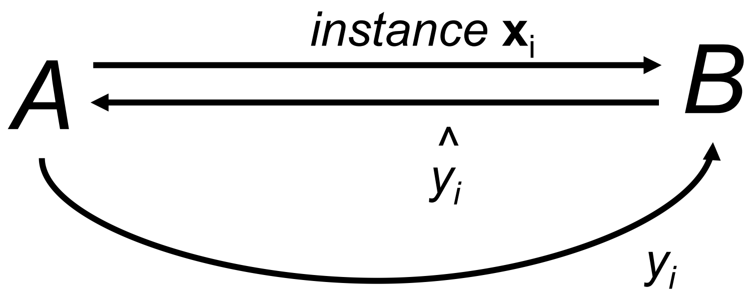
Another analytic approach

- Start with a simple learner and analyze what it does
- Goals:
  - capture *geometric* intuitions about what makes learning hard or easy
  - analyze performance worst-case settings
  - analyze existing plausible learning methods
    - e.g. in studying human learning, biology, …
- This particular analysis is simple enough to give some insight into "margin" learning
- See: Freund & Schapire, 1998

MISTAKE BOUNDS FOR THE PERCEPTRON

The perceptron

[Rosenblatt, 1957]



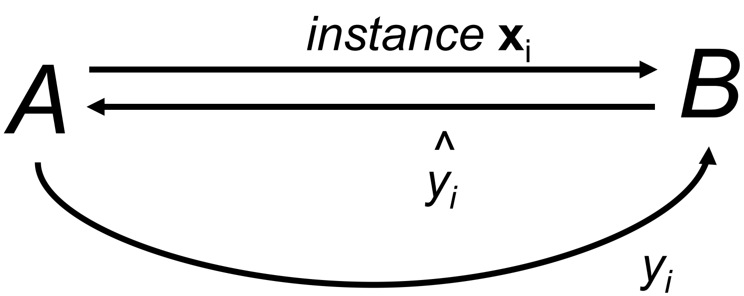
Compute: $\hat{y}_i = \text{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

- On-line setting:
  - Adversary A provides student B with an *instance* **x**
  - Student B predicts a class (+1, -1) according to a simple linear classifier: sign($\mathbf{v}_k \cdot \mathbf{x}$)
  - Adversary gives student the answer (+1,-1) for that instance
- Will do a *worst-case* analysis of the mistakes made by the student over *any* sequence of instances from the adversary
  - … that follow a few rules

The perceptron

[Rosenblatt, 1957]



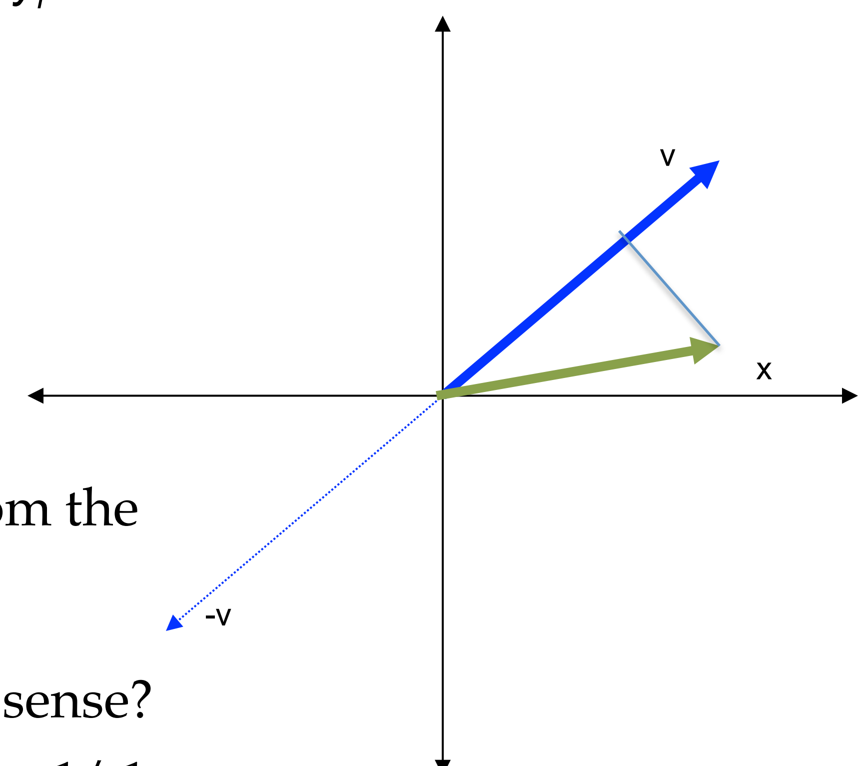
Compute: $\hat{y}_i = \text{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

- Recall dot product definition:

$$\mathbf{x} \bullet \mathbf{v} = \sum_{\mathbf{i}} x_i v_i$$

- and intuition:
  - project vector **x** onto vector **v**
  - dot product is the distance from the origin of that projection

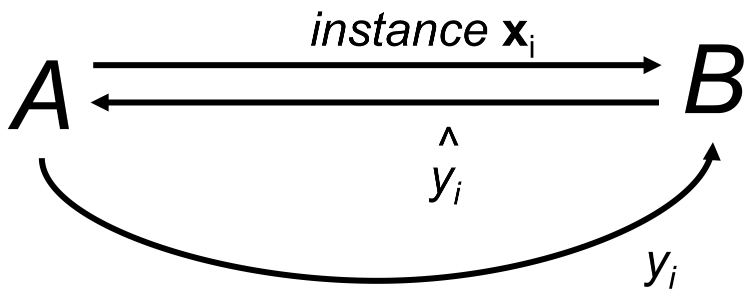


- So why does this algorithm make sense?

  cases: actual = +1/-1, predicted = +1/-1

The perceptron

[Rosenblatt, 1957]



Compute: $\hat{y}_i = \text{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

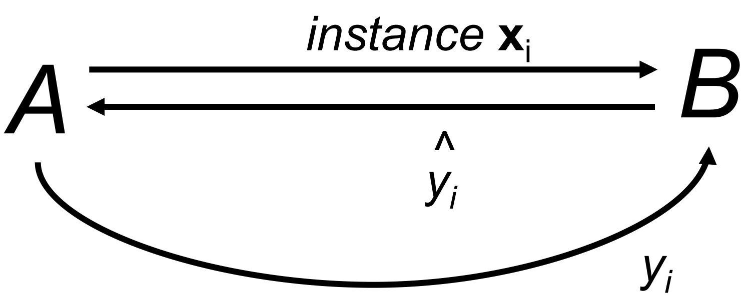
Logistic update: $\mathbf{v}_{k+1} = \mathbf{v}_k + \varepsilon(y_i - p_i)\mathbf{x}_i$

$\varepsilon=1 \Rightarrow \quad = \mathbf{v}_k + y_i \mathbf{x}_i - p_i \mathbf{x}_i$

cases: actual = 1/0, predicted = 1/0

The perceptron

[Rosenblatt, 1957]



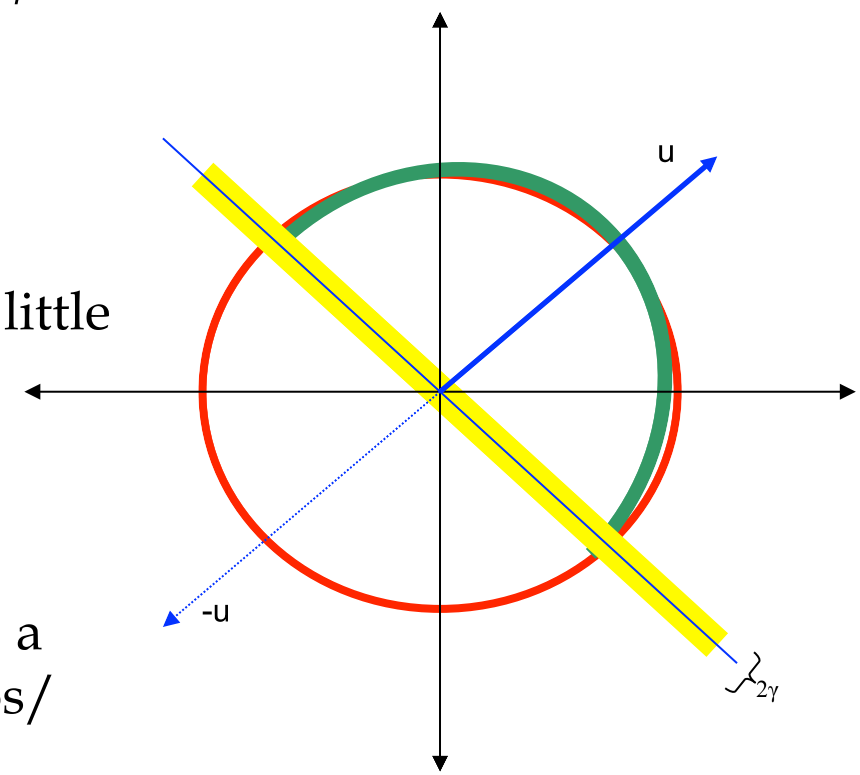
Compute: $\hat{y}_i = \text{sign}(\mathbf{v}_k \,.\, \mathbf{x}_i)$

