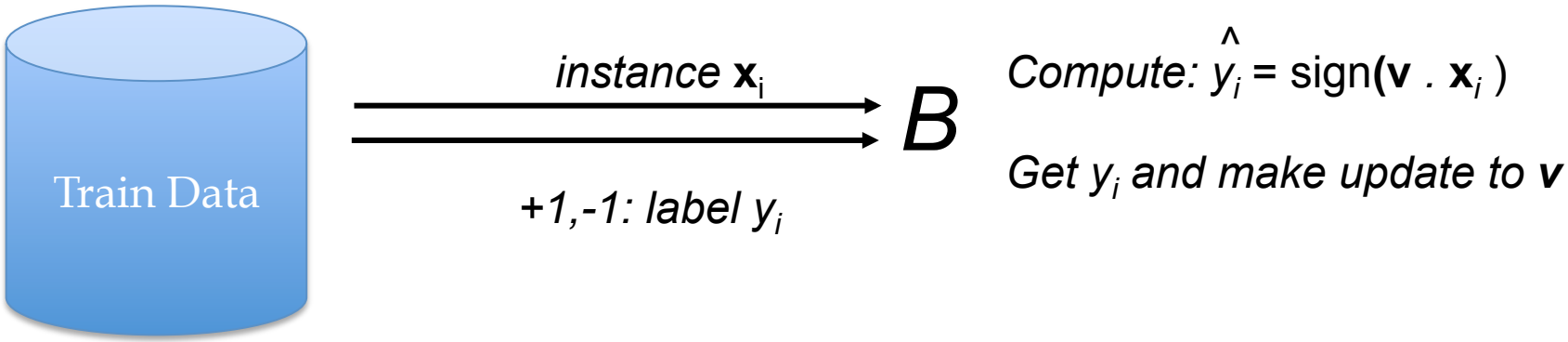
Perceptrons

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

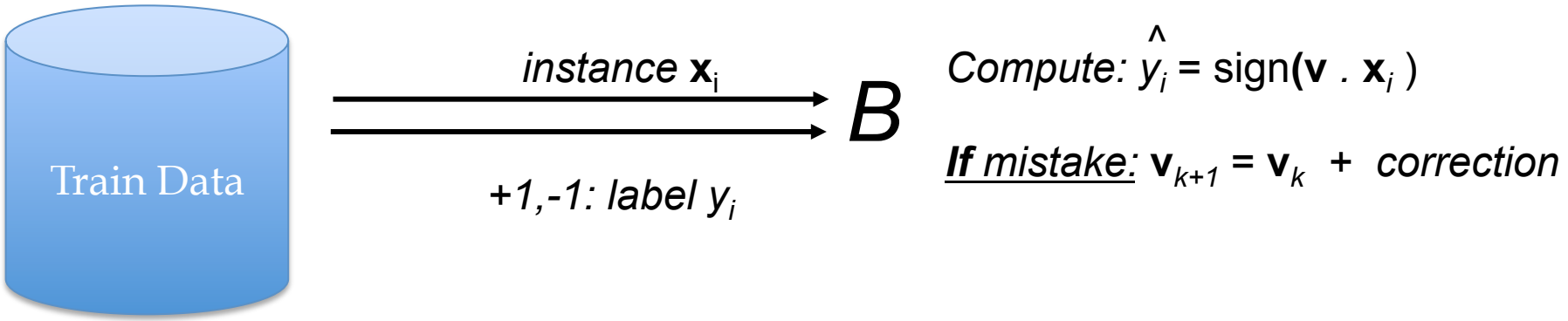


To detect interactions:

- 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



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

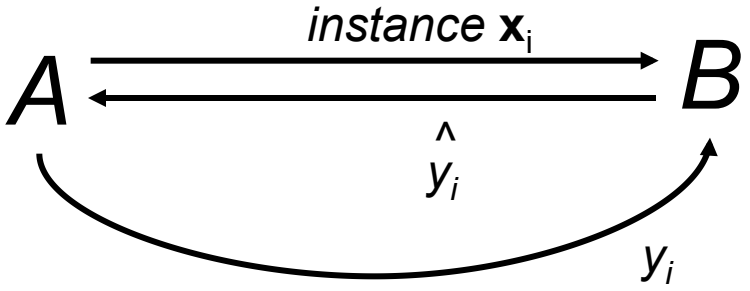
Theory: the prediction game

- 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, $\hat{\mathbf{y}}$, 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

The prediction game

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

The voted perceptron



Compute: $\hat{y}_i = \mathbf{v}_k \cdot \mathbf{x}_i$

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

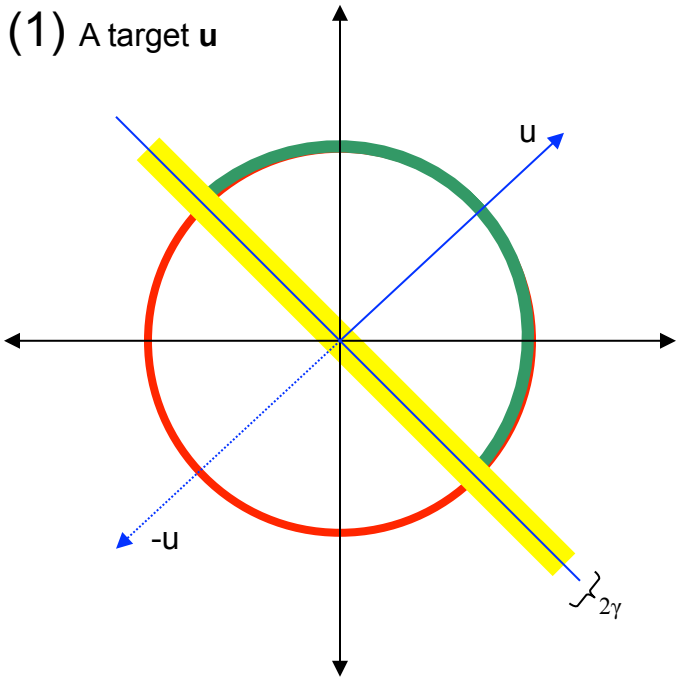
$$\exists \mathbf{u} : \forall (\mathbf{x}_i, y_i) \text{ given by } A, (\mathbf{u} \cdot \mathbf{x}) y_i > \gamma$$

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

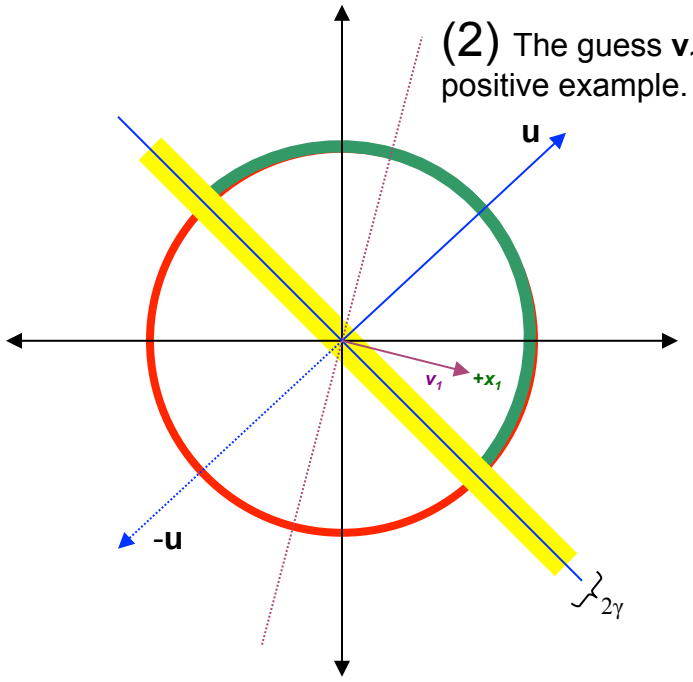
Radius R. A must provide examples "near the origin", ie

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

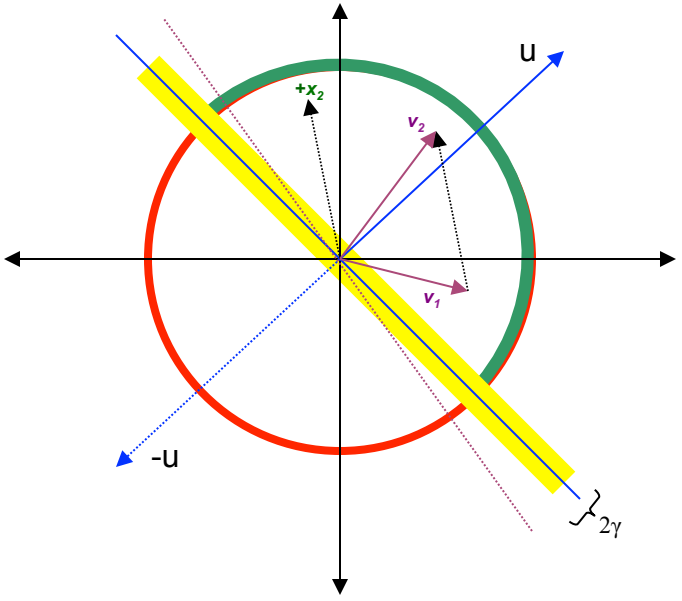
(1) A target **u**



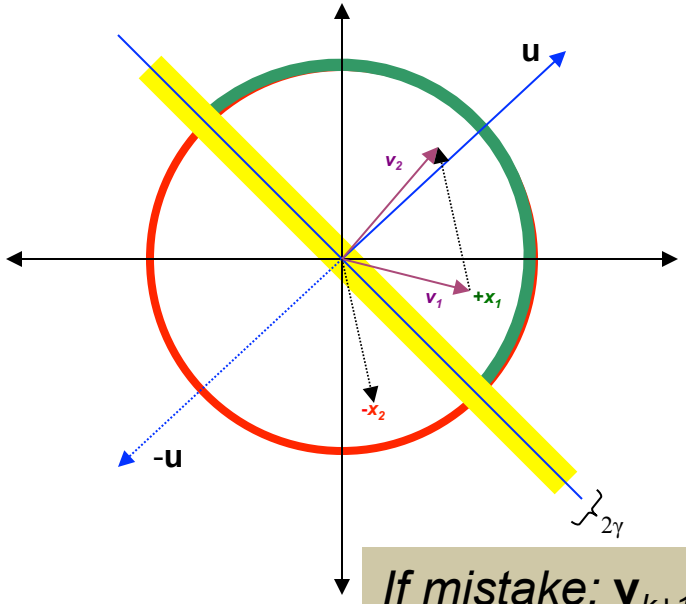
(2) The guess $\mathbf{v}_1$ after one positive example.



