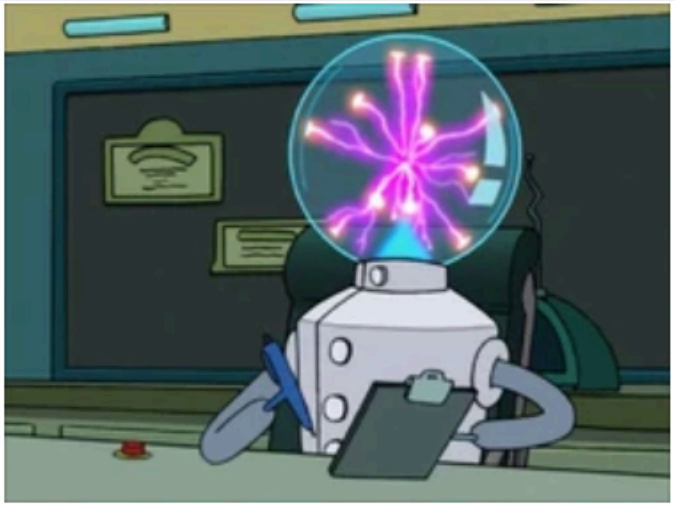
Dr. Perceptron

> *"Now, consider the following: You were admitted to this robot asylum. Therefore, you must be a robot. Diagnosis complete."*
> **—Dr. Perceptron to Fry**[source]

Dr. Perceptron is the head doctor at the Hal Institute for Criminally Insane Robots. He was destroyed briefly by Roberto during his escape from the Institute, but was apparently fixed/rebuilt and returned to work for Bender's second stay.

In 3008, Dr. Perceptron was damaged during a group therepy session, but like his encounter with Roberto, was quickly repaired to continue his duties at the Institute.

Dr. Perceptron



Gender	Male ♂
Species	Robot
Planet	Earth
Profession	Doctor of Freudian Circuit Analysis
First appearance	Insane in the Mainframe
Voiced by	Maurice LaMarche

Appearances Edit

- *Insane in the Mainframe*
- *Bender's Game*

http://futurama.wikia.com/wiki/Dr._Perceptron

Quick review of Tuesday

- Learning as optimization
- Optimizing conditional log-likelihood Pr(y|**x**) with logistic regression
- Stochastic gradient descent for logistic regression
  - Stream multiple times (epochs) thru data
  - Keep model in memory
- L2-regularization
- Sparse/lazy L2 regularization
- The “hash trick”: allow feature collisions, use array indexed by hash code instead of hash table for parameters.

Quick look ahead

- Experiments with a hash-trick implementation of logistic regression
- Next question:
  - how do you parallelize SGD, or more generally, this kind of streaming algorithm?
  - each example affects the next prediction ➔ order matters ➔ parallelization changes the behavior
  - we will step back to perceptrons and then step forward to **parallel perceptrons**

Debugging Machine Learning Algorithms

William Cohen

Debugging for non-ML systems

- "If it compiles, ship it."

Debugging for ML systems

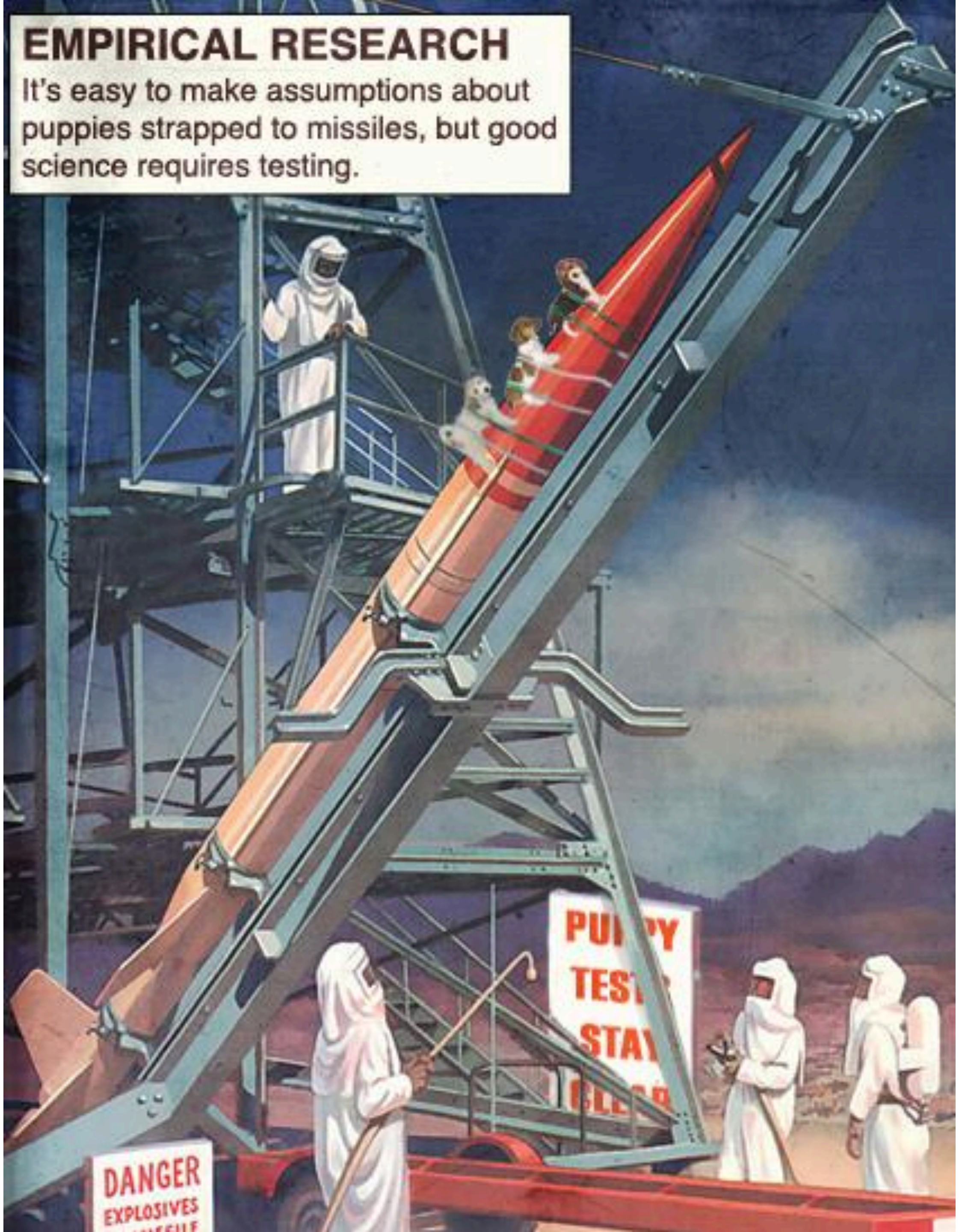
1. It's definitely *exactly* the algorithm you read about in that paper
2. It also compiles
3. It gets 87% accuracy on the author's dataset
   - but he got 91%
   - so it's not working?
   - or, your eval is wrong?
   - or, *his* eval is wrong?

Debugging for ML systems

1. It's definitely *exactly* the algorithm you read about in that paper
2. It also compiles
3. It gets 97% accuracy on the author's dataset
   - but he got 91%
   - so you have a best paper award!
   - or, maybe a bug…

Debugging for ML systems

- It's always hard to debug software
- It's *especially* hard for ML
  - a wide range of almost-correct modes for a program to be in


EMPIRICAL RESEARCH
It's easy to make assumptions about puppies strapped to missiles, but good science requires testing.
DANGER
EXPLOSIVES

Debugging advice

1. Write tests
2. For subtle problems, write tests
3. If you're still not sure why it's not working, write tests
4. If you get really stuck:
   - take a walk and come back to it in a hour
   - ask a friend
     - If s/he's also in 10-605 s/he can still help as long as no notes are taken (my rules)
   - take a break and write some tests

Debugging ML systems

Write tests

- For a generative learner, write a generator and *generate* training/test data from the *assumed* distribution
  - Eg, for NB: use one small multinomial for pos examples, another one for neg examples, and a weighted coin for the class priors.
- The learner should (usually) recover the actual parameters of the generator
  - given enough data, modulo convexity, …
- Test it on the weird cases (eg, uniform class priors, highly skewed multinomials)