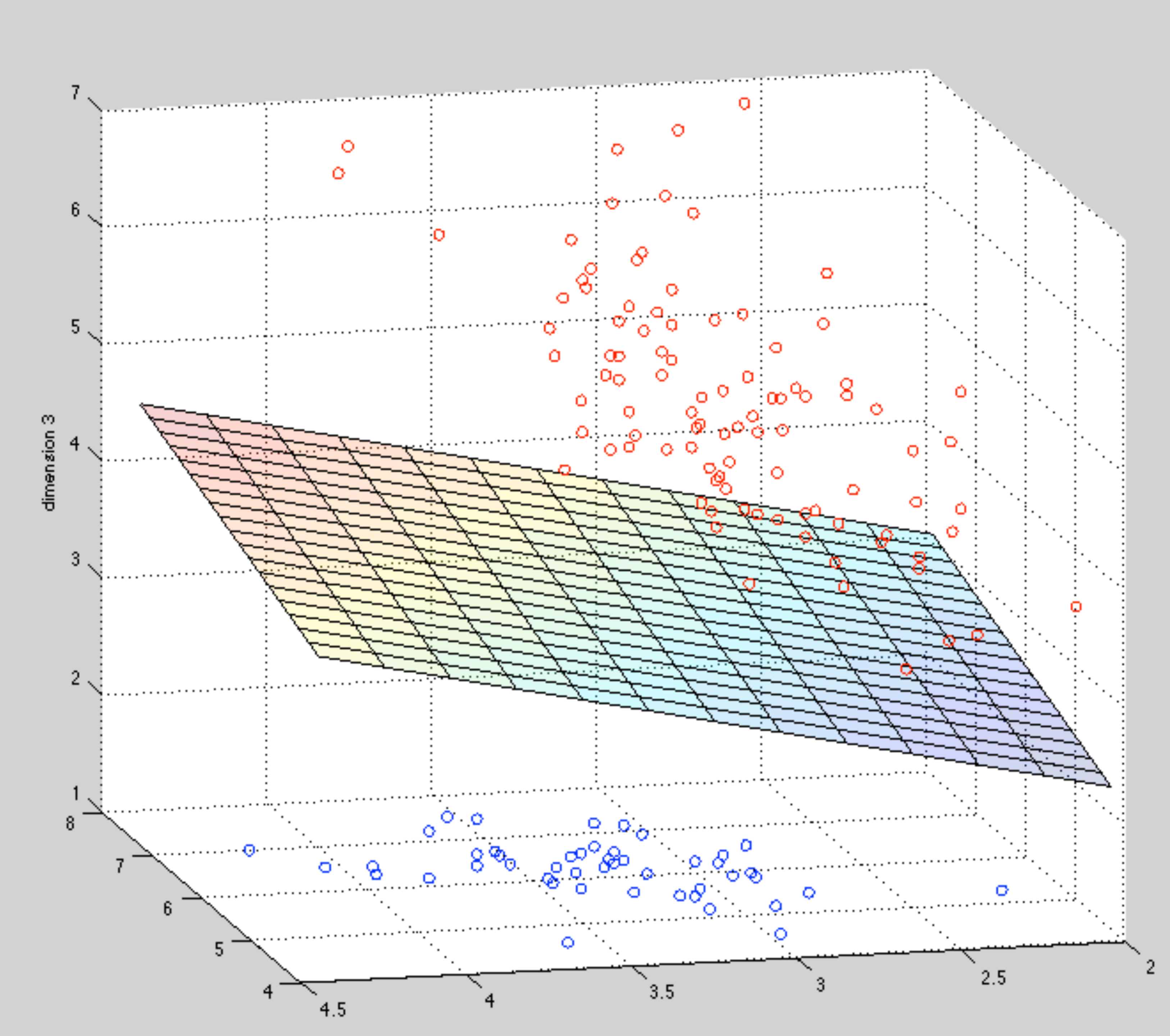
If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i\, \mathbf{x}_i$

- Amazingly simple algorithm
- Quite effective
- Very easy to *understand* if you do a little linear algebra

•*Two rules:*

  - Examples are not too "big"
  - There is a "good" answer -- i.e. a line that clearly separates the pos/neg examples

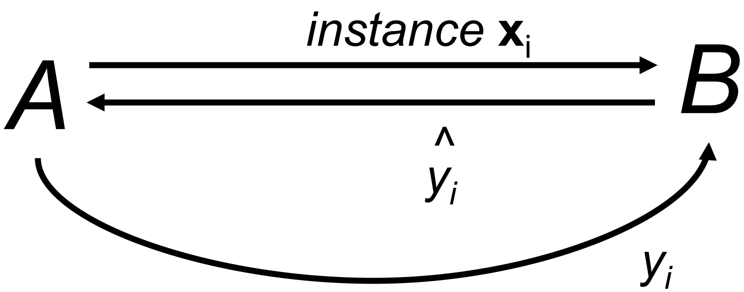




dimension 3
7
6
5
4
3
2
1
8
7
6
5
4
4.5
4
3.5
3
2.5
2

The perceptron

[Rosenblatt, 1957]



Compute: $\hat{y}_i = \text{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

Rule 1: Radius R: A must provide examples "near the origin"

$$\forall \mathbf{x}_i \text{ given by A}, \|\mathbf{x}_i\|_2^2 \leq R^2$$

$$\|\mathbf{x}\|_2 = \sqrt{(x_1^2 + \ldots + x_n^2)}$$

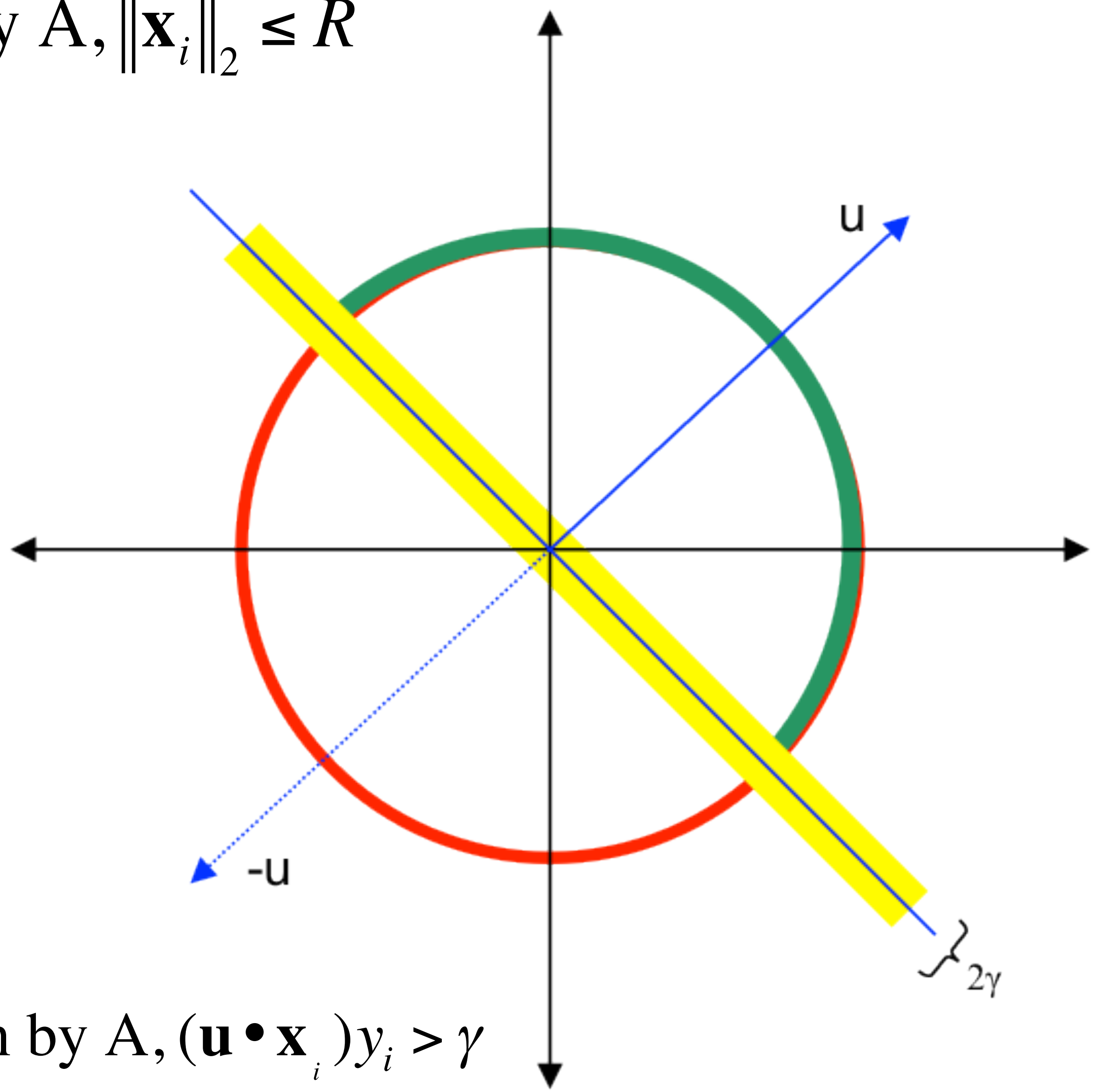
Rule 2: Margin γ: A must provide examples that can be separated with some vector **u** with margin γ >0 and unit norm