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

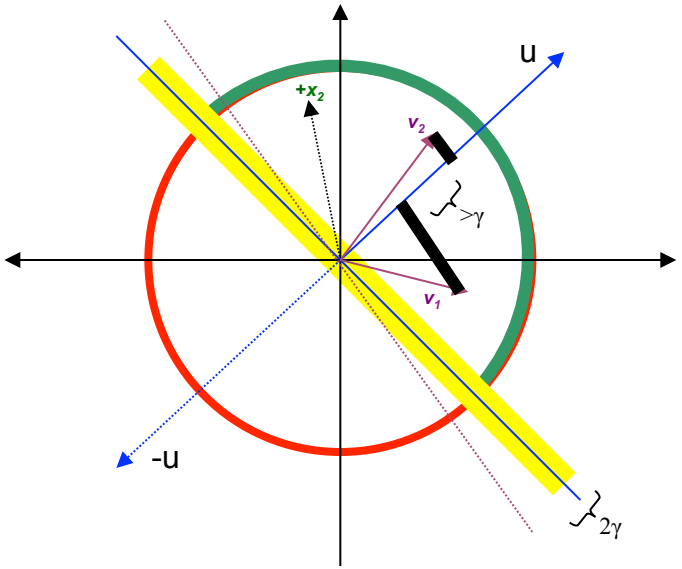


(3b) The guess $\mathbf{v}_2$ after the one positive and one negative example: $\mathbf{v}_2=\mathbf{v}_1-\mathbf{x}_2$

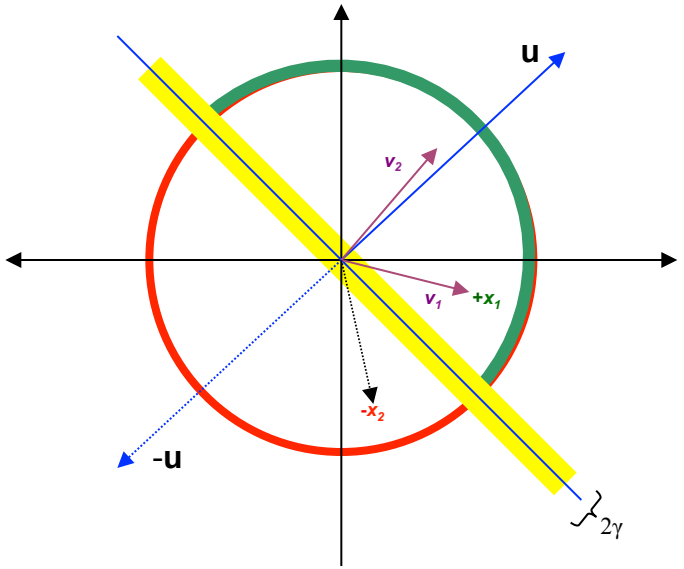


If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$

(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: $\mathbf{v_2}=\mathbf{v_1}-\mathbf{x_2}$

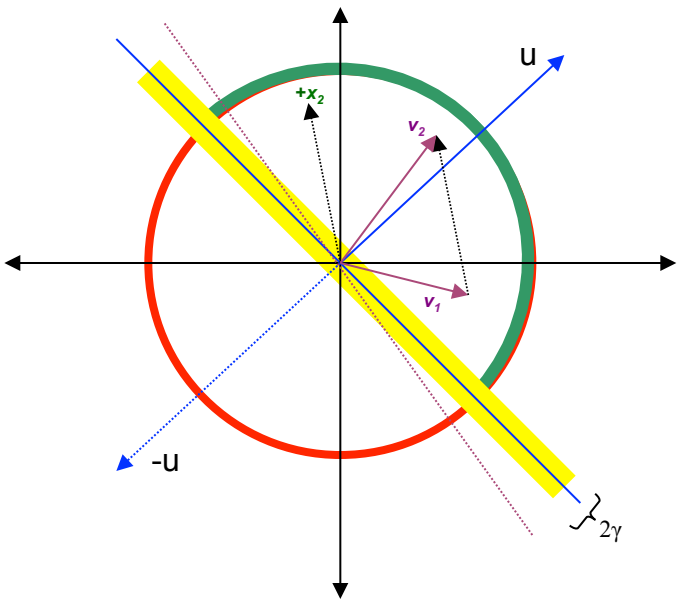


Lemma 1 $\forall k$, $\mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$. *In other words, the dot product between* $\mathbf{v}_k$ *and* $\mathbf{u}$ *increases with each mistake, at a rate depending on the margin* γ.

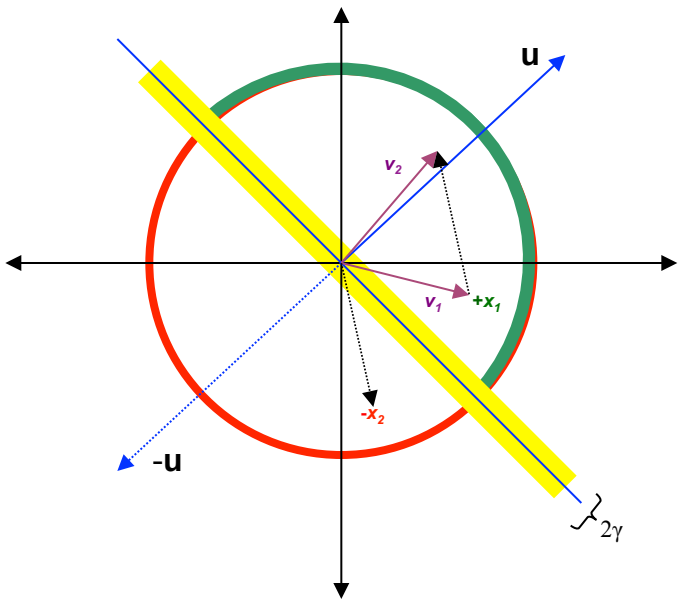
Proof:

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

(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:

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

$$\Rightarrow \quad \|\mathbf{v}_{k+1}\|^2 = \|\mathbf{v}_k\|^2 + 2y_i\mathbf{x}_i \cdot \mathbf{v}_k + y_i^2\|\mathbf{x}_i\|^2$$

$$\Rightarrow \quad \|\mathbf{v}_{k+1}\|^2 = \|\mathbf{v}_k\|^2 + [\text{something negative}] + 1\|\mathbf{x}_i\|^2$$

$$\Rightarrow \quad \|\mathbf{v}_{k+1}\|^2 \leq \|\mathbf{v}_k\|^2 + \|\mathbf{x}\|^2$$

$$\Rightarrow \quad \|\mathbf{v}_{k+1}\|^2 \leq \|\mathbf{v}_k\|^2 + R^2$$

$$\Rightarrow \quad \|\mathbf{v}_k\|^2 \leq kR^2$$

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.*

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

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

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

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

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 (!)
  - 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 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

$$\mathbf{x}^1 = (x_1^1, x_2^1, \ldots, x_m^1) \rightarrow (x_1^1, x_2^1, \ldots, x_m^1, \Delta, 0, \ldots, 0)$$

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

$\ldots$

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

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 $\mathbf{u'} = (u_1,\ldots,u_n, y_1d_1/\Delta, \ldots\ y_md_m/\Delta) * 1/Z$
  - So $Z=\text{sqrt}(1 + D^2/\Delta^2)$, for $D=\text{sqrt}(d_1^2+\ldots+d_m^2)$
  - 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 $\Delta = \text{sqrt}(RD)$ ➔ $k <= ((R + D)/ \gamma)^2$
- 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

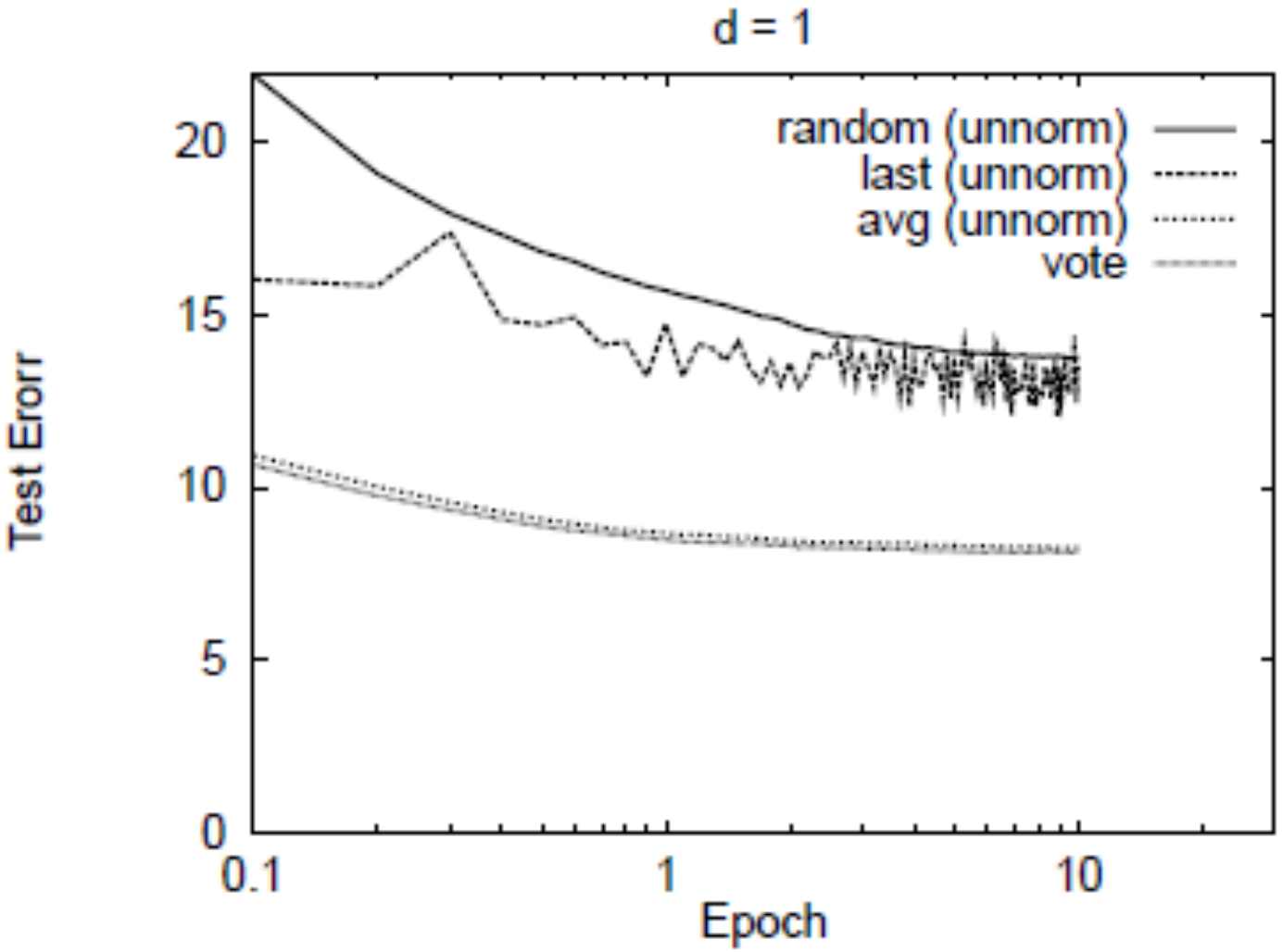
$$P(\text{error in x}) = \sum_k P(\text{error on x}|\text{picked } \mathbf{v}_k) P(\text{picked } \mathbf{v}_k)$$

$$= \sum_k \frac{1}{m_k} \frac{m_k}{m} = \sum_k \frac{1}{m} = \frac{k}{m}$$

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
20
15
10
5
0
0.1
1
10
Epoch