Debugging ML systems

Write tests

- For a discriminative learner, similar trick…
- Also, use what you know: eg, for SGD
  - does taking one gradient step (on a sample task) lower the loss on the training data?
  - does it lower the loss *as expected?*
    - (f(x)-f(x+d))/d should approximate f'(x)
  - does regularization work *as expected*?
    - large mu ➔ smaller param values
  - record training set/test set loss
    - with and without regularization

Debugging ML systems

Compare to a “baseline” mathematically clean method vs scalable, efficient method

- lazy/sparse vs naïve regularizer
- hashed feature values vs hashtable feature values
- …

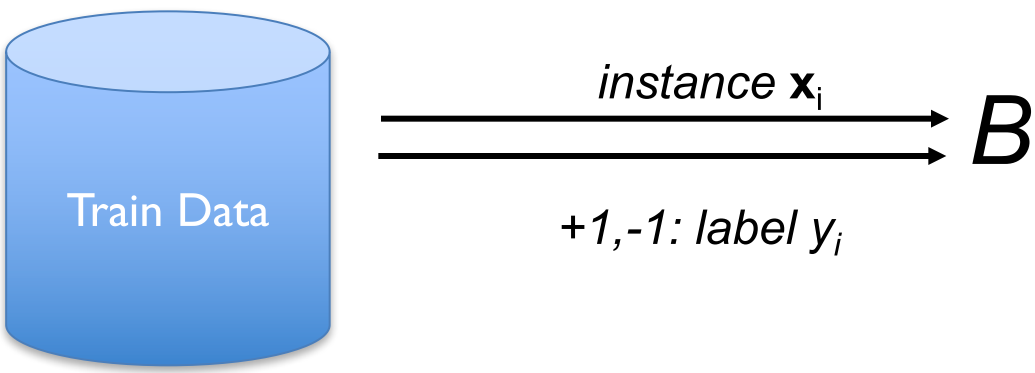
ON-LINE ANALYSIS AND REGRET

On-line learning/regret analysis

- Optimization
  - is a great model of what you **want** to do
  - a less good model of what you have **time** to do

- Example:
  - How much to we lose when we replace gradient descent with SGD?
  - what if we can only approximate the local gradient?
  - what if the distribution changes over time?
  - ...
- One powerful analytic approach: online-learning aka regret analysis (~aka on-line optimization)

On-line learning



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

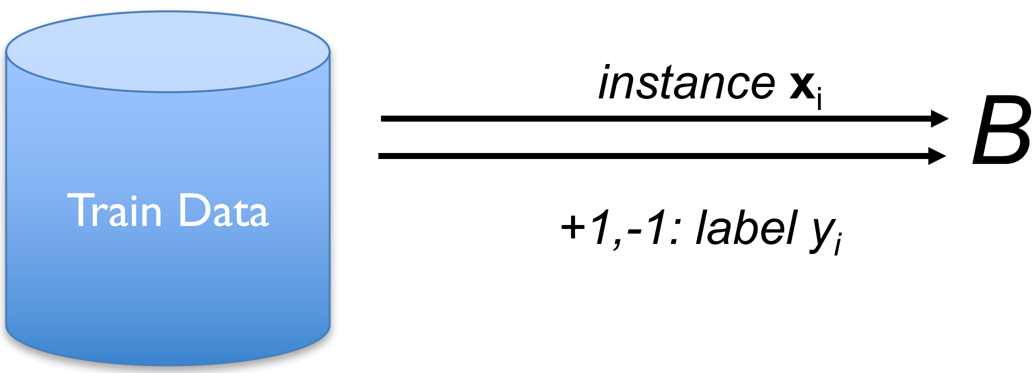
*Get y_i and make update to **v***

To detect interactions:

- increase/decrease $\mathbf{v}_k$ only if we need to (for that example)
- otherwise, leave it unchanged

- We can be sensitive to duplication by stopping updates when we get better performance

On-line learning



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

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + \textit{correction}$

To detect interactions:

- increase/decrease $\mathbf{v}_k$ only if we need to (for that example)
- otherwise, leave it unchanged

- We can be sensitive to duplication by stopping updates when we get better performance

Theory: the prediction game

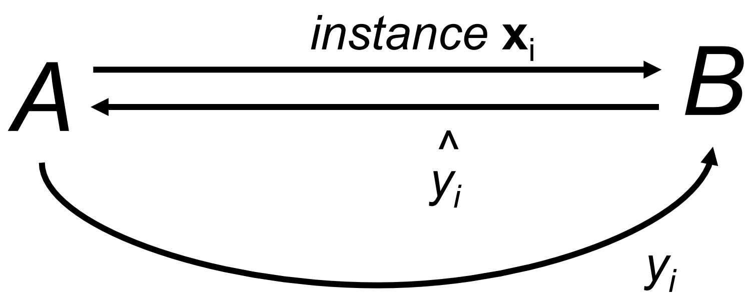
- Player A:
  - picks a “target concept” c
    - for now - from a finite set of possibilities C (e.g., all decision trees of size *m)*
  - for t=1,….,
    - Player A picks $\mathbf{x}=(x_1,\ldots,x_n)$ and sends it to B
      - For now, from a finite set of possibilities (e.g., all binary vectors of length *n)*
    - B predicts a label, $\hat{y}$, and sends it to A
    - A sends B the true label $y=c(\mathbf{x})$
    - we record if B made a *mistake* or not
  - We care about the *worst case* number of mistakes B will make over *all possible* concept & training sequences of any length
    - The “Mistake bound” for B, $M_B(C)$, is this bound

Perceptrons

The prediction game

- Are there practical algorithms where we can compute the mistake bound?

The voted perceptron



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

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

Margin γ**.** A must provide examples that can be separated with some vector $\mathbf{u}$ with margin $\gamma > 0$, ie

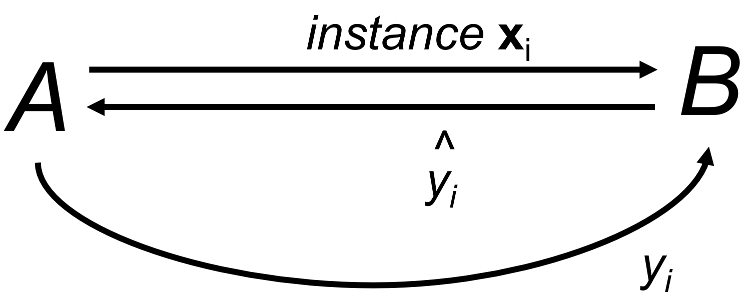
$$\exists \mathbf{u} : \forall (\mathbf{x}_i, y_i) \text{ given by } A,\ (\mathbf{u} \cdot \mathbf{x}) y_i > \gamma$$

and furthermore, $\|\mathbf{u}\| = 1$.

Radius R**.** A must provide examples "near the origin", ie

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

The voted perceptron



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

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

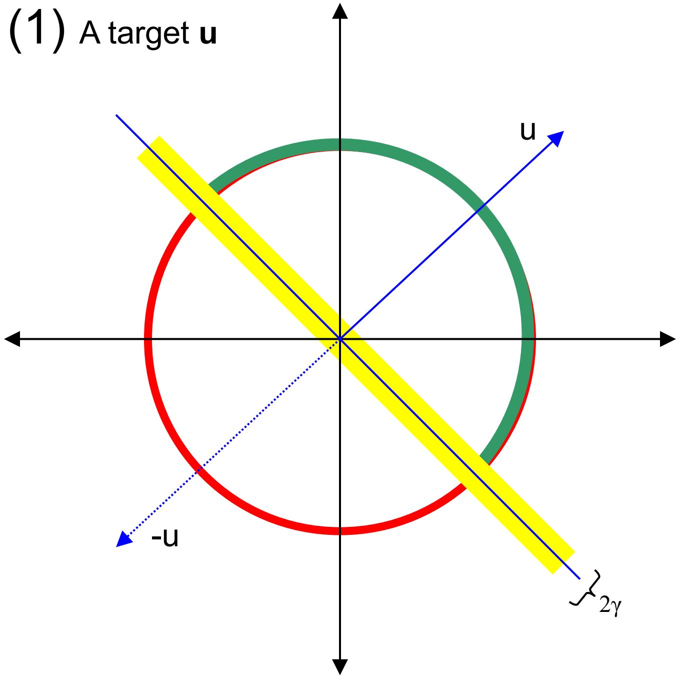
$$\boxed{\begin{array}{l} y=-1,\ p=+1\text{: } -\mathbf{x} \\ y=+1, p=-1\text{: } +\mathbf{x} \end{array}}$$

Aside: this is related to the SGD update:

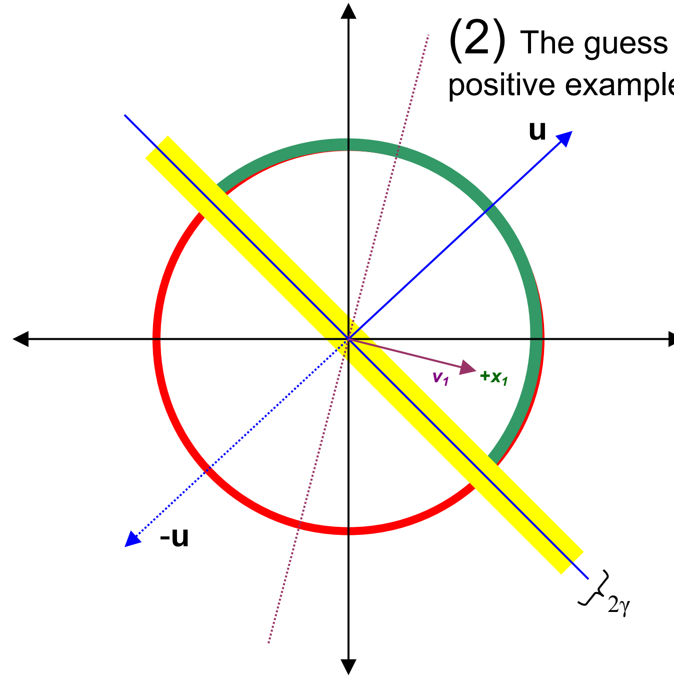
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \lambda(y - p)\mathbf{x}$$

$$\boxed{\begin{array}{l} y=p\text{: no update} \\ y=0,\ p=1\text{: } -\mathbf{x} \\ y=1, p=0\text{: } +\mathbf{x} \end{array}}$$

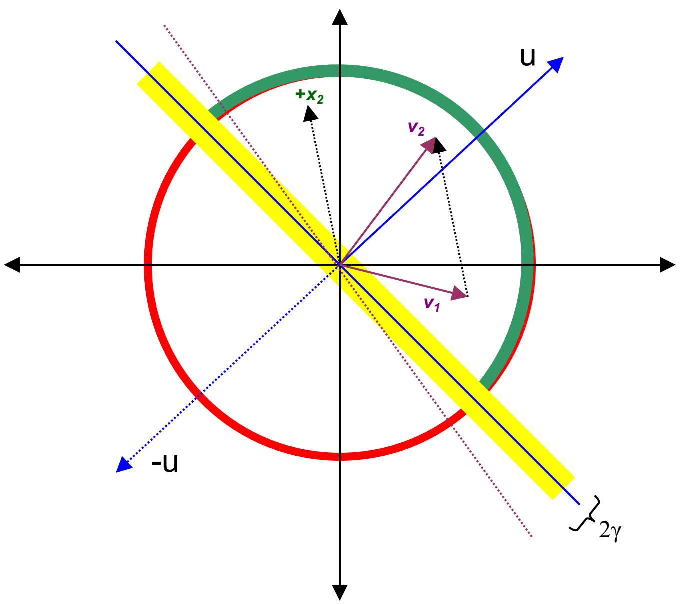
(1) A target **u**



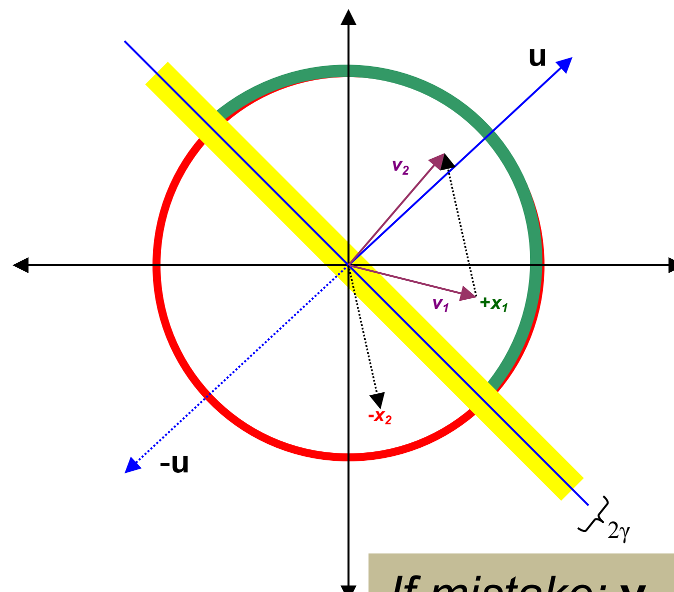
(2) The guess $\mathbf{v_1}$ after one positive example.



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

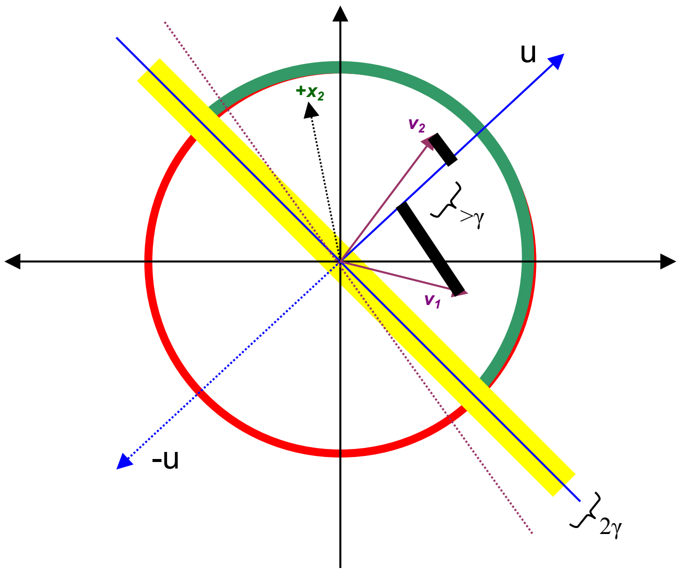


(3b) The guess $\mathbf{v_2}$ after the one positive and one negative example: $\mathbf{v}_2=\mathbf{v}_1-\mathbf{x}_2$

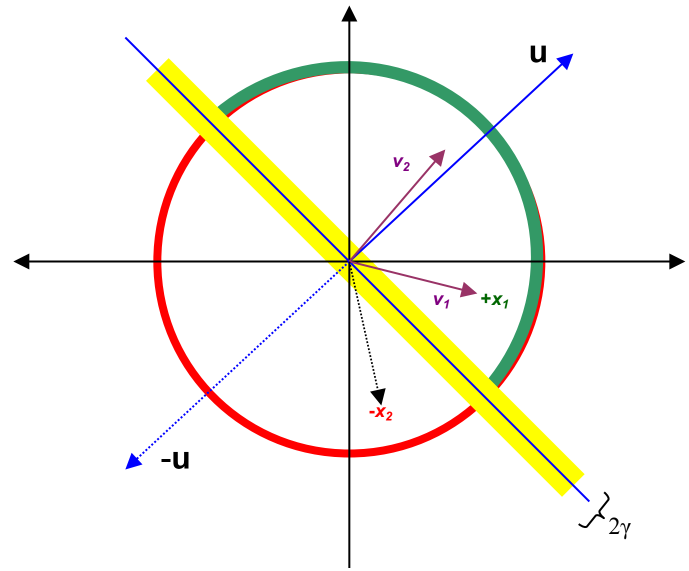


If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i\, \mathbf{x}_i$

(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: $\mathbf{v}_2=\mathbf{v}_1-\mathbf{x}_2$

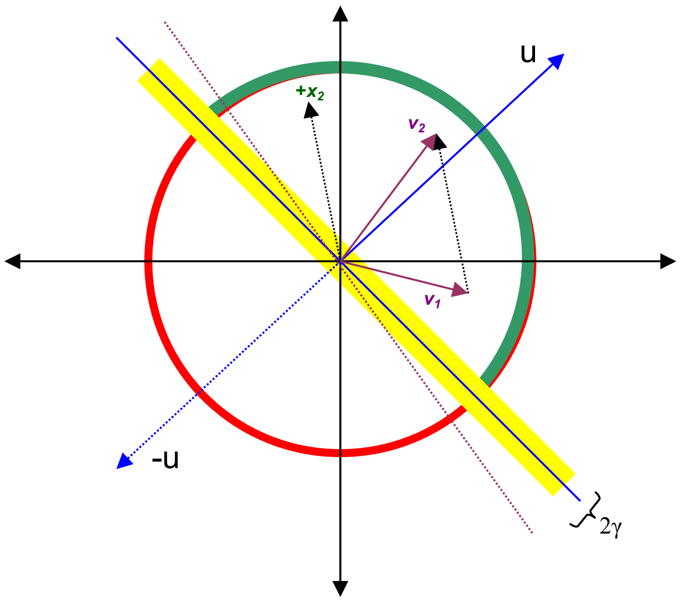


Lemma 1 $\forall k$, $\mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$. *In other words, the dot product between* $\mathbf{v}_k$ *and* $\mathbf{u}$ *increases with each mistake, at a rate depending on the margin* γ.