$$\exists \mathbf{u} : \forall \mathbf{x}_i \text{ given by A}, (\mathbf{u} \bullet \mathbf{x}_i) y_i > \gamma$$

$$\text{and } \|\mathbf{u}\|_2 = 1$$

"margin"

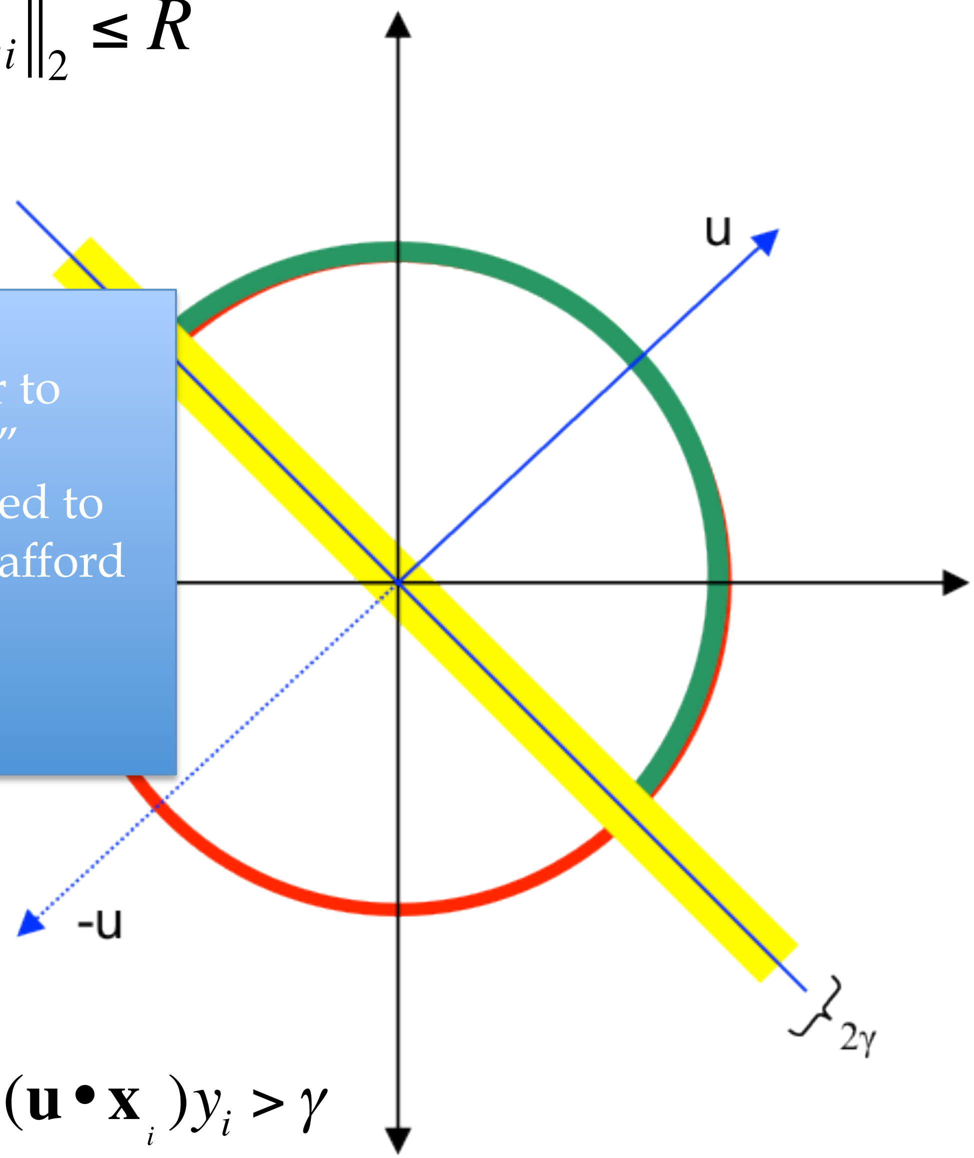
$\forall \mathbf{x}_i$ given by A, $\|\mathbf{x}_i\|_2 \leq R$



$\exists \mathbf{u} : \forall \mathbf{x}_i$ given by A, $(\mathbf{u} \bullet \mathbf{x}_i) y_i > \gamma$

and $\|\mathbf{u}\|_2 = 1$

$\forall \mathbf{x}_i$ given by A, $\|\mathbf{x}_i\|_2 \leq R$



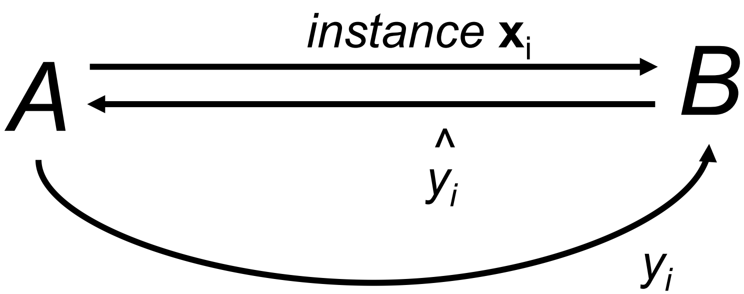
Comments:

1. Scale shouldn't matter to "how hard learning is"
2. Wide margin (compared to R) means that we can afford larger errors in our estimation of **u**

$\exists \mathbf{u} : \forall \mathbf{x}_i$ given by A, $(\mathbf{u} \bullet \mathbf{x}_i) y_i > \gamma$

and $\|\mathbf{u}\|_2 = 1$

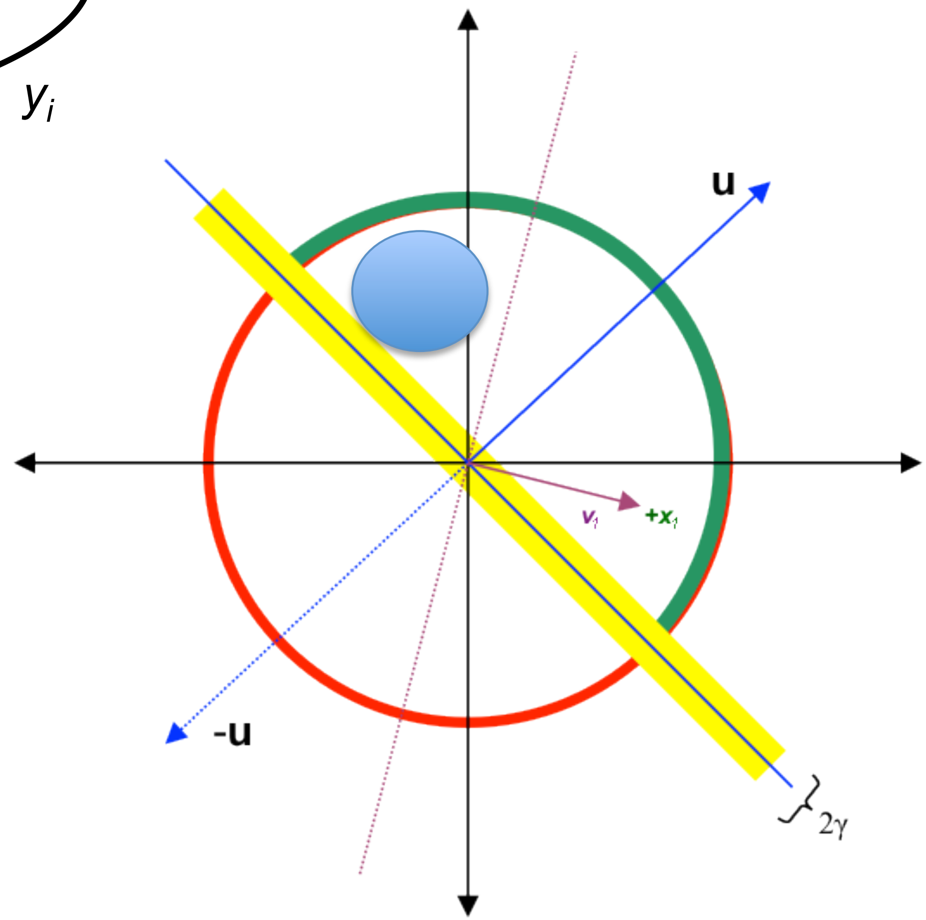
The perceptron: after one positive $\mathbf{x}_i$