STOPPED - TUES

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

SPARSIFYING THE AVERAGED PERCEPTRON UPDATE

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**
  - t = t+1
  - if sign(**vk.x**) != *y*
    - **vk** = **vk** + *y**x
- Return **va**/t

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

So when there's a mistake at time t on **x**,*y*:

*y****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** + $\sum_{j \in S_k} y_j \mathbf{x}_j$~~
  - t = t+1
  - if sign(**vk.x**) != *y*
    - **vk = vk** + *y**x
    - **va = va** + (T-t)**y****x**
- Return **va**/T

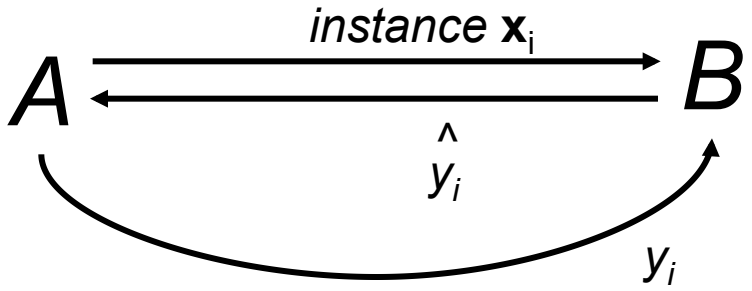
T = the total number of examples in the stream…

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

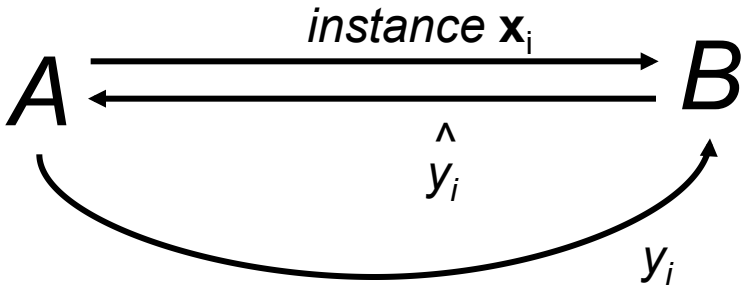
If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$ →

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

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

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

The kernel perceptron



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

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

Compute: $\hat{y}_i = \mathbf{v}_k \cdot \mathbf{x}_i$ →

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

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

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:

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

- Polynomial kernel:

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

- Gaussian kernel:

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

- 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

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

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

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

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

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

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

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

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

Two different computational ways of getting the same behavior

Kernels 101

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

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 $\mathbf{z}$

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)

Kernels 101

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

$K(x,x') = K(x',x)$ ➔ 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 **z**

Fun fact: Gram matrix positive semi-definite ⇔ $K(\mathbf{x}_i,\mathbf{x}_j)=\phi(\mathbf{x}_i), \phi(\mathbf{x}_j)$ for some ϕ
Proof: $\phi(\mathbf{x})$ uses the eigenvectors of K to represent **x**

HASH KERNELS AND "THE HASH TRICK"

Question

- Most of the weights in a classifier are
  - small and not important

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*

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

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

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

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

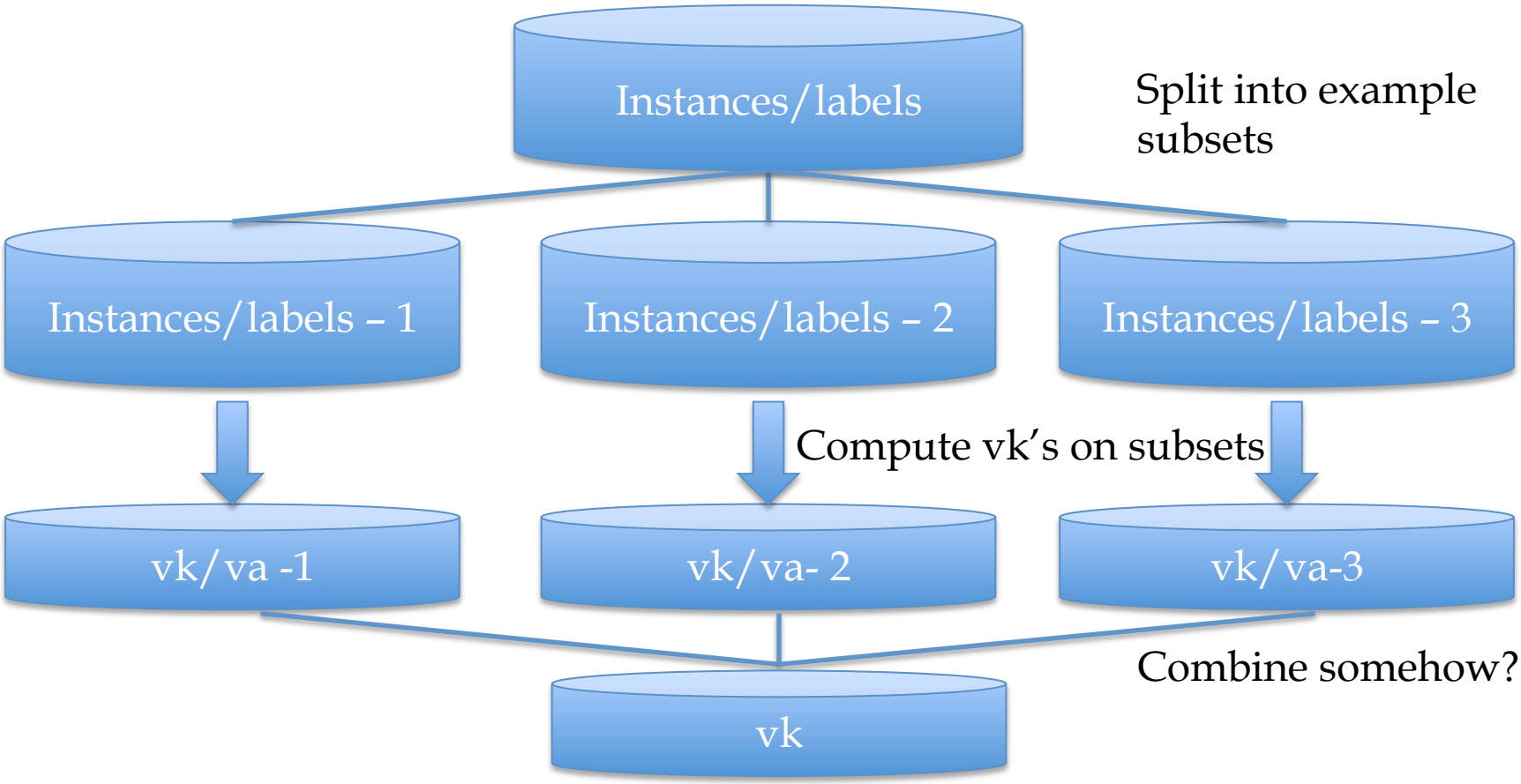
$$\frac{\left| \|x_i - x_j\|_\phi^2 - \|x_i - x_j\|_2^2 \right|}{\|x_i - x_j\|_2^2} \leq \sqrt{\frac{2}{m} + 64\eta^2 \log^2 \frac{n}{2\delta}}.$$