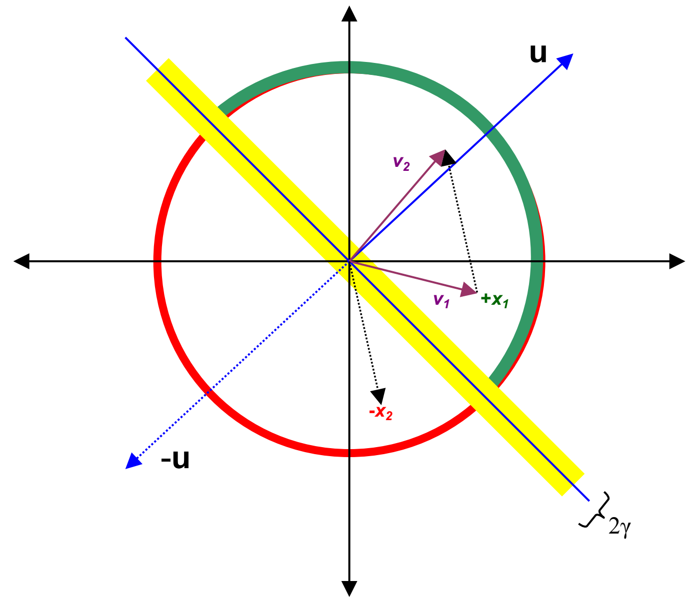
Proof:

$$
\begin{aligned}
& \mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k + y_i \mathbf{x}_i) \cdot \mathbf{u} \\
\Rightarrow \quad & \mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k \cdot \mathbf{u}) + y_i(\mathbf{x}_i \cdot \mathbf{u}) \\
\Rightarrow \quad & \mathbf{v}_{k+1} \cdot \mathbf{u} \geq \mathbf{v}_k \cdot \mathbf{u} + \gamma \\
\Rightarrow \quad & \mathbf{v}_k \cdot \mathbf{u} \geq k\gamma
\end{aligned}
$$

(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 $\forall k$, $\|\mathbf{v}_k\|^2 \leq kR^2$. *In other words, the norm of* $\mathbf{v}_k$ *grows "slowly", at a rate depending on* R^2.

Proof:

$$
\begin{aligned}
& \mathbf{v}_{k+1} \cdot \mathbf{v}_{k+1} = (\mathbf{v}_k + y_i \mathbf{x}_i) \cdot (\mathbf{v}_k + y_i \mathbf{x}_i) \\
\Rightarrow \quad & \|\mathbf{v}_{k+1}\|^2 = \|\mathbf{v}_k\|^2 + 2 y_i \mathbf{x}_i \cdot \mathbf{v}_k + y_i^2 \|\mathbf{x}_i\|^2 \\
\Rightarrow \quad & \|\mathbf{v}_{k+1}\|^2 = \|\mathbf{v}_k\|^2 + [\text{something negative}] + 1\|\mathbf{x}_i\|^2 \\
\Rightarrow \quad & \|\mathbf{v}_{k+1}\|^2 \leq \|\mathbf{v}_k\|^2 + \|\mathbf{x}\|^2 \\
\Rightarrow \quad & \|\mathbf{v}_{k+1}\|^2 \leq \|\mathbf{v}_k\|^2 + R^2 \\
\Rightarrow \quad & \|\mathbf{v}_k\|^2 \leq kR^2
\end{aligned}
$$

Lemma 1 $\forall k$, $\mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$. *In other words, the dot product between* $\mathbf{v}_k$ *and* $\mathbf{u}$ *increases with each mistake, at a rate depending on the margin* γ.

Lemma 2 $\forall k$, $\|\mathbf{v}_k\|^2 \leq kR$. *In other words, the norm of* $\mathbf{v}_k$ *grows "slowly", at a rate depending on* R.

$$
\begin{aligned}
& (k\gamma)^2 \leq (\mathbf{v}_k \cdot \mathbf{u})^2 \\
\Rightarrow \quad & k^2\gamma^2 \leq \|\mathbf{v}_k\|^2 \|\mathbf{u}\|^2 \\
\Rightarrow \quad & k^2\gamma^2 \leq \|\mathbf{v}_k\|^2
\end{aligned}
$$

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

$$
\begin{aligned}
\Rightarrow \quad & k^2\gamma^2 \leq kR^2 \\
\Rightarrow \quad & k\gamma^2 \leq R^2 \\
\Rightarrow \quad & k \leq \frac{R^2}{\gamma^2} = \left(\frac{R}{\gamma}\right)^2
\end{aligned}
$$

Radius R. A must provide examples "near the origin", ie

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

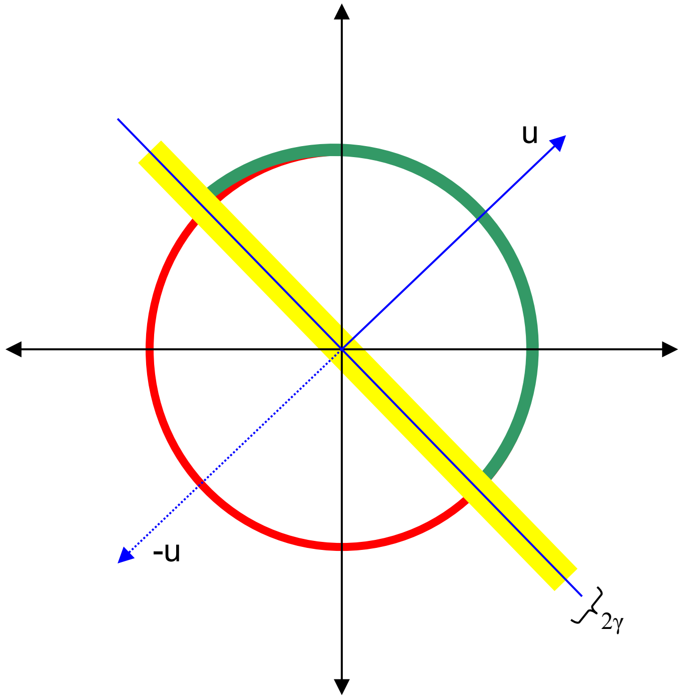
One Weird Trick for Making Perceptrons More Expressive

What if the separating line doesn't go thru the origin?

Replace $\mathbf{x} = (x^1,\ldots,x^n)$ with $(x^0,\ldots,x^n)$ where $x^0 = 1$ for every example $\mathbf{x}$.

Then $y = sign\left(\sum_j x^j w^j\right)$ becomes $sign\left(x^0 w^0 + \sum_{j\geq 1} x^j w^j\right)$ which is