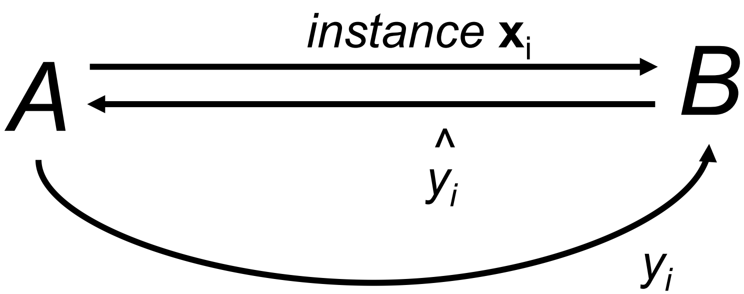
Compute: $\hat{y}_i = \mathrm{sign}(\mathbf{v}_k \,.\, \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i\, \mathbf{x}_i$

What region would cause a mistake on a positive example?

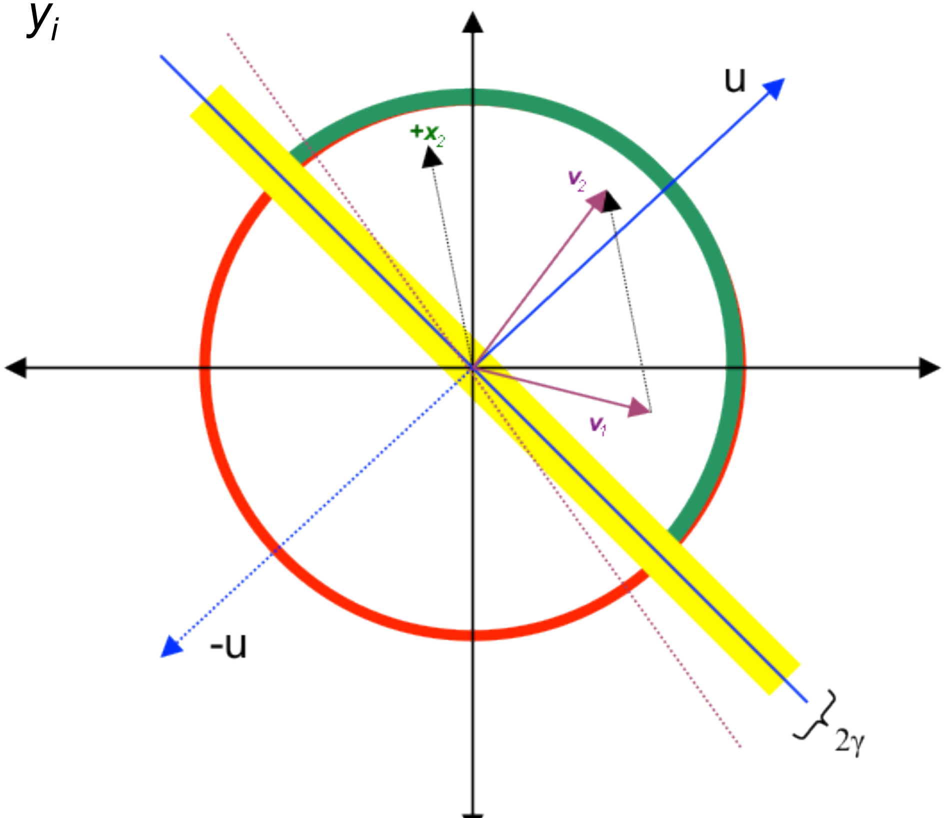


The perceptron: after two positive $\mathbf{x}_i$

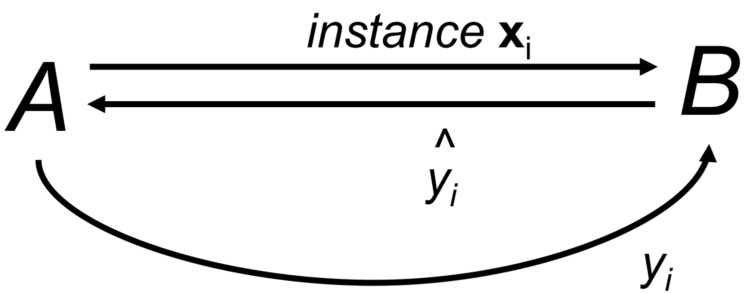


Compute: $\hat{y}_i = \mathrm{sign}(\mathbf{v}_k \,.\, \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i\, \mathbf{x}_i$



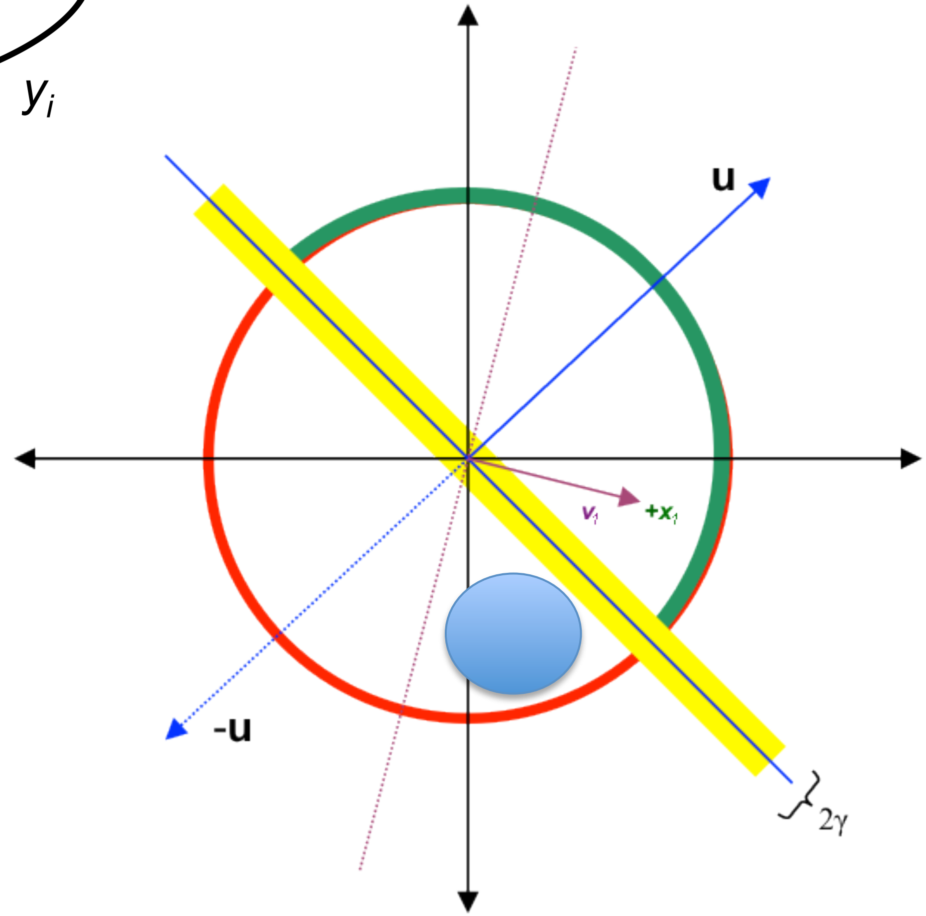
The perceptron: after one positive $\mathbf{x}_i$



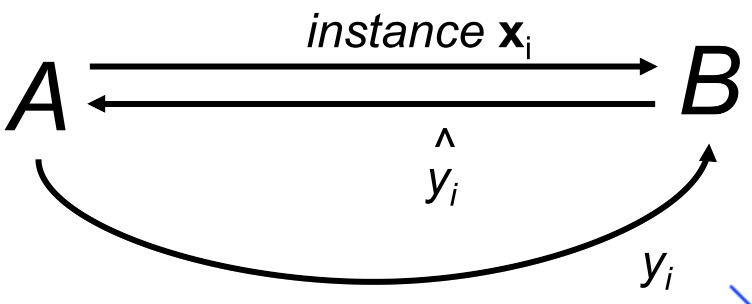
Compute: $\hat{y}_i = \mathrm{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

What region would cause a mistake on a negative example?