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

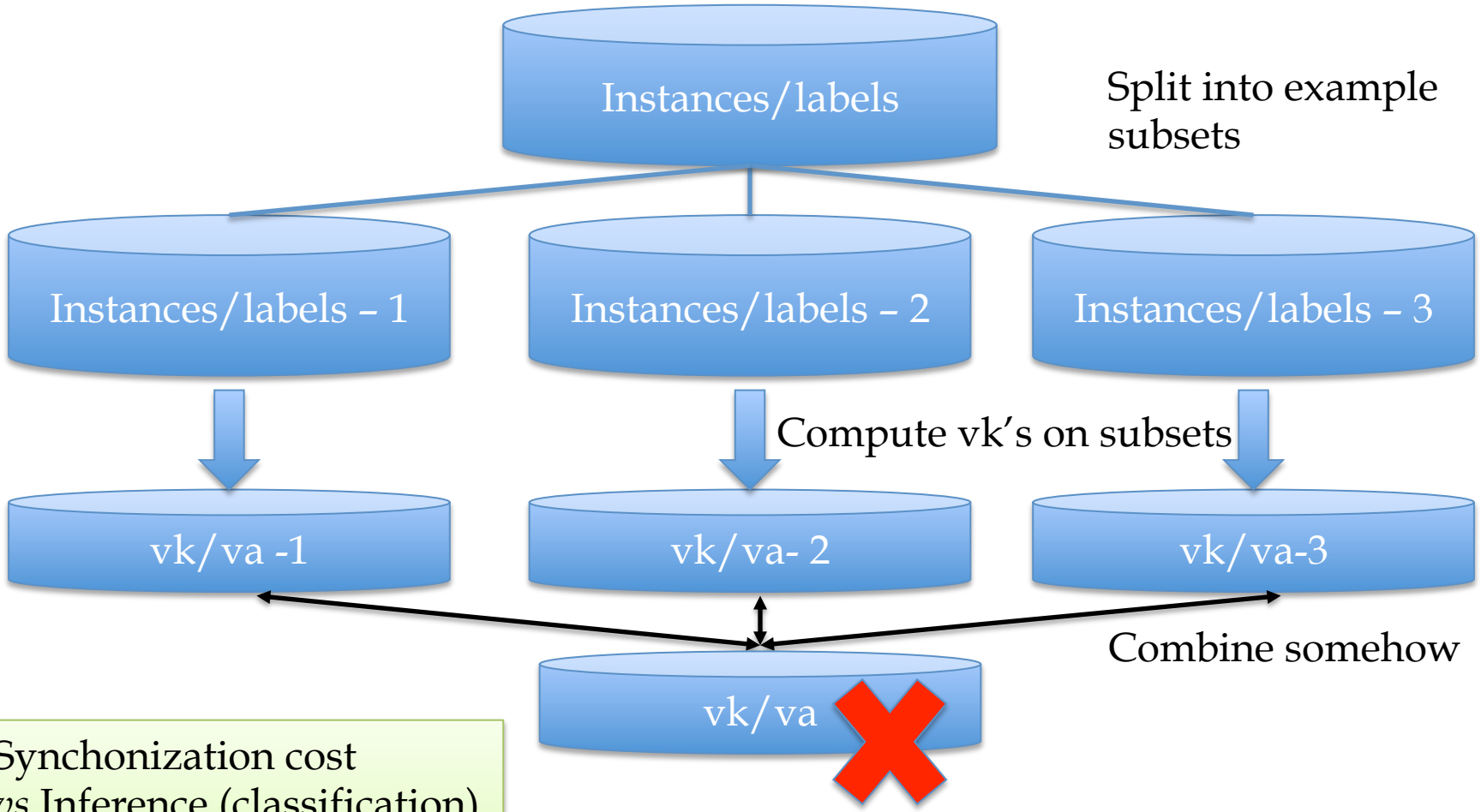
The hash kernel: implementation

- 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*→*w1,w2,…,*

Parallelizing perceptrons



Parallelizing perceptrons



Synchonization cost
vs Inference (classification) cost

A hidden agenda

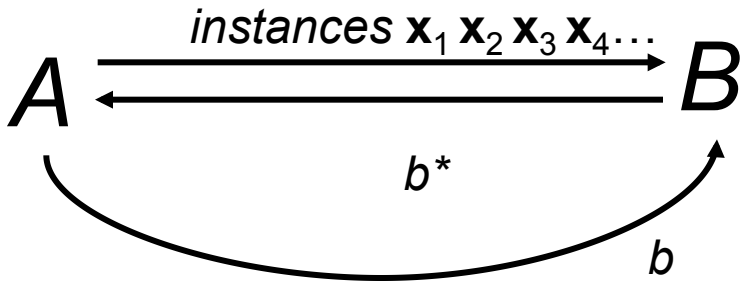
- Part of machine learning is good grasp of theory
- Part of ML is a good grasp of what hacks tend to work
- These are not always the same
  - Especially in big-data situations

- Catalog of useful tricks so far
  - Brute-force estimation of a joint distribution
  - Naive Bayes
  - Stream-and-sort, request-and-answer patterns
  - BLRT and KL-divergence (and when to use them)
  - TF-IDF weighting – especially IDF
    - it's often useful even when we don't understand why
  - Perceptron/mistake bound model
    - often leads to fast, competitive, easy-to-implement methods
    - parallel versions are non-trivial to implement/understand

The Voted Perceptron for Ranking and Structured Classification

William Cohen

The voted perceptron *for ranking*



Compute: $y_i = \hat{\mathbf{v}}_k \cdot \mathbf{x}_i$
Return: the index b^* of the "best" $\mathbf{x}_i$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{x}_b - \mathbf{x}_{b^*}$

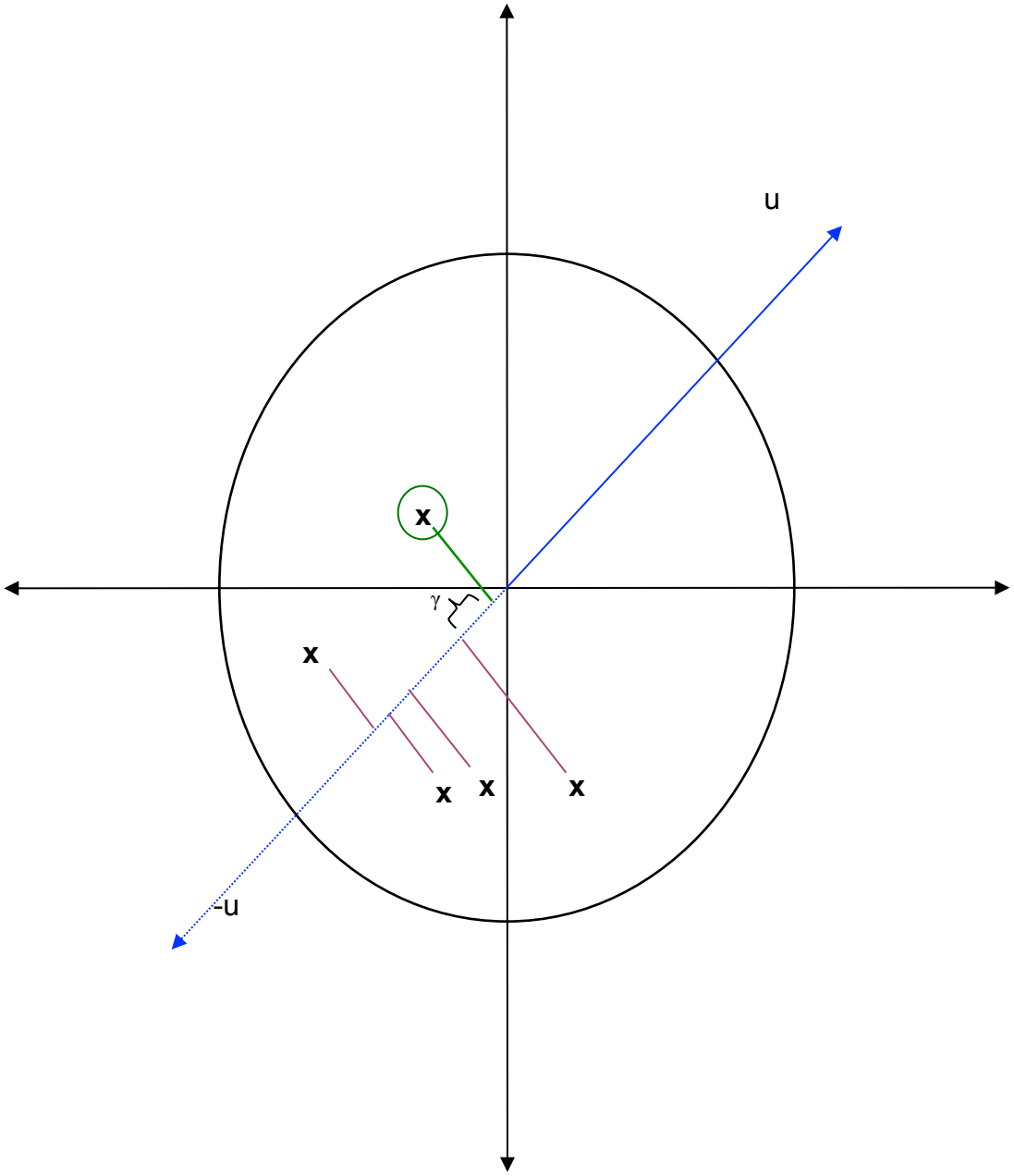
Margin γ. *A* must provide examples that can be correctly ranked with some vector $\mathbf{u}$ with margin $\gamma > 0$, ie

$$\exists \mathbf{u} : \forall \mathbf{x}_{i,1}, \ldots, \mathbf{x}_{i,n_i}, \ell \text{ given by } A, \forall j \neq \ell,\ \mathbf{u} \cdot \mathbf{x}_\ell - \mathbf{u} \cdot \mathbf{x}_j > \gamma$$

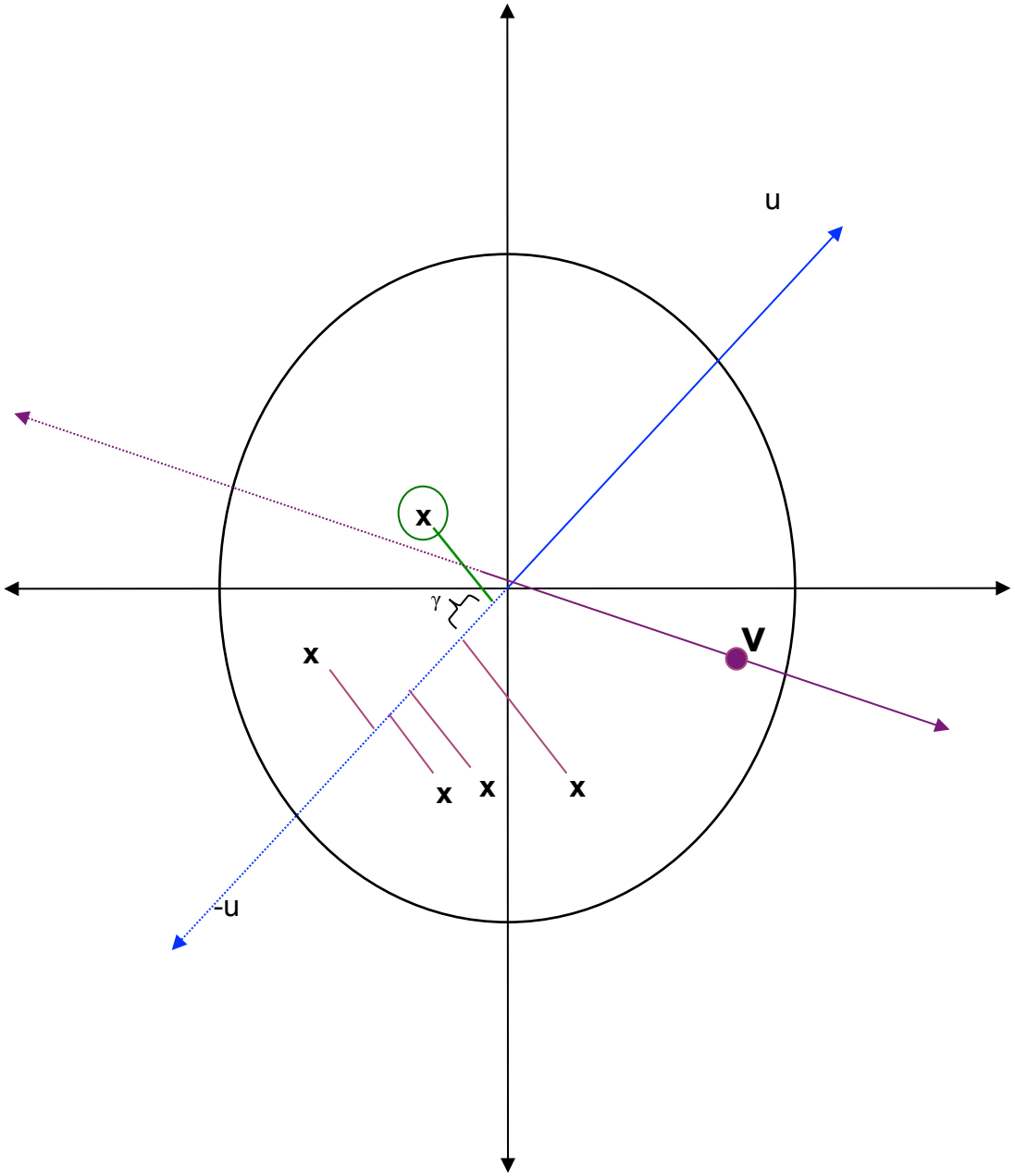
and furthermore, $\|\mathbf{u}\|^2 = 1$.