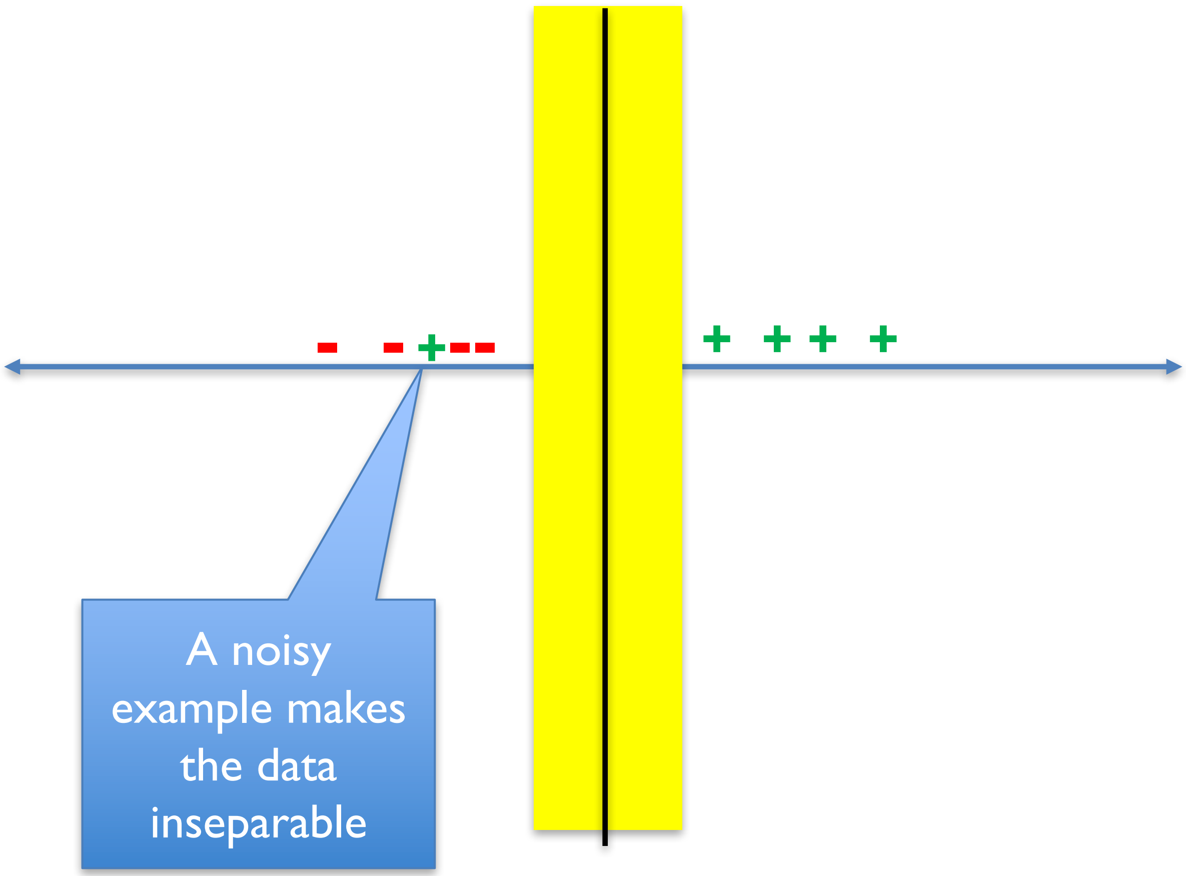
$$sign\left(w^0 + \sum_{j\geq 1} x^j w^j\right)$$



Summary

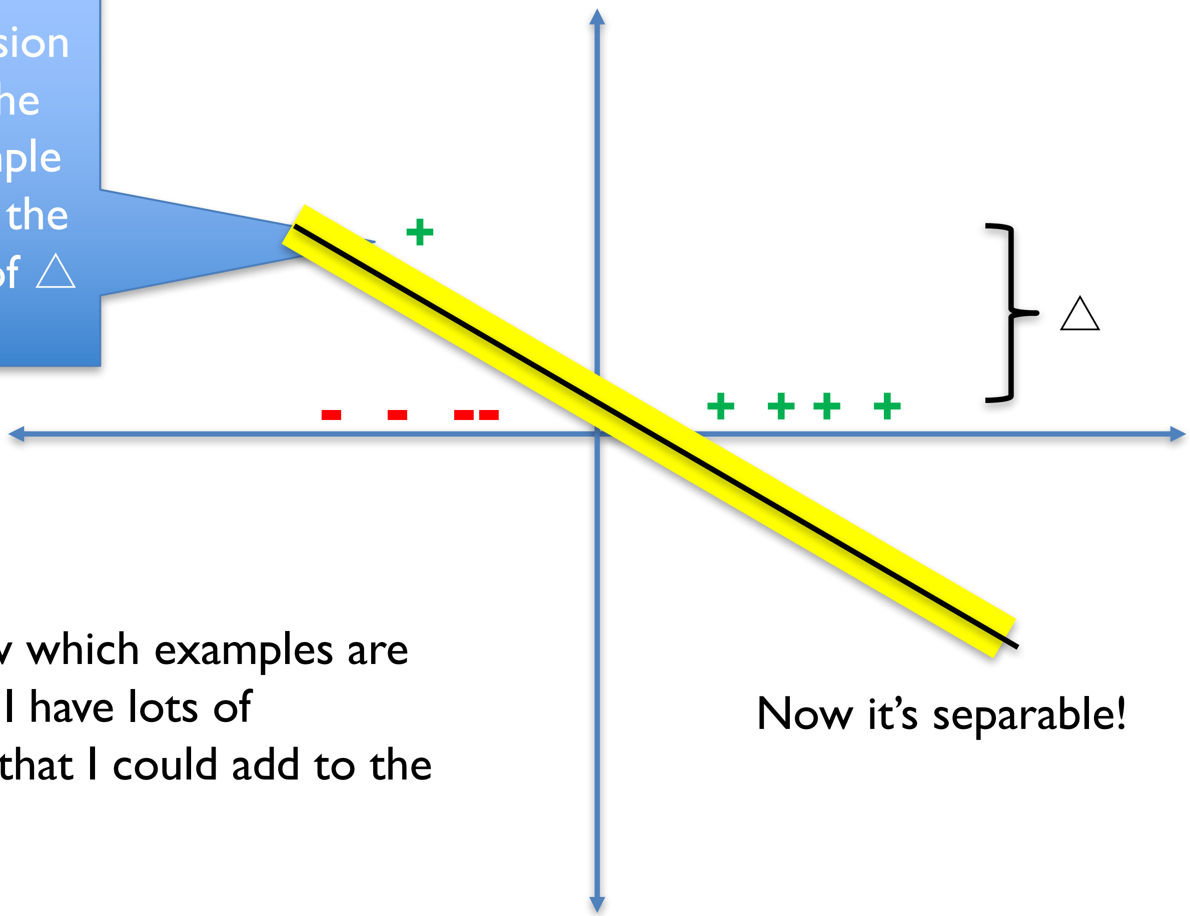
- We have shown that
  - *If* : exists a **u** with unit norm that has margin γ on examples in the seq $(\mathbf{x}_1,y_1),(\mathbf{x}_2,y_2),\ldots.$
  - *Then* : the perceptron algorithm makes $< R^2/\gamma^2$ mistakes on the sequence (where $R \geq ||\mathbf{x}_i||$)
  - *Independent* of dimension of the data or classifier (!)
  - This doesn't follow from M(C)<=VCDim(C)
- 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 idea of the "delta trick"



The idea of the "delta trick"

So let's add a new dimension and give the noisy example an offset in the dimension of △



Now it's separable!

I don't know which examples are noisy….but I have lots of dimensions that I could add to the data….

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

- 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

n new features

doc17: i, found, aardvark, today → i, found, aardvark, today, doc17
doc37: aardvarks, are, dangerous → aardvarks, are, dangerous, doc37
….

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 $\mathbf{u'} = \mathbf{(}u_1,\ldots,u_n,\ y_1 d_1/\Delta,\ \ldots\ y_m d_m/\Delta) * 1/Z$
  - 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

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 $R >= ||\mathbf{x}_i||$)
  - *Independent* of dimension of the data or classifier (!)
- 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 averaged perceptron

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

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.
3. (Actually, use some sort of deterministic approximation to this).