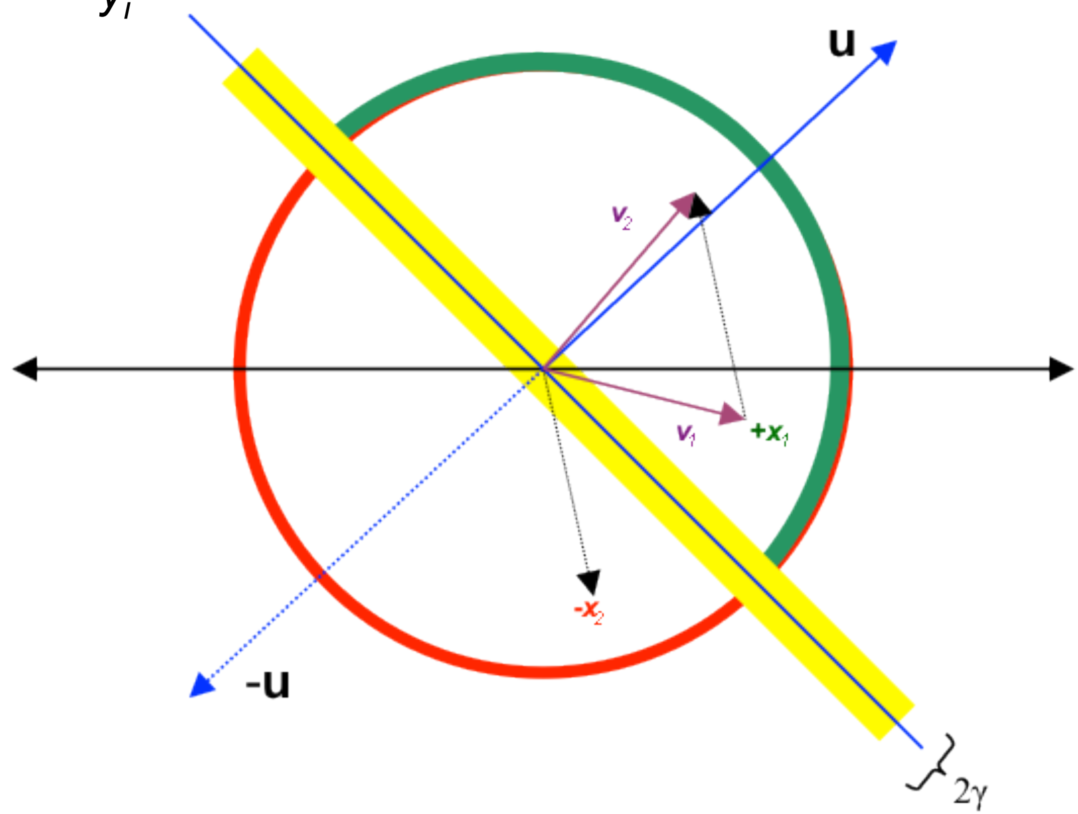


The perceptron: after one pos + one neg $\mathbf{x}_i$

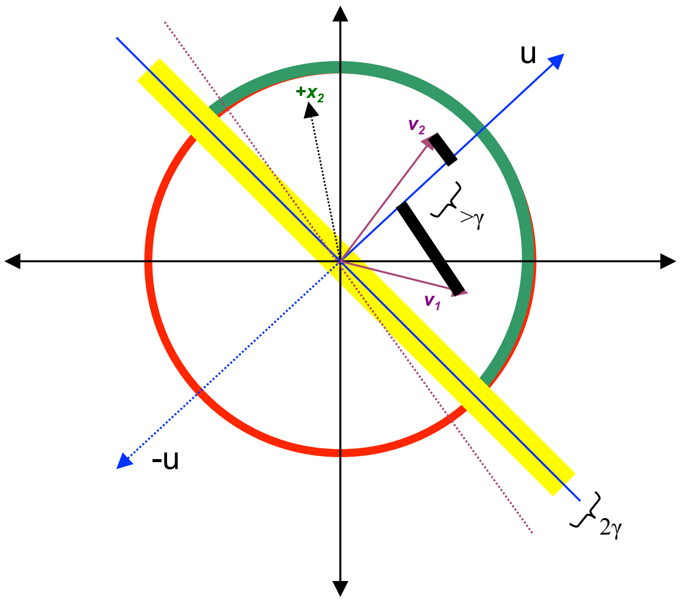


Compute: $\hat{y}_i = \mathrm{sign}(\mathbf{v}_k \,.\, \mathbf{x}_i)$

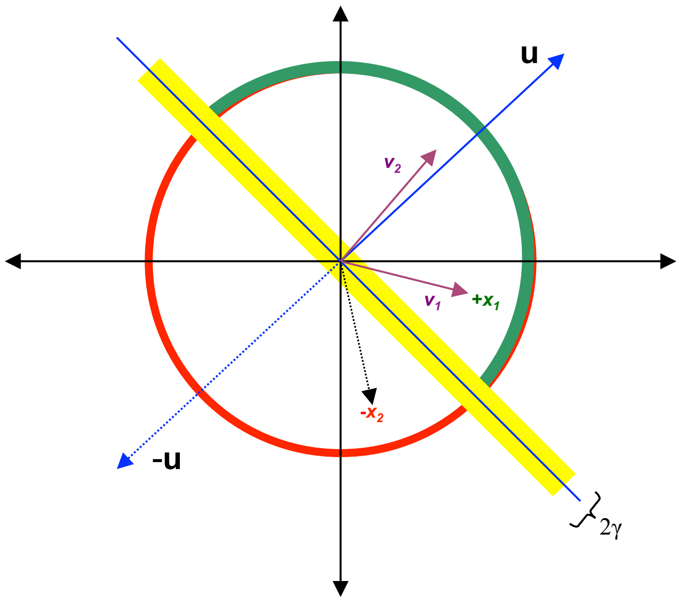
If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$



The guess $\mathbf{v_2}$ after the two positive examples: $\mathbf{v}_2=\mathbf{v}_1+\mathbf{x}_2$



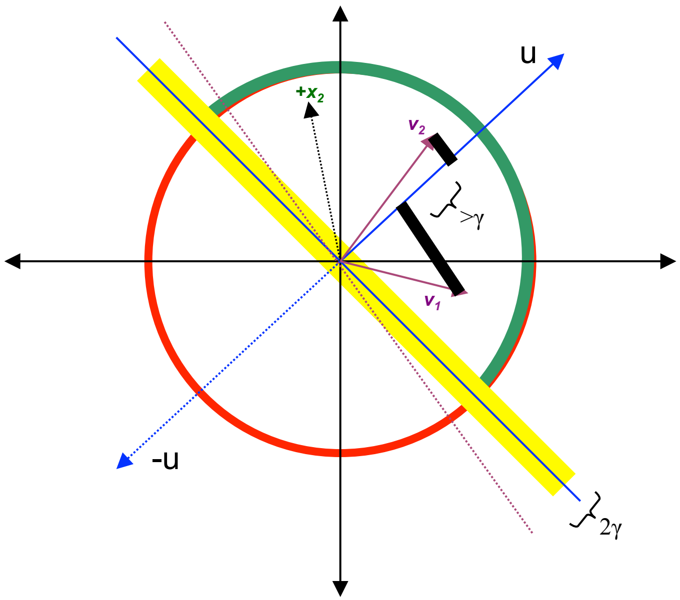
The guess $\mathbf{v_2}$ after the one positive and one negative example: $\mathbf{v}_2=\mathbf{v}_1-\mathbf{x}_2$



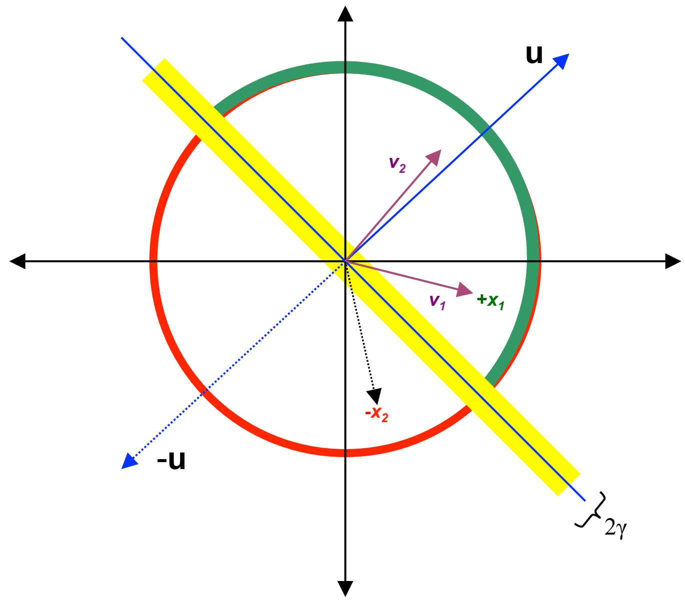
Lemma 1: the dot product between $\mathbf{v}_k$ and $\mathbf{u}$ increases with each mistake by at last γ: i.e.,

$$\forall k : \mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$$

The guess $\mathbf{v}_2$ after the two positive examples: $\mathbf{v}_2=\mathbf{v}_1+\mathbf{x}_2$