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

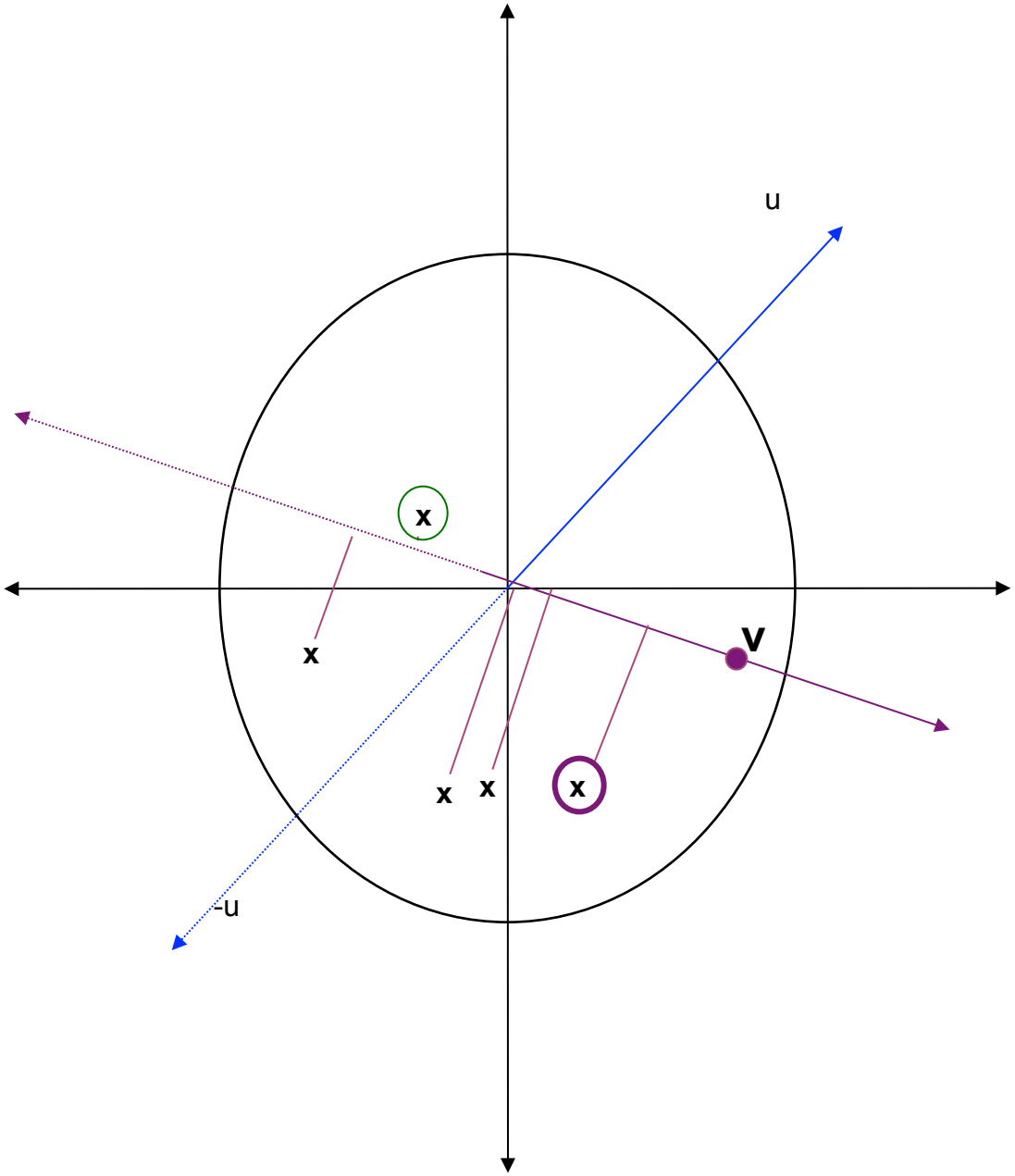
$$\forall \mathbf{x}_i \text{ given by } A,\ \|\mathbf{x}\|^2 < R^2$$