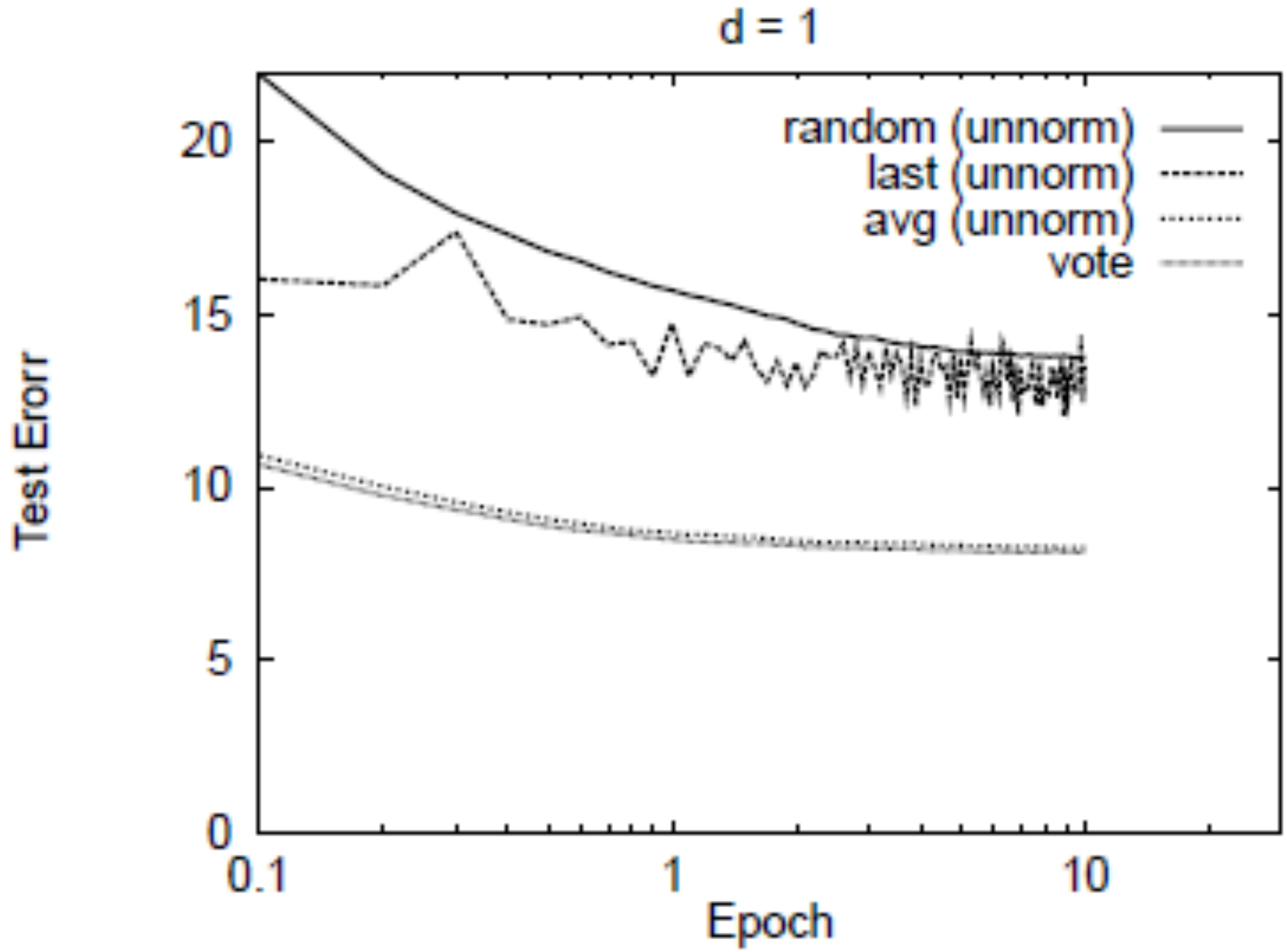
predict using sign(v*. **x**)

$$\mathbf{v}_* = \sum_k \left(\frac{m_k}{m}\mathbf{v}_k\right)$$

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.
3. (Actually, use some sort of deterministic approximation to this).

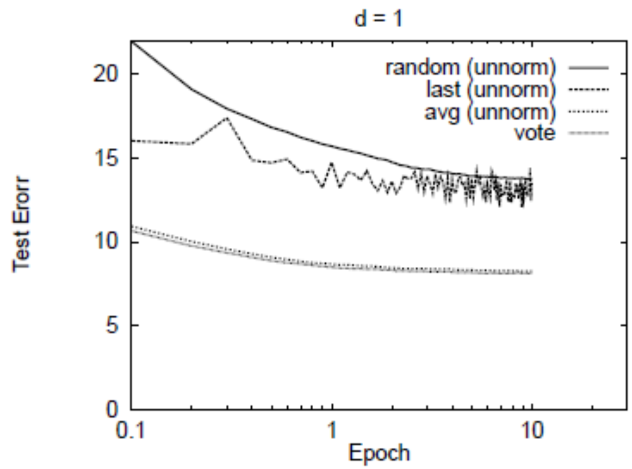

d = 1
random (unnorm)
last (unnorm)
avg (unnorm)
vote
Test Erorr
Epoch
20
15
10
5
0
0.1
1
10

SPARSIFYING THE AVERAGED PERCEPTRON UPDATE

Complexity of perceptron learning

- Algorithm: O(n)
- **v**=0
    - init hashtable
- for each example **x**,*y*:
  - if sign(**v.x**) != *y*
    - **v** = **v** + *y***x** O(|**x**|)=O(|d|)
      - for x_i!=0, v_i += $y x_i$

Final hypothesis (last): **v**



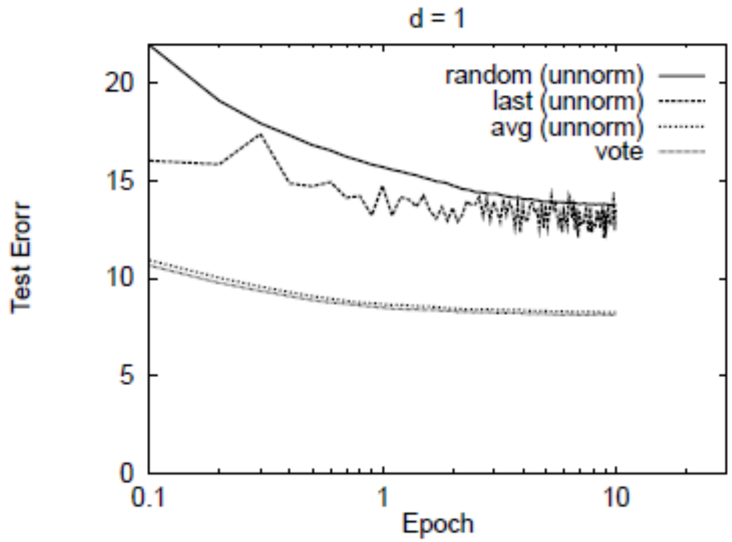
Complexity of *averaged* perceptron

- Algorithm: ~~O(n)~~ O(n|V|)
- **vk=0**
- **va = 0**
- for each example **x**,*y:*
  - if sign(**vk.x**) != *y*
    - **va = va +** mk***vk**
    - **vk** = **vk** + *y***x**
    - m = m+1
    - mk = 1
  - else
    - mk++

- init hashtables

O(|V|)

O(|**x**|)=O(|d|)



Final hypothesis (avg): **va**/m

Complexity of perceptron learning

- Algorithm: O(n)
- **v=0**
- for each example **x**,*y*:
  - if sign(**v.x**) != *y*
    - **v** = **v** + *y***x** O(|**x**|)=O(|d|)

- init hashtable
- for x_i!=0, v_i += $y x_i$

Complexity of *averaged* perceptron

- Algorithm: ~~O(n)~~ O(n|V|)
- **vk=0**
- **va = 0**
- for each example **x**,*y:*
  - if sign(**vk.x**) != *y* O(|V|)
    - **va = va + vk**
    - **vk** = **vk** + *y***x**
    - mk = 1 O(|**x**|)=O(|d|)
  - else
    - nk++

- init hashtables
- for vk_i!=0, va_i += vk_i
- for x_i!=0, v_i += yx_i

Alternative averaged perceptron

- Algorithm:
- **vk** = 0
- **va** = 0
- for each example **x**,*y:*
  - **va** = **va** + **vk**
  - m = m+1
  - if sign(**vk.x**) != *y*
    - **vk** = **vk** + *y**x
- Return **va**/m

Observe:

$$\mathbf{vk} = \sum_{j \in S_k} y_j \mathbf{x}_j$$

S_k is the set of examples including the first k mistakes

Alternative averaged perceptron

- Algorithm:
- $\mathbf{vk} = 0$
- $\mathbf{va} = 0$
- for each example $\mathbf{x}, y$:
  - $\mathbf{va} = \mathbf{va} + \sum_{j \in S_k} y_j \mathbf{x}_j$
  - $m = m+1$
  - if sign($\mathbf{vk}.\mathbf{x}$) != y
    - $\mathbf{vk} = \mathbf{vk} + y^*\mathrm{x}$
- Return $\mathbf{va}/m$

So when there's a mistake at time t on $\mathbf{x}, y$:

$y^*\mathbf{x}$ is added to **va** on *every subsequent iteration*

Suppose you know T, the total number of examples in the stream…

Alternative averaged perceptron

- Algorithm:
- **vk** = 0
- **va** = 0
- for each example **x**,*y:*
  - ~~**va** = **va** +~~ $\cancel{\sum_{j \in S_k} y_j \mathbf{x}_j}$
  - m = m+1
  - if sign(**vk.x**) != *y*
    - **vk** = **vk** + *y**x
    - **va** = **va** + (T-m)**y****x**
- Return **va**/T