The guess $\mathbf{v}_2$ after the one positive and one negative example: $\mathbf{v}_2=\mathbf{v}_1-\mathbf{x}_2$



Lemma 1: the dot product between $\mathbf{v}_k$ and $\mathbf{u}$ increases with each mistake by at last γ: i.e.,

$$\forall k : \mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$$

$$\mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k + y_i \mathbf{x}_i) \cdot \mathbf{u}$$

$$\mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k \cdot \mathbf{u}) + y_i (\mathbf{x}_i \cdot \mathbf{u})$$

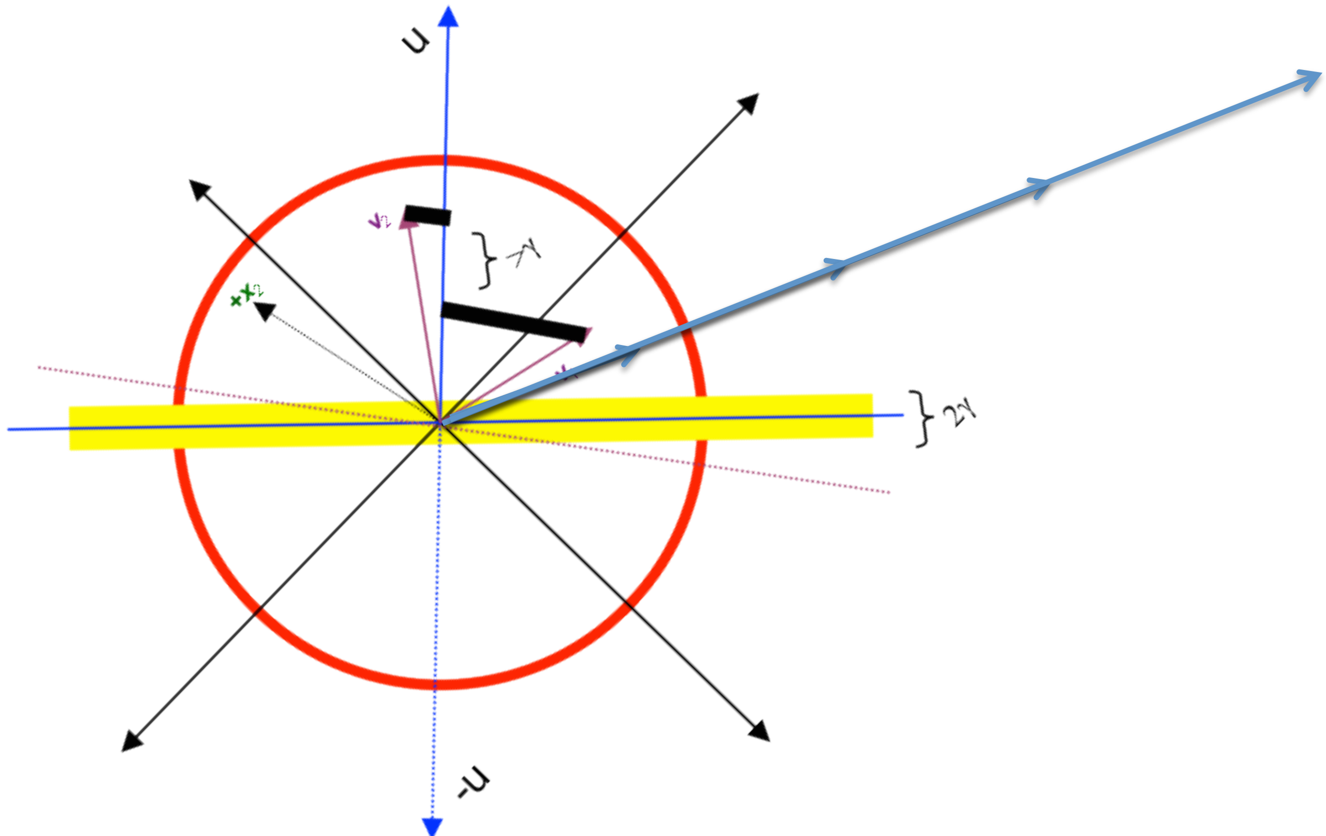
$$\mathbf{v}_{k+1} \cdot \mathbf{u} \geq (\mathbf{v}_k \cdot \mathbf{u}) + \gamma$$

$\exists \mathbf{u} : \forall \mathbf{x}_i$ given by A, $(\mathbf{u} \bullet \mathbf{x}_i) y_i > \gamma$

SO …

$$\mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$$

Some people see this more readily when **u** is "up"

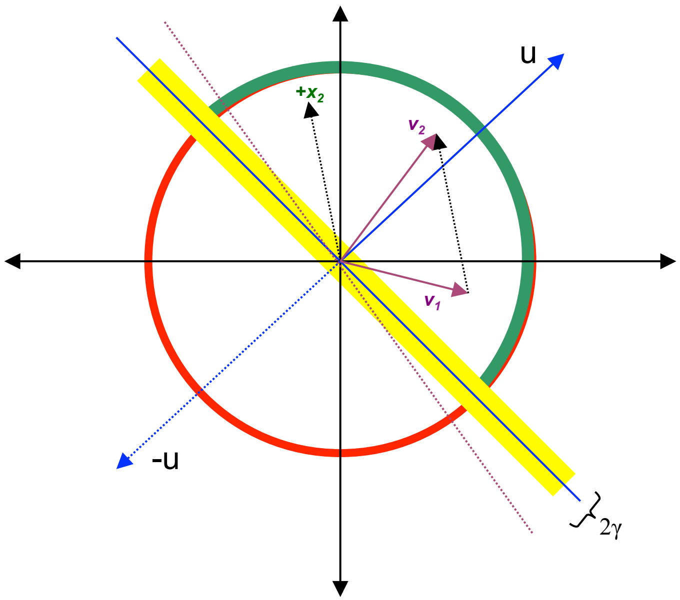


Lemma 1: the dot product between $\mathbf{v}_k$ and **u** increases with each mistake by at last γ: i.e.,

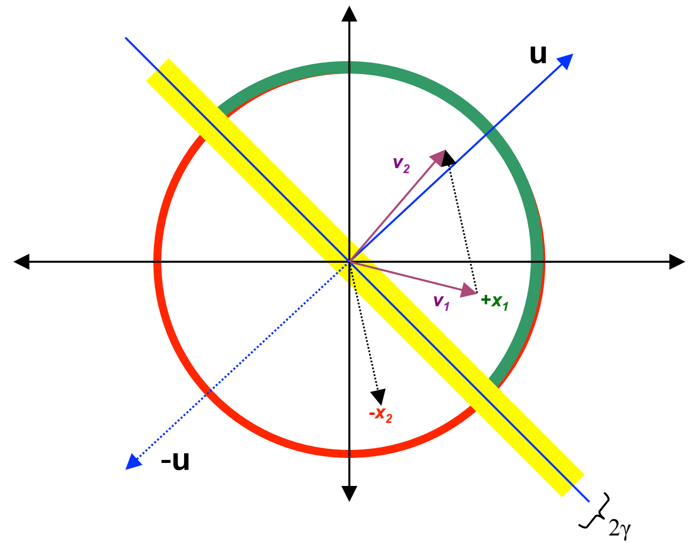
$$\forall k : \mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$$

Another observation: increasing the dot product of $\mathbf{v}_k$ with **u** (going "up") doesn't mean we're converging to **u.**

(3a) The guess $\mathbf{v_2}$ after the two positive examples: $\mathbf{v}_2=\mathbf{v}_1+\mathbf{x}_2$



(3b) The guess $\mathbf{v_2}$ after the one positive and one negative example: $\mathbf{v}_2=\mathbf{v}_1-\mathbf{x}_2$