Ranking some x's with the target vector **u**

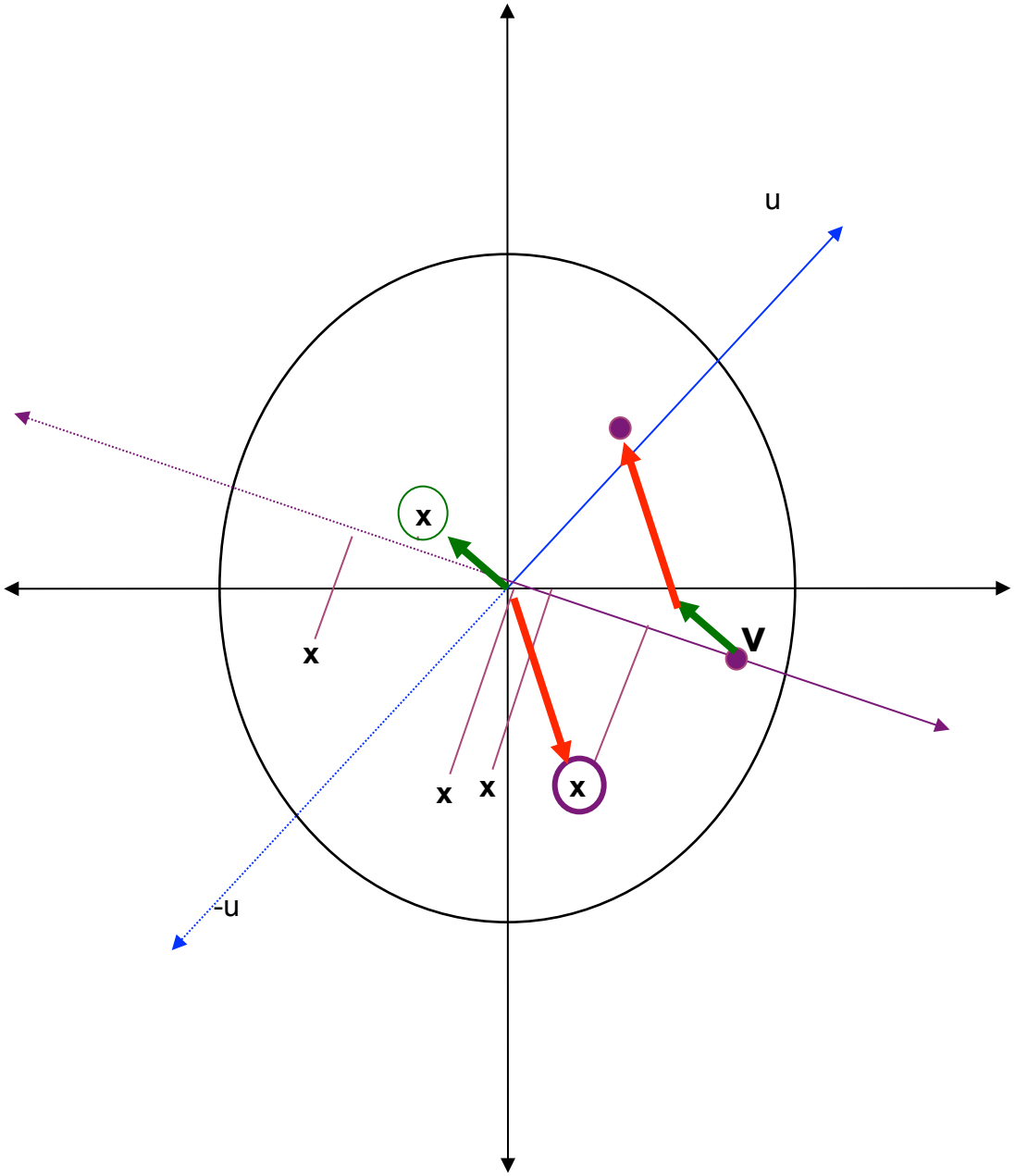


Ranking some x' s with some guess vector **v –** part 1

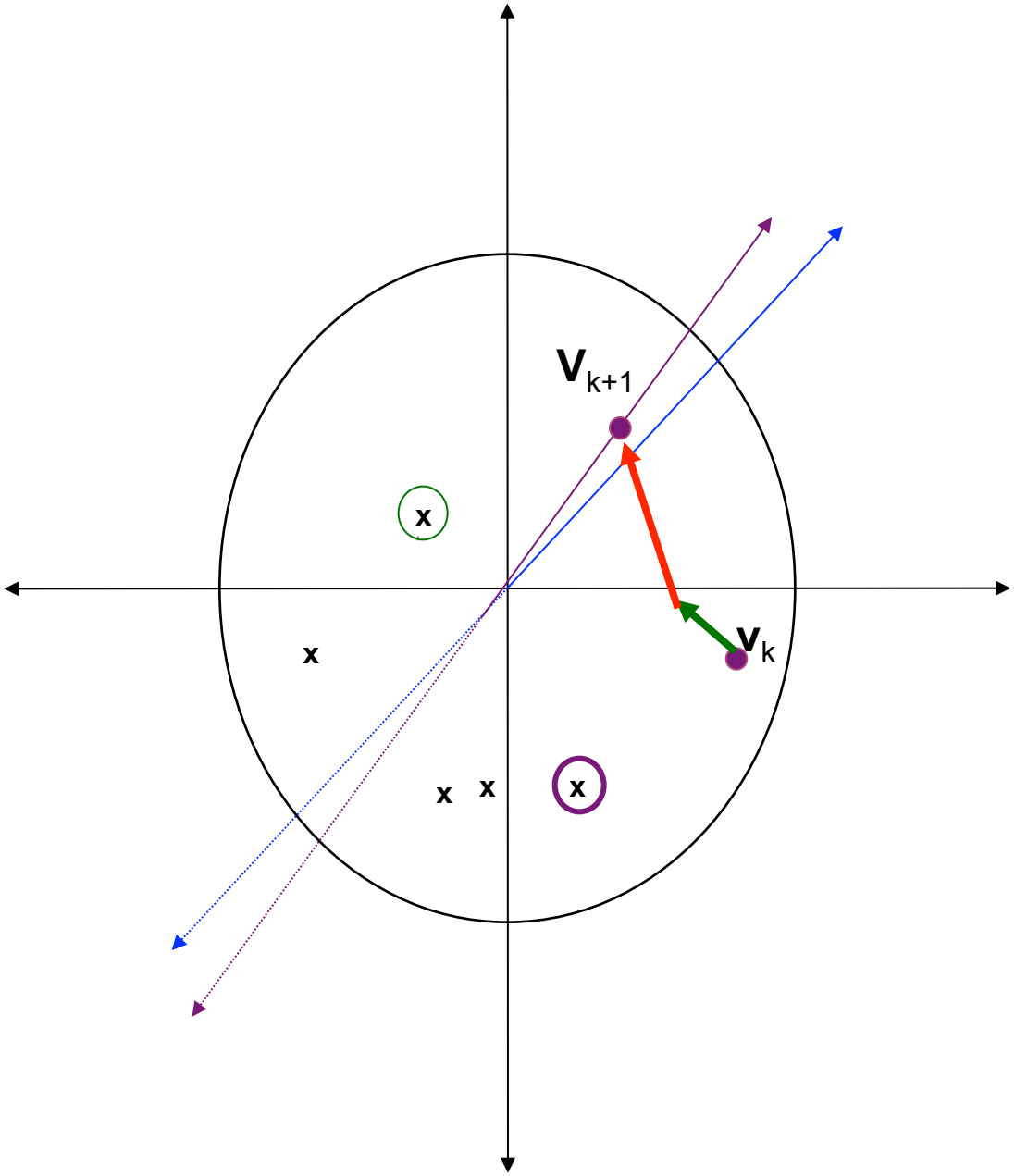


Ranking some x's with some guess vector **v –** part 2.

The purple-circled x is x_{b^*} - the one the learner has chosen to rank highest. The green circled x is x_b, the right answer.



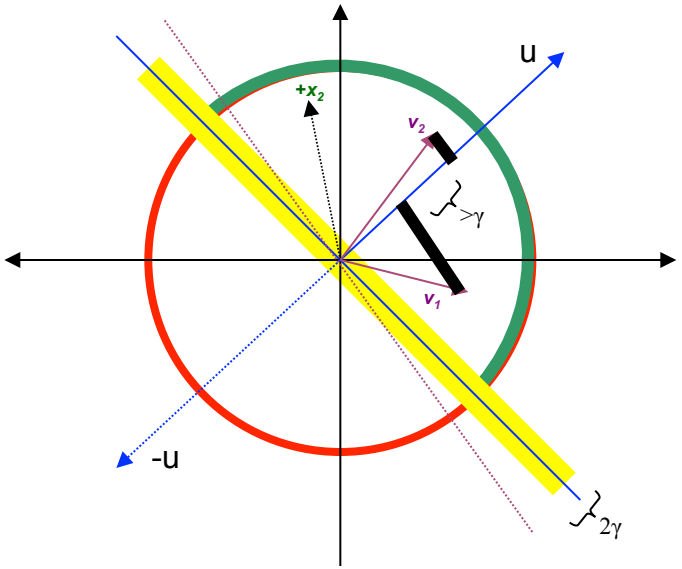
Correcting **v** by adding $x_b - x_{b^*}$

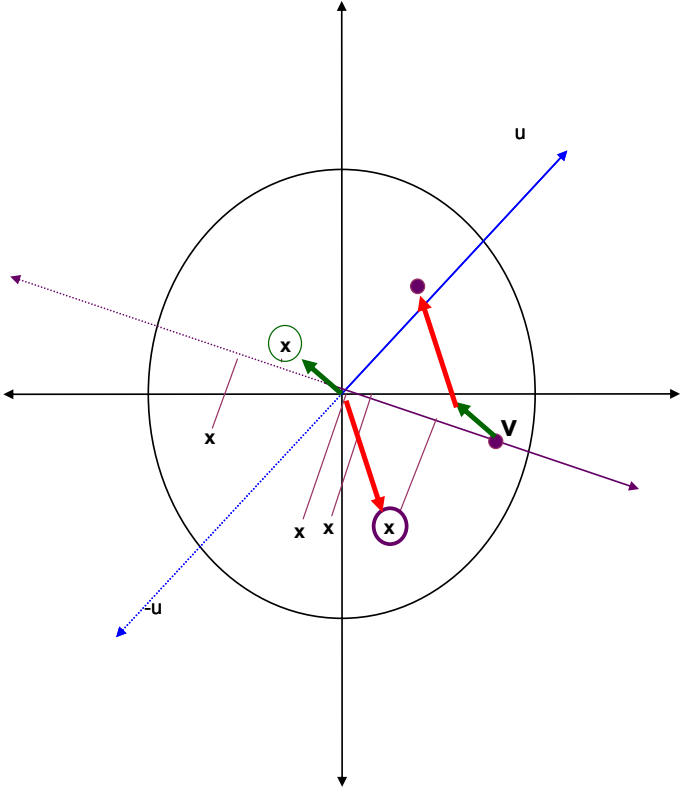


Correcting **v** by adding $x_b - x_{b^*}$

(part 2)

(3a) The guess $\mathbf{v_2}$ after the two positive examples: $\mathbf{v_2}=\mathbf{v_1}+\mathbf{x_2}$



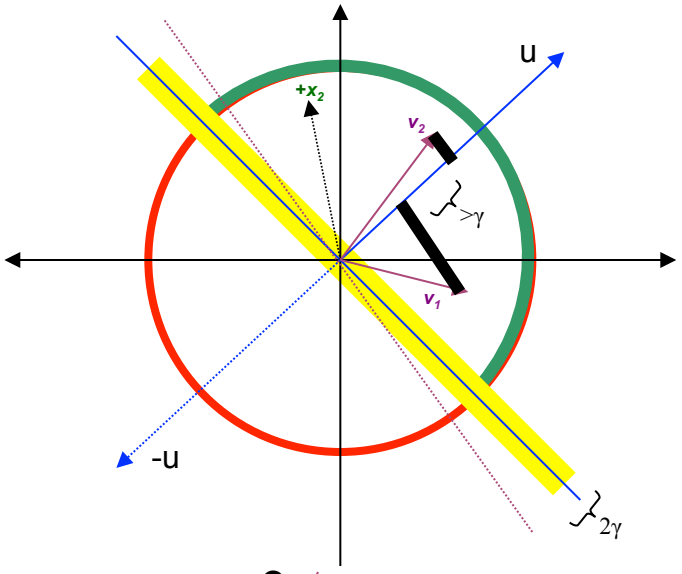


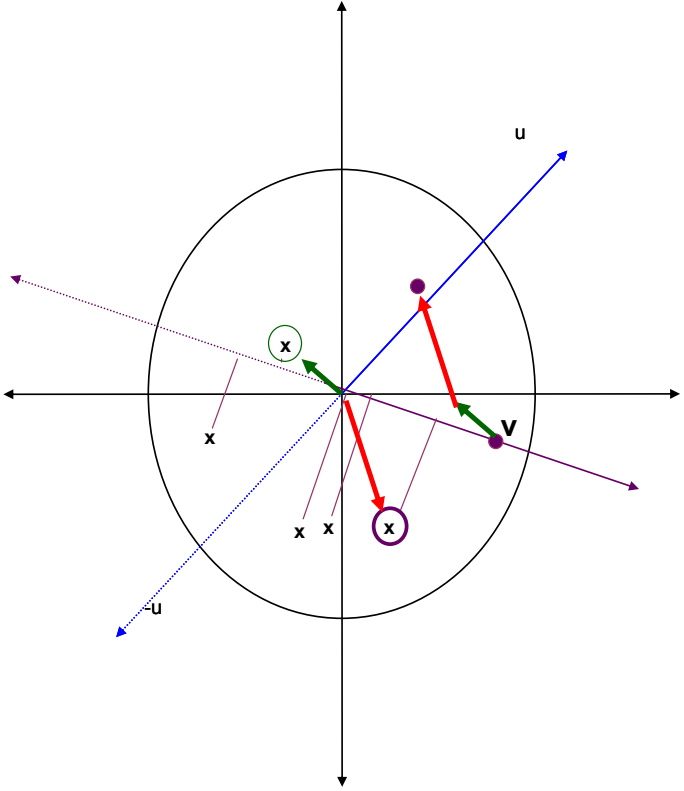
Lemma 1 $\forall k$, $\mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$. *In other words, the dot product between* $\mathbf{v}_k$ *and* $\mathbf{u}$ *increases with each mistake, at a rate depending on the margin* γ.

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: $\mathbf{v}_2=\mathbf{v}_1+\mathbf{x}_2$





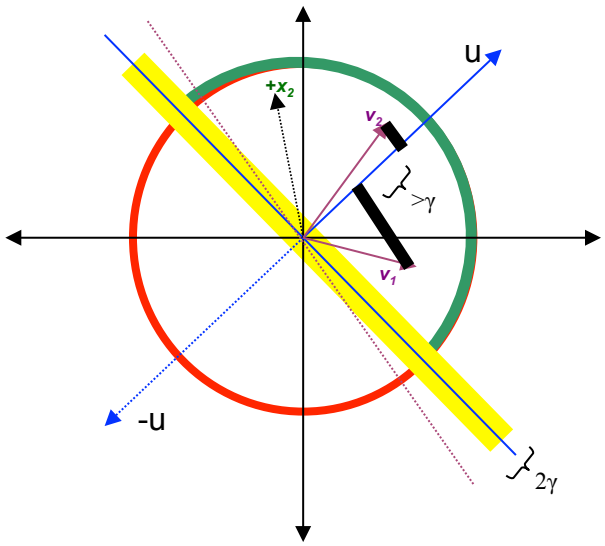
3

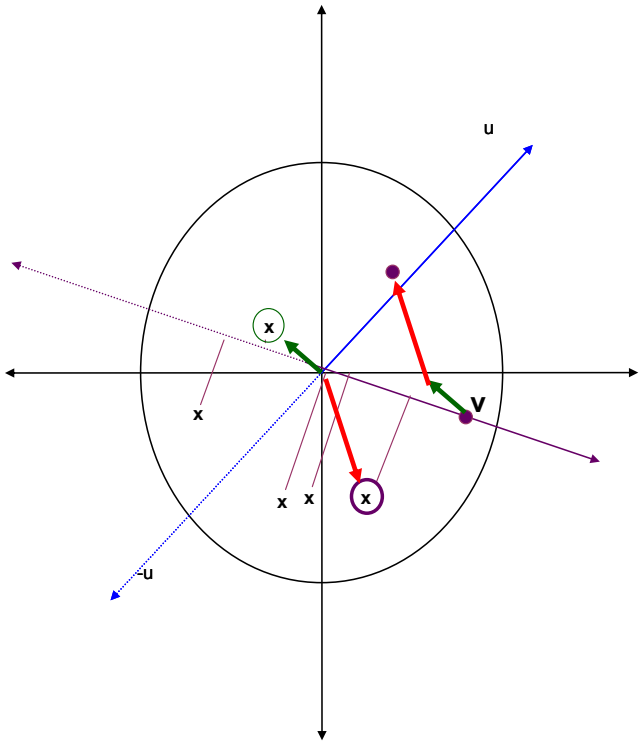
Lemma 1 *$\forall k$, $\mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$. In other words, the dot product between $\mathbf{v}_k$ and $\mathbf{u}$ increases with each mistake, at a rate depending on the margin γ.*