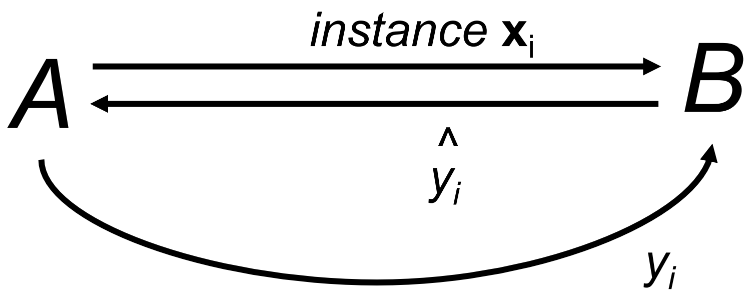
T = the total number of examples in the stream…(all epochs)

All subsequent additions of **x** to **va**

Unpublished? I figured this out recently, Leon Bottou knows it too

KERNELS AND PERCEPTRONS

The kernel perceptron



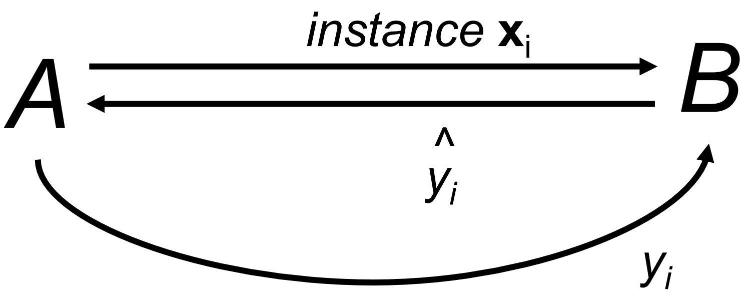
Compute: $\hat{y}_i = \mathbf{v}_k \cdot \mathbf{x}_i$ $\longrightarrow$ $$\text{Compute}: \hat{y} = \sum_{\mathbf{x}_{k^+} \in FN} \mathbf{x}_i . \mathbf{x}_{k^+} - \sum_{\mathbf{x}_{k^-} \in FP} \mathbf{x}_i . \mathbf{x}_{k^-}$$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i\, \mathbf{x}_i$ $\longrightarrow$ If false positive (too high) mistake : add $\mathbf{x}_i$ to FP

If false positive (too low) mistake : add $\mathbf{x}_i$ to FN

Mathematically the same as before … but allows use of the kernel trick

The kernel perceptron



$$\boxed{K(\mathbf{x},\mathbf{x}_k) \equiv \mathbf{x}\cdot\mathbf{x}_k}$$

$$\boxed{\hat{y} = \sum_{\mathbf{x}_{k^+}\in FN} K(\mathbf{x}_i,\mathbf{x}_{k^+}) - \sum_{\mathbf{x}_{k^-}\in FP} K(\mathbf{x}_i,\mathbf{x}_{k^-})}$$

Compute: $\hat{y}_i = \mathbf{v}_k \,.\, \mathbf{x}_i$ $\longrightarrow$ $\cancel{\text{Compute}: \hat{y} = \sum_{\mathbf{x}_{k^+}\in FN} \mathbf{x}_i.\mathbf{x}_{k^+} - \sum_{\mathbf{x}_{k^-}\in FP} \mathbf{x}_i.\mathbf{x}_{k^-}}$

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

If false positive (too high) mistake : add $\mathbf{x}_i$ to FP

If false positive (too low) mistake : add $\mathbf{x}_i$ to FN

Mathematically the same as before … but allows use of the "kernel trick"

Other kernel methods (SVM, Gaussian processes) aren't constrained to limited set (+1/-1/0) of weights on the K(**x**,**v**) values.

Some common kernels

- Linear kernel:

$$\boxed{K(\mathbf{x},\mathbf{x}') \equiv \mathbf{x}\cdot\mathbf{x}'}$$

- Polynomial kernel:

$$\boxed{K(\mathbf{x},\mathbf{x}') \equiv (\mathbf{x}\cdot\mathbf{x}'+1)^d}$$

- Gaussian kernel:

$$\boxed{K(\mathbf{x},\mathbf{x}') \equiv e^{-\|\mathbf{x}-\mathbf{x}'\|^2/\sigma}}$$

- More later….

Kernels 101

- Duality
  - and computational properties
  - Reproducing Kernel Hilbert Space (RKHS)
- Gram matrix
- Positive semi-definite
- Closure properties

Explicitly map from **x** to ϕ(**x**) – i.e. to the point corresponding to **x** in the *Hilbert space*

Kernels 101

Implicitly map from **x** to ϕ(**x**) by changing the kernel function K

- Duality: two ways to look at this

$$\boxed{\hat{y} = \mathbf{x} \cdot \mathbf{w} = K(\mathbf{x}, \mathbf{w})}$$

$$\boxed{\mathbf{w} = \sum_{\mathbf{x}_{k^+} \in FN} \mathbf{x}_{k^+} - \sum_{\mathbf{x}_{k^-} \in FP} \mathbf{x}_{k^-}}$$

$$\boxed{\hat{y} = \sum_{\mathbf{x}_{k^+} \in FN} K(\mathbf{x}_i, \mathbf{x}_{k^+}) - \sum_{\mathbf{x}_{k^-} \in FP} K(\mathbf{x}_i, \mathbf{x}_{k^-})}$$

$$\boxed{K(\mathbf{x}, \mathbf{x}_k) \equiv \phi(\mathbf{x}) \cdot \phi(\mathbf{x}_k)}$$

$$\boxed{\hat{y} = \phi(\mathbf{x}) \cdot \mathbf{w}}$$

$$\boxed{\mathbf{w} = \sum_{\mathbf{x}_{k^+} \in FN} \phi(\mathbf{x}_{k^+}) - \sum_{\mathbf{x}_{k^-} \in FP} \phi(\mathbf{x}_{k^-})}$$

$$\boxed{\hat{y} = \sum_{\mathbf{x}_{k^+} \in FN} K(\mathbf{x}_i, \mathbf{x}_{k^+}) - \sum_{\mathbf{x}_{k^-} \in FP} K(\mathbf{x}_i, \mathbf{x}_{k^-})}$$

$$\boxed{K(\mathbf{x}, \mathbf{x}_k) \equiv \phi(\mathbf{x}') \cdot \phi(\mathbf{x}'_k)}$$

Two different computational ways of getting the same behavior

Kernels 101

$K(x,x') = K(x',x)$ ➔ Gram matrix is *symmetric*

$K(x,x) > 0$ ➔ diagonal of K is positive ➔ K is "positive semi-definite" ➔ $\mathbf{z}^T \mathbf{K} \mathbf{z} >= 0$ for all $\mathbf{z}$

- Duality
- Gram matrix: $\mathbf{K}$: $k_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$

K=

K(1,1)	K(1,2)	K(1,3)	…	K(1,m)
K(2,1)	K(2,2)	K(2,3)	…	K(2,m)
…	…	…	…	…
K(m,1)	K(m,2)	K(m,3)	…	K(m,m)

Review: the hash trick

Learning as optimization for regularized logistic regression

- Algorithm: $w^j = w^j + \lambda(y - p)x^j - \lambda 2\mu w^j$
- Initialize hashtables *W, A* and set $k=0$
- For each iteration t=1,…T
  - For each example $(\mathbf{x}_i, y_i)$
    - $p_i = \ldots$; $k{+}{+}$
    - For each feature j: $x_i^j > 0$:
      - » $W[j]$ *= $(1 - \lambda 2\mu)^{k-A[j]}$
      - » $W[j] = W[j] + \lambda(y_i - p^i)x_j$
      - » $A[j] = k$

Learning as optimization for regularized logistic regression

- Algorithm: $$w^j = w^j + \lambda(y-p)x^j - \lambda 2\mu w^j$$
- Initialize arrays *W, A* of size *R* and set *k=0*
- For each iteration t=1,…T
  - For each example ($\mathbf{x}_i, y_i$)
    - Let V be hash table so that $$V[h] = \sum_{j:hash(x_i^j)\%R=h} x_i^j$$
    - $p_i = \ldots$; $k{+}{+}$
    - For each hash value *h: V[h]*>0:
      - » $W[h] \; {*}{=} \; (1 - \lambda 2\mu)^{k-A[j]}$
      - » $W[h] = W[h] + \lambda(y_i - p^i)V[h]$
      - » $A[h] = k$