Lemma 2: The norm of $\mathbf{v}_k$ grows slowly with each mistake, i.e.,

$$\forall k, \left\|\mathbf{v}_k\right\|_2^2 \le kR^2$$

$$\mathbf{v}_{k+1}\cdot\mathbf{v}_{k+1} = (\mathbf{v}_k + y_i\mathbf{x}_i)\cdot(\mathbf{v}_k + y_i\mathbf{x}_i)$$

$$\left\|\mathbf{v}_{k+1}\right\|_2^2 = \left\|\mathbf{v}_k\right\|_2^2 + 2y_i\mathbf{x}_i + y_i^2\left\|\mathbf{x}_i\right\|_2^2$$

$$\left\|\mathbf{v}_{k+1}\right\|_2^2 \le \left\|\mathbf{v}_k\right\|_2^2 + 1\left\|\mathbf{x}_i\right\|_2^2$$

$$\left\|\mathbf{v}_{k+1}\right\|_2^2 \le \left\|\mathbf{v}_k\right\|_2^2 + R^2$$

$$\forall \mathbf{x}_i \text{ given by A}, \left\|\mathbf{x}_i\right\|_2^2 \le R^2$$

so …

$$\left\|\mathbf{v}_k\right\|_2^2 \le kR^2$$

Lemma 1: the dot product between $\mathbf{v}_k$ and $\mathbf{u}$ increases with each mistake by at last γ: i.e.,

$$\forall k : \mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$$

Lemma 2: The norm of $\mathbf{v}_k$ grows slowly with each mistake, i.e.,

$$\forall k, \|\mathbf{v}_k\|_2^2 \leq kR^2$$

$$k\gamma \leq \mathbf{v}_k \cdot \mathbf{u} \quad \text{and} \quad \|\mathbf{v}_k\|_2^2 \leq kR^2$$

Remember that $\|\mathbf{v}\|_2^2 = \mathbf{v} \cdot \mathbf{v}$

$$k^2\gamma^2 \leq \|\mathbf{v}_k \cdot \mathbf{u}\|_2^2 \quad \text{and} \quad \|\mathbf{v}_k\|_2^2 \leq kR^2$$

$$k^2\gamma^2 \leq \|\mathbf{v}_k\|_2^2 \cdot \|\mathbf{u}\|_2^2 \quad \text{and} \quad \|\mathbf{v}_k\|_2^2 \leq kR^2$$

...and $\|\mathbf{u}\|_2 = 1$

$$k^2\gamma^2 \leq \|\mathbf{v}_k\|_2^2 \quad \text{and} \quad \|\mathbf{v}_k\|_2^2 \leq kR^2$$

$$k^2\gamma^2 \leq \|\mathbf{v}_k\|_2^2 \leq kR^2$$

$$k^2\gamma^2 \leq kR^2$$

$$k < \left(\frac{R}{\gamma}\right)^2$$

Summary

- We have shown that
  - *If* : exists a **u** with unit norm that has margin γ on examples in the seq $(\mathbf{x}_1,y_1),(\mathbf{x}_2,y_2),\ldots.$
  - *Then* : the perceptron algorithm makes $< R^2/\gamma^2$ mistakes on the sequence (where $R \geq ||\mathbf{x}_i||$)
  - *Independent* of dimension of the data or classifier (!)

- This is surprising in several ways:
  - You can bound errors in an adversarial setting
    - General case: you bound "regret", i.e., how well you do on-line vs the best fixed classifier
  - We're making claims about generalization after a few examples
    - Statistical efficiency
  - We don't care about how many features there are, only how "big" the example is.
    - Important special case: for each example, only a few features have non-zero values (*sparse* examples)

Summary

- We have shown that
  - *If* : exists a **u** with unit norm that has margin γ on examples in the seq $(\mathbf{x}_1,y_1),(\mathbf{x}_2,y_2),\ldots.$
  - *Then* : the perceptron algorithm makes $< R^2/\gamma^2$ mistakes on the sequence (where $R \geq ||\mathbf{x}_i||$)
  - *Independent* of dimension of the data or classifier (!)
- We *don't* know if this algorithm could be better
  - There are many variants that rely on similar analysis (ROMMA, Passive-Aggressive, MIRA, …)
- We *don't* know what happens if the data's not separable
  - Unless I explain the "Δ trick" to you
- We *don't* know what classifier to use "after" training

The Δ Trick

- The proof assumes the data is separable by a wide margin
- We can *make* that true by adding an “id” feature to each example
  - sort of like we added a constant feature

$$\mathbf{x}^1 = (x_1^1, x_2^1, \ldots, x_m^1) \rightarrow (x_1^1, x_2^1, \ldots, x_m^1, \overbrace{\Delta, 0, \ldots, 0}^{n \text{ new features}})$$

$$\mathbf{x}^2 = (x_1^2, x_2^2, \ldots, x_m^2) \rightarrow (x_1^2, x_2^2, \ldots, x_m^2, 0, \Delta, \ldots, 0)$$

$$\ldots$$

$$\mathbf{x}^n = (x_1^n, x_2^n, \ldots, x_m^n) \rightarrow (x_1^n, x_2^n, \ldots, x_m^n, 0, 0, \ldots, \Delta)$$

The Δ Trick

- Replace $\mathbf{x}_i$ with $\mathbf{x}'_i$ so $\mathbf{X}$ becomes $[\mathbf{X} \mid \mathbf{I}\,\Delta]$
- Replace R^2 in our bounds with $R^2 + \Delta^2$
- Let $d_i = \max(0, \gamma - y_i\, \mathbf{x}_i\, \mathbf{u})$
- Let $\mathbf{u}' = (u_1, \ldots, u_n, y_1 d_1/\Delta, \ldots\ y_m d_m/\Delta) * 1/Z$
  - So $Z = \mathrm{sqrt}(1 + D^2/\Delta^2)$, for $D = \mathrm{sqrt}(d_1^2 + \ldots + d_m^2)$
  - Now $[\mathbf{X}|\mathbf{I}\Delta]$ is separable by $\mathbf{u}'$ with margin γ
- Mistake bound is $(R^2 + \Delta^2)Z^2 / \gamma^2$
- Let $\Delta = \mathrm{sqrt}(RD) \rightarrow k <= ((R + D)/\gamma)^2$
- Conclusion: a little noise is ok

THE VOTED PERCEPTRON

On-line to batch learning

Imagine we run the on-line perceptron and see this result.

i	guess	input	result
1	$\mathbf{v}_0$	$\mathbf{x}_1$	X (a mistake)
2	$\mathbf{v}_1$	$\mathbf{x}_2$	$\surd$ (correct!)
3	$\mathbf{v}_1$	$\mathbf{x}_3$	$\surd$
4	$\mathbf{v}_1$	$\mathbf{x}_4$	X (a mistake)
5	$\mathbf{v}_2$	$\mathbf{x}_5$	$\surd$
6	$\mathbf{v}_2$	$\mathbf{x}_6$	$\surd$
7	$\mathbf{v}_2$	$\mathbf{x}_7$	$\surd$
8	$\mathbf{v}_2$	$\mathbf{x}_8$	X
9	$\mathbf{v}_3$	$\mathbf{x}_9$	$\surd$
10	$\mathbf{v}_3$	$\mathbf{x}_{10}$	X

Which $\mathbf{v}_i$ should we use?

Maybe the *last* one?

Here it's never gotten any test cases right!
(Experimentally, the classifiers move around a lot.)

Maybe the "best one"?

But we "improved" it with later mistakes…

$$
\begin{aligned}
P(\text{error in } \mathbf{x}) &= \sum_k P(\text{error on } \mathbf{x} | \text{picked } \mathbf{v}_k) P(\text{picked } \mathbf{v}_k) \\
&= \sum_k \frac{1}{m_k} \frac{m_k}{m} = \sum_k \frac{1}{m} = \frac{k}{m}
\end{aligned}
$$

Imagine we run the on-line perceptron and see this result.

i	guess	input	result
1	$\mathbf{v}_0$	$\mathbf{x}_1$	X (a mistake)
2	$\mathbf{v}_1$	$\mathbf{x}_2$	$\surd$ (correct!)
3	$\mathbf{v}_1$	$\mathbf{x}_3$	$\surd$
4	$\mathbf{v}_1$	$\mathbf{x}_4$	X (a mistake)
5	$\mathbf{v}_2$	$\mathbf{x}_5$	$\surd$
6	$\mathbf{v}_2$	$\mathbf{x}_6$	$\surd$
7	$\mathbf{v}_2$	$\mathbf{x}_7$	$\surd$
8	$\mathbf{v}_2$	$\mathbf{x}_8$	X
9	$\mathbf{v}_3$	$\mathbf{x}_9$	$\surd$
10	$\mathbf{v}_3$	$\mathbf{x}_{10}$	X

1. Pick a $\mathbf{v}_k$ at random according to m_k/m, the fraction of examples it was used for.
2. Predict using the $\mathbf{v}_k$ you just picked.