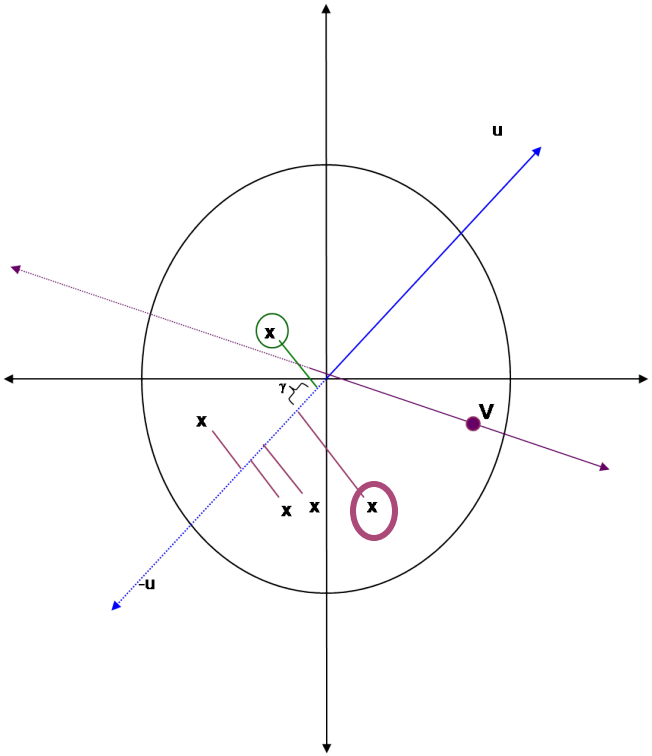
$$\mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k + y_i \mathbf{x}_i) \cdot \mathbf{u}$$
$$\Rightarrow \quad \mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k \cdot \mathbf{u}) + y_i(\mathbf{x}_i \cdot \mathbf{u})$$
$$\Rightarrow \quad \mathbf{v}_{k+1} \cdot \mathbf{u} \geq \mathbf{v}_k \cdot \mathbf{u}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

$$\mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k + \mathbf{x}_{i,\ell} - \mathbf{x}_{i,\hat{\ell}}) \cdot \mathbf{u}$$
$$\Rightarrow \quad \mathbf{v}_{k+1} \cdot \mathbf{u} = \mathbf{v}_k \cdot \mathbf{u} + \mathbf{x}_{i,\ell} \cdot \mathbf{u} - \mathbf{x}_{i,\hat{\ell}} \cdot \mathbf{u}$$
$$\Rightarrow \quad \mathbf{v}_{k+1} \cdot \mathbf{u} \geq \mathbf{v}_k \cdot \mathbf{u} + \gamma$$
$$\Rightarrow \quad \mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$$

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







3

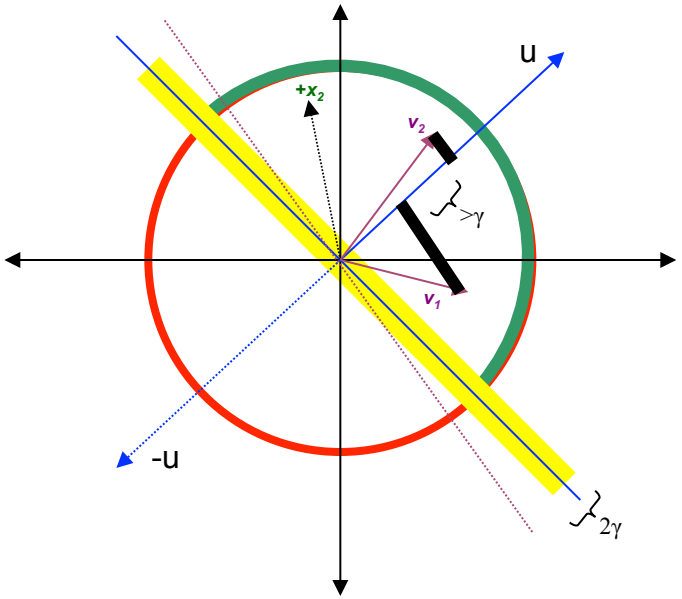
Lemma 1 *$\forall k$, $\mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$. In other words, the dot product between $\mathbf{v}_k$ and $\mathbf{u}$ increases with each mistake, at a rate depending on the margin γ.*

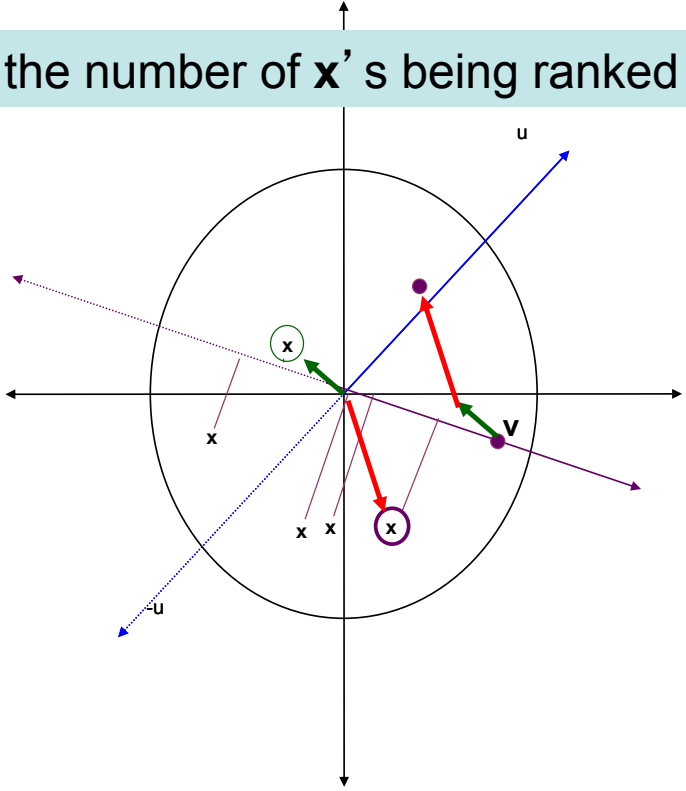
$$\mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k + y_i \mathbf{x}_i) \cdot \mathbf{u}$$
$$\Rightarrow \quad \mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k \cdot \mathbf{u}) + y_i(\mathbf{x}_i \cdot \mathbf{u})$$
$$\Rightarrow \quad \mathbf{v}_{k+1} \cdot \mathbf{u} \geq \mathbf{v}_k \cdot \mathbf{u}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

$$\mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k + \mathbf{x}_{i,\ell} - \mathbf{x}_{i,\hat{\ell}}) \cdot \mathbf{u}$$
$$\Rightarrow \quad \mathbf{v}_{k+1} \cdot \mathbf{u} = \mathbf{v}_k \cdot \mathbf{u} + \mathbf{x}_{i,\ell} \cdot \mathbf{u} - \mathbf{x}_{i,\hat{\ell}} \cdot \mathbf{u}$$
$$\Rightarrow \quad \mathbf{v}_{k+1} \cdot \mathbf{u} \geq \mathbf{v}_k \cdot \mathbf{u} + \gamma$$
$$\Rightarrow \quad \mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$$

Notice this doesn't depend *at all* on the number of **x**'s being ranked

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





Lemma 4 $\forall k,\ \|\mathbf{v}_k\|^2 \leq 2kR.$

Theorem 2 *Under the rules of the ranking perceptron game, it is always the case that* $k < 2R/\gamma^2$.

Neither proof depends on the *dimension* of the **x**'s.

Ranking perceptrons ➔ structured perceptrons

- The API:
  - A sends B a (maybe **huge**) set of items to rank
  - B finds the single **best** one according to the current weight vector
  - A tells B which one was actually best
- Structured classification on a sequence
  - Input: list of words: $\mathbf{x}=(w_1,\ldots,w_n)$
  - Output: list of labels: $\mathbf{y}=(y_1,\ldots,y_n)$
  - If there are K classes, there are K^n labels possible for **x**