The hash trick as a kernel

Hash Kernels

Qinfeng Shi, James Petterson
Australian National University and NICTA,
Canberra, Australia

Gideon Dror
Department of Computer Science
Academic College of Tel-Aviv-Yaffo, Israel

John Langford, Alex Smola, Alex Strehl
Yahoo! Research
New York, NY and Santa Clara, CA, USA

Vishy Vishwanathan
Department of Statistics
Purdue University, IN, USA

Some details

Slightly different hash to avoid systematic bias

$$V[h] = \sum_{j:hash(j)\%R==h} x_i^j$$

$$\varphi[h] = \sum_{j:hash(j)\%m==h} \xi(j)x_i^j, \quad \text{where } \xi(j) \in \{-1,+1\}$$

m is the number of buckets you hash into (R in my discussion)

Some details

Slightly different hash to avoid systematic bias

$$\varphi[h] = \sum_{j:hash(j)\%m==h} \xi(j) x_i^j, \quad \text{where } \xi(j) \in \{-1,+1\}$$

Lemma 2 *The hash kernel is unbiased, that is* $\mathbf{E}_\phi[\langle x, x' \rangle_\phi] = \langle x, x' \rangle$. *Moreover, the variance is* $\sigma^2_{x,x'} = \frac{1}{m}\left(\sum_{i \neq j} x_i^2 {x'_j}^2 + x_i x'_i x_j x'_j\right)$, *and thus, for* $\|x\|_2 = \|x'\|_2 = 1$, $\sigma^2_{x,x'} = O\left(\frac{1}{m}\right)$.

m is the number of buckets you hash into (R in my discussion)

Some details

Theorem 3 *Let $\epsilon < 1$ be a fixed constant and x be a given instance. Let $\eta = \frac{\|x\|_\infty}{\|x\|_2}$. Under the assumptions above, the hash kernel satisfies the following inequality*

$$\Pr\left\{\frac{\left|\,\|x\|_\phi^2 - \|x\|_2^2\,\right|}{\|x\|_2^2} \geq \sqrt{2}\sigma_{x,x} + \epsilon\right\} \leq \exp\left(-\frac{\sqrt{\epsilon}}{4\eta}\right).$$

I.e. – a hashed vector is probably close to the original vector

Some details

Corollary 4 *For two vectors x and x', let us define*

$$\sigma := \max(\sigma_{x,x}, \sigma_{x',x'}, \sigma_{x-x',x-x'})$$
$$\eta := \min\left(\frac{\|x\|_\infty}{\|x\|_2}, \frac{\|x'\|_\infty}{\|x'\|_2}, \frac{\|x-x'\|_\infty}{\|x-x'\|_2}\right).$$

Also let $\Delta = \|x\|^2 + \|x'\|^2 + \|x - x'\|^2$. Under the assumptions above, we have that

$$\Pr\left[|\langle x, x'\rangle_\phi - \langle x, x'\rangle| > (\sqrt{2}\sigma + \epsilon)\Delta/2\right] < 3e^{-\frac{\sqrt{\epsilon}}{4\eta}}.$$

I.e. the inner products between x and x' are probably not changed too much by the hash function: a classifier will probably still work

Some details

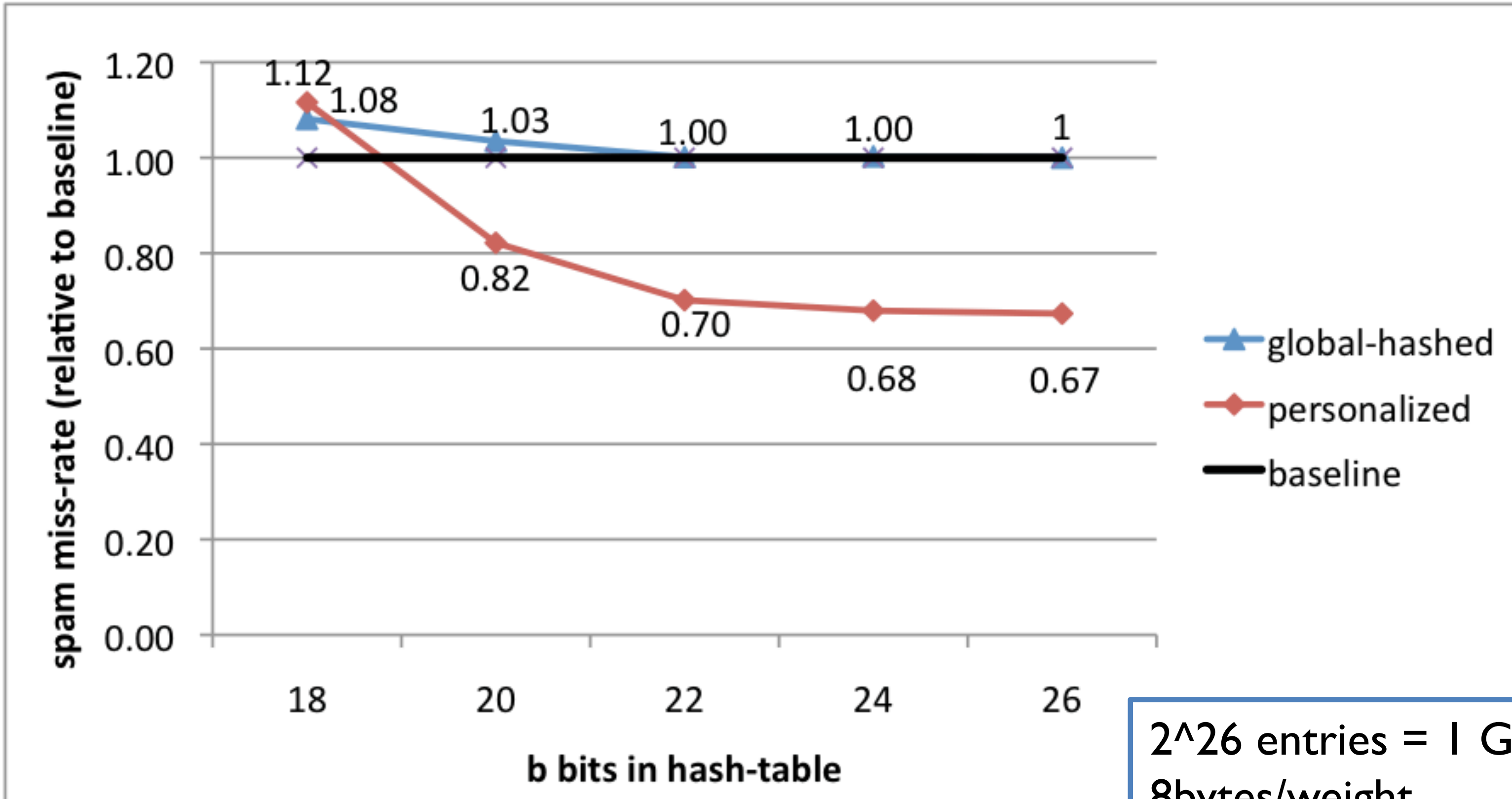
Corollary 5 *Denote by* $X = \{x_1, \ldots, x_n\}$ *a set of vectors which satisfy* $\|x_i - x_j\|_\infty \leq \eta \|x_i - x_j\|_2$ *for all pairs* i, j. *In this case with probability* $1 - \delta$ *we have for all* i, j

$$\frac{\left| \|x_i - x_j\|_\phi^2 - \|x_i - x_j\|_2^2 \right|}{\|x_i - x_j\|_2^2} \leq \sqrt{\frac{2}{m} + 64\eta^2 \log^2 \frac{n}{2\delta}}.$$

This means that the number of observations n (or correspondingly the size of the un-hashed kernel matrix) only enters *logarithmically* in the analysis.

The hash kernel: implementation

- One problem: debugging is harder
  - Features are no longer meaningful
  - There's a new way to ruin a classifier
    - Change the hash function ☹
- You can separately compute the set of all words that hash to h and guess what features mean
  - Build an inverted index $h \rightarrow w1, w2, \ldots,$



2^26 entries = 1 Gb @ 8bytes/weight

Figure 2. The decrease of uncaught spam over the baseline classifier averaged over all users. The classification threshold was chosen to keep the not-spam misclassification fixed at 1%. The hashed global classifier (*global-hashed*) converges relatively soon, showing that the distortion error ϵ_d vanishes. The personalized classifier results in an average improvement of up to 30%.