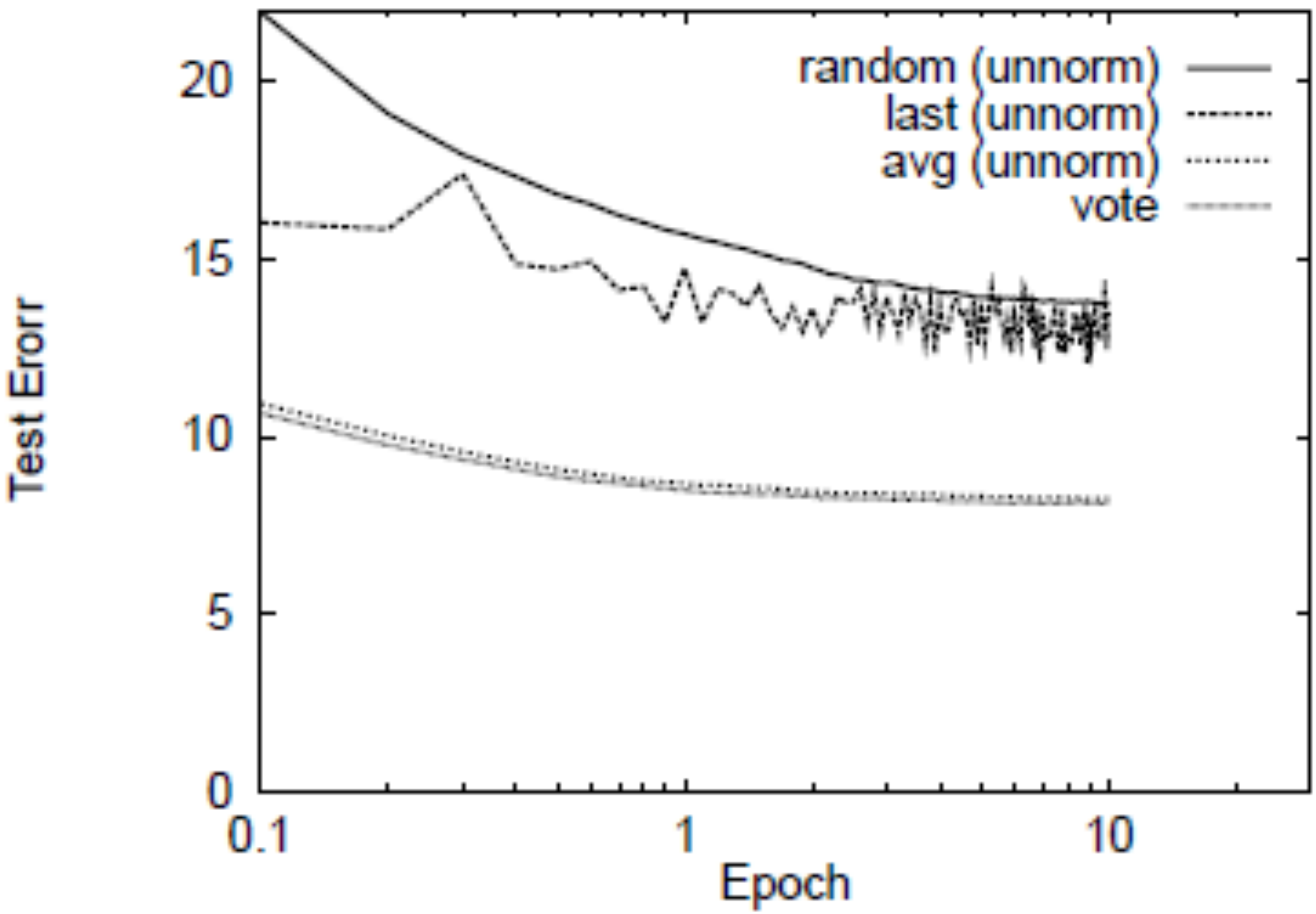
$$
\begin{aligned}
P(\text{error in }\mathbf{x}) &= \sum_k P(\text{error on }\mathbf{x}|\text{picked }\mathbf{v}_k)P(\text{picked }\mathbf{v}_k) \\
&= \sum_k \frac{1}{m_k}\frac{m_k}{m} = \sum_k \frac{1}{m} = \frac{k}{m}
\end{aligned}
$$

Imagine we run the on-line perceptron and see this result.

i	guess	input	result
1	$\mathbf{v}_0$	$\mathbf{x}_1$	X (a mistake)
2	$\mathbf{v}_1$	$\mathbf{x}_2$	$\surd$ (correct!)
3	$\mathbf{v}_1$	$\mathbf{x}_3$	$\surd$
4	$\mathbf{v}_1$	$\mathbf{x}_4$	X (a mistake)
5	$\mathbf{v}_2$	$\mathbf{x}_5$	$\surd$
6	$\mathbf{v}_2$	$\mathbf{x}_6$	$\surd$
7	$\mathbf{v}_2$	$\mathbf{x}_7$	$\surd$
8	$\mathbf{v}_2$	$\mathbf{x}_8$	X
9	$\mathbf{v}_3$	$\mathbf{x}_9$	$\surd$
10	$\mathbf{v}_3$	$\mathbf{x}_{10}$	X

1. Disadvantage: we need to keep around every **v** used in learning. This can be expensive.
2. Better: use a deterministic approximation to this*: a sum of the* $\mathbf{v}_k$*'s, weighted by* m_k/m

From Freund & Schapire, 1998: Classifying digits with VP



Breaking it down: the perceptron

- Let $\mathbf{v}_0$ be an all-zeros vector
- Let k=0
- For each "epoch" t=1,2,….T:
  - Randomly shuffle the examples -- *voting proof wants them i.i.d.*
    - For each example $\mathbf{x}_i, y_i$:
      - If $\mathbf{v}_k \,.\, \mathbf{x}_i\, y_i < 0$, then -- *a mistake was made*
        - » $\mathbf{v}_{k+1} \leftarrow \mathbf{v}_k + \mathbf{x}_i\, y_i$ -- *update the perceptron*
        - » $k \leftarrow k+1$

Breaking it down: the perceptron

- Let **v** be an all-zeros vector
- For each "epoch" t=1,2,….T:
  - Randomly shuffle the examples -- *voting proof wants them i.i.d.*
    - For each example $\mathbf{x}_i, y_i$:
      - If $\mathbf{v} . \mathbf{x}_i\ y_i < 0$, then -- *a mistake was made*
        - » $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{x}_i\ y_i$ -- *update the perceptron*

Breaking it down: the voted perceptron

- Let $\mathbf{v}_o$ be an all-zeros vector; $m_0 = 0$; k=0; m=0
- Let $\mathbf{a}$ be an all-zeros vector
- For each "epoch" t=1,2,….T:
  - Randomly shuffle the examples *-- voting proof wants them i.i.d.*
    - For each example $\mathbf{x}_i, y_i$:
      - $m \leftarrow m+1$
      - If $\mathbf{v}_k \,.\, \mathbf{x}_i\, y_i < 0$, then *-- a mistake was made*
        - » $\mathbf{a} \leftarrow \mathbf{a} + m_k\ \mathbf{v}_k$ *-- update the average*
        - » $\mathbf{v}_{k+1} \leftarrow \mathbf{v}_k + \mathbf{x}_i\, y_i$ *-- update the perceptron*
        - » $m_{k+1} \leftarrow 1$ *-- initialize the weight of k-th perceptron*
        - » $k \leftarrow k + 1$
      - Else: $m_k \leftarrow m_k + 1$ *-- upweight the k-th classifier*

- $\mathbf{a} = \mathbf{a} + m_k\ \mathbf{v}_k$
- $\mathbf{a} = \mathbf{a}\ /\ m$

ASIDE: SPARSE VECTORS

Voted perceptron and text

- One important case: *sparse* examples, where example example has only a few non-zero features.
- Example: $\mathbf{x}=(x_1,x_2,\ldots,x_n)$ represents an d-word document
  - x_i = number of occurrences of word i
    - words #1: *aaliyeh* #2:*aardvark* … #46737: *zymurgy*
  - Usually $s \ll m$
  - 2-Norm of $\mathbf{x} < d$ … so $R^2 < d^2$
  - ….Most of the x_i's are zero


BOOLE ORDERS LUNCH
NO, NO, YES, NO, NO, YES, YES, NO, NO, NO, YES...
Menu

Voted perceptron and sparse vectors

- A (Java) vector is not a good representation for this:

1	2	3	4	5	6	7	8	9	…
0	0	0	0	1	0	0	0	3	…

- Better: record only the indices and contents of the non-zero values

(5,1),(9,3),…

- This is a *sparse vector*
  - same API, different implementation
- Matlab implements sparse vectors and matrices
  - they will be much faster when your data is sparse.
- Another kind of sparsity we care about: sparse *classifiers* (most *weights* are zero)