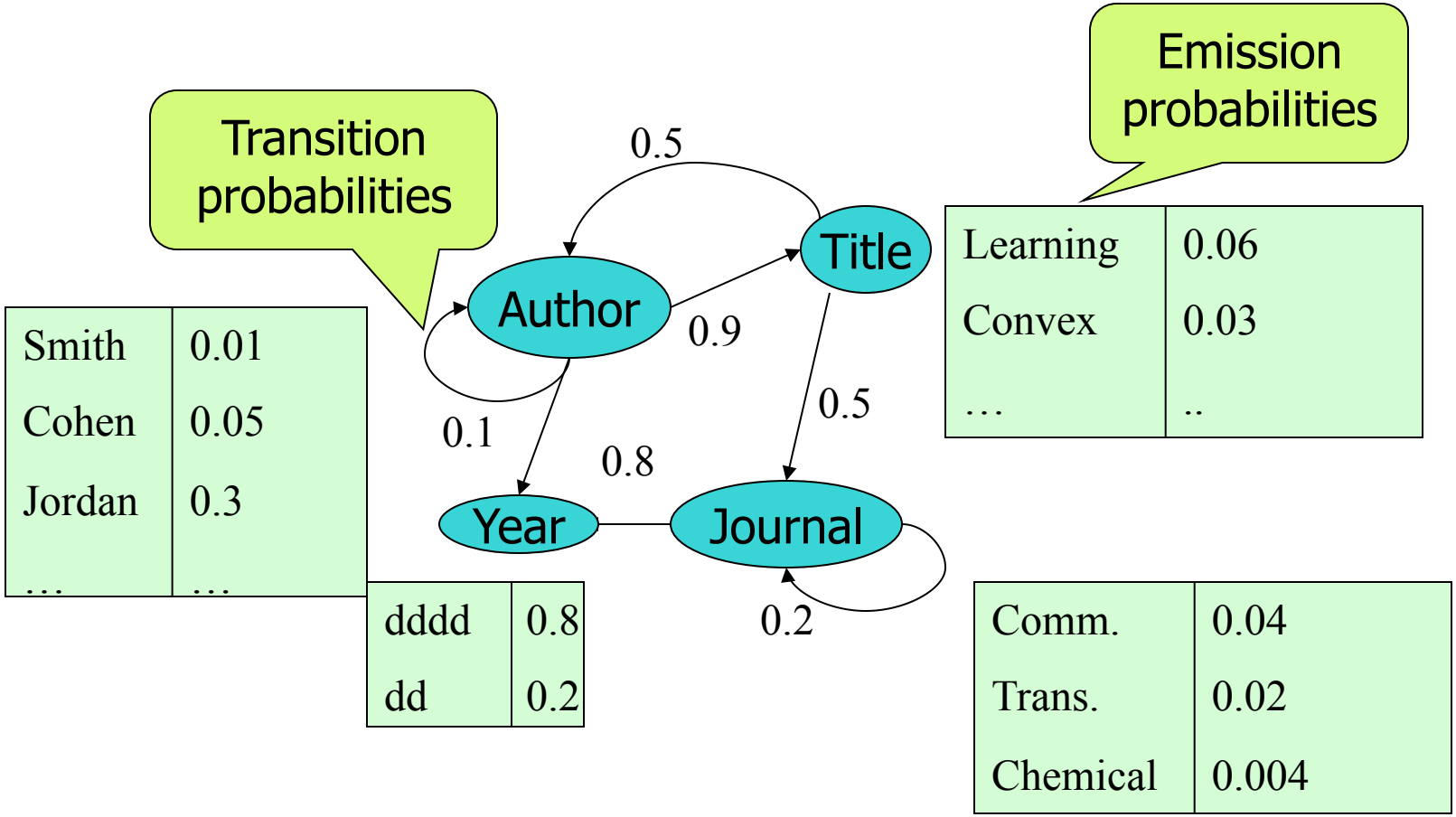
Borkar et al's: HMMs for segmentation

- Example: Addresses, bib records
- Problem: some DBs may split records up differently (eg no "mail stop" field, combine address and apt #, …) or not at all
- Solution: Learn to segment textual form of records

Author Year Title Journal Volume Page

P.P.Wangikar, T.P. Graycar, D.A. Estell, D.S. Clark, J.S. Dordick (1993) Protein and Solvent Engineering of Subtilising BPN' in Nearly Anhydrous Organic Media J.Amer. Chem. Soc. 115, 12231-12237.

IE with Hidden Markov Models



Inference for linear-chain MRFs

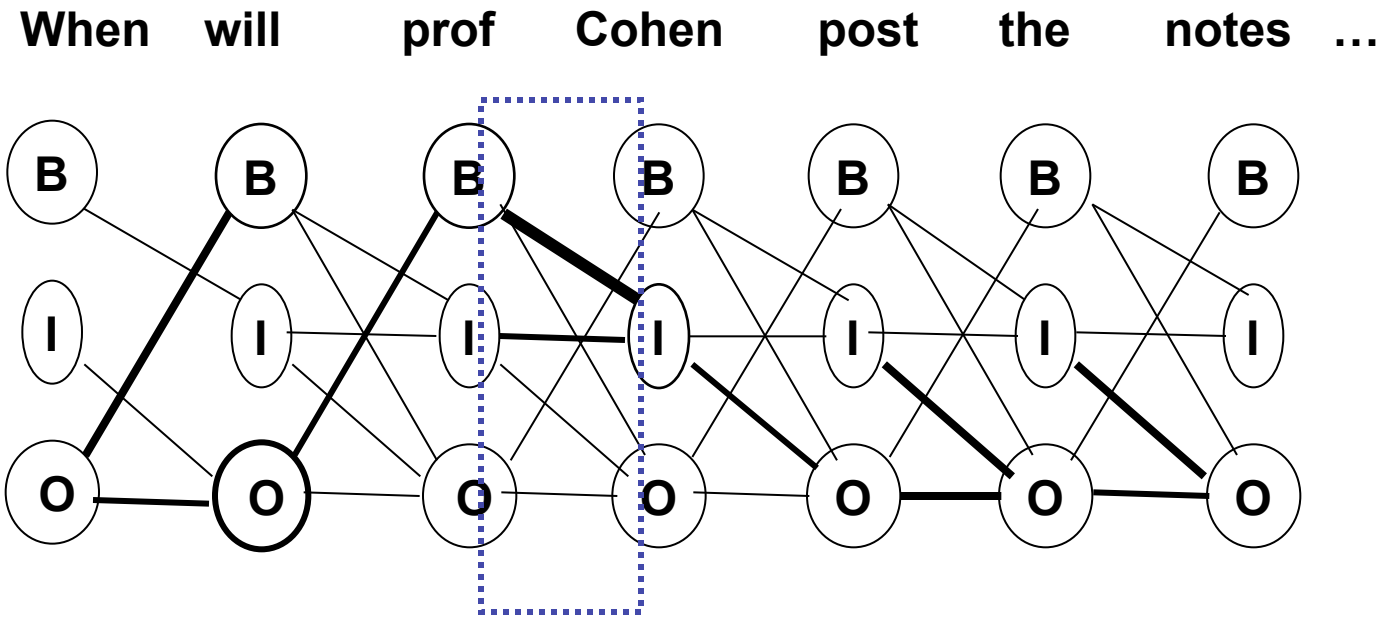
When will prof Cohen post the notes …

Idea 1: features are properties of *two adjacent tokens*, and the *pair* of labels assigned to them.

- (y(i)==B or y(i)==I) and (token(i) is capitalized)
- (y(i)==I and y(i-1)==B) and (token(i) is hyphenated)
- (y(i)==B and y(i-1)==B)
    - eg “tell Ziv William is on the way”

Idea 2: construct a graph where each *path* is a possible sequence labeling.

Inference for a linear-chain MRF



- Inference: find the highest-weight path
- This can be done efficiently using dynamic programming (Viterbi)

Ranking perceptrons ➔ structured perceptrons

- The API:
  - A sends B a (maybe **huge**) set of items to rank
  - B finds the single **best** one according to the current weight vector
  - A tells B which one was actually best
- Structured classification on a sequence
  - Input: list of words: $\mathbf{x}=(w_1,\ldots,w_n)$
  - Output: list of labels: $\mathbf{y}=(y_1,\ldots,y_n)$
  - If there are K classes, there are K^n labels possible for **x**

Ranking perceptrons ➔ structured perceptrons

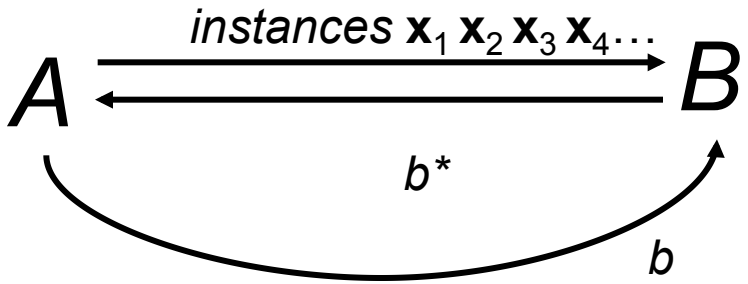
- The API:
  - A sends B a (maybe **huge**) set of items to rank
  - B finds the single **best** one according to the current weight vector
  - A tells B which one was actually best

- Structured classification on a sequence
  - Input: list of words: $\mathbf{x}=(w_1,\ldots,w_n)$
  - Output: list of labels: $\mathbf{y}=(y_1,\ldots,y_n)$
  - If there are K classes, there are K^n labels possible for **x**

Ranking perceptrons ➔ structured perceptrons

- New API:
  - A sends B the word sequence **x**
  - B finds the single **best** **y** according to the current weight vector using Viterbi
  - A tells B which **y** was actually best
  - This is equivalent to ranking pairs g=(**x**,**y′**)
- Structured classification on a sequence
  - Input: list of words: **x**=(w_1,…,w_n)
  - Output: list of labels: **y**=(y_1,…,y_n)
  - If there are K classes, there are K^n labels possible for **x**

The voted perceptron *for ranking*

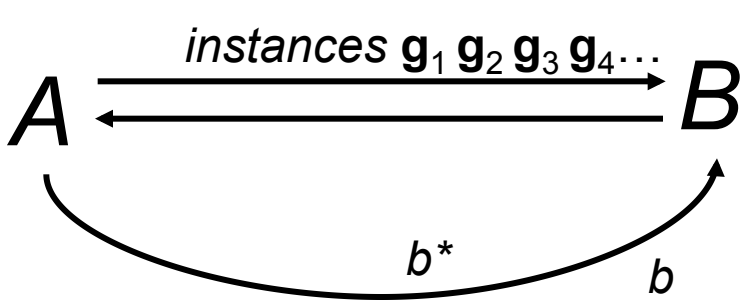


Compute: $y_i = \hat{\mathbf{v}}_k \cdot \mathbf{x}_i$
Return: the index b^* of the "best" $\mathbf{x}_i$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{x}_b - \mathbf{x}_{b^*}$

Change number one is notation: replace **x** with **g**

The voted perceptron *for NER*



Compute: $y_i = \hat{\mathbf{v}}_k \cdot \mathbf{g}_i$
Return: the index b^* of the "best" $\mathbf{g}_i$

If mistake: $\mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{g}_b - \mathbf{g}_{b^*}$

1. A sends B feature functions, and instructions for creating the instances **g**:
   - A sends a word vector $\mathbf{x}_i$**.** Then B could create the instances $\mathbf{g}_1 = \mathbf{F}(\mathbf{x}_i,\mathbf{y}_1)$, $\mathbf{g}_2 = \mathbf{F}(\mathbf{x}_i,\mathbf{y}_2)$, …
   - but instead B just returns the **y*** that gives the best score for the dot product $\mathbf{v}_k \cdot \mathbf{F}(\mathbf{x}_i,\mathbf{y}^*)$ by using Viterbi.
2. A sends B the correct label sequence $\mathbf{y}_i$.
3. On errors, B sets $\mathbf{v}_{k+1} = \mathbf{v}_k + \mathbf{g}_b - \mathbf{g}_{b^*} = \mathbf{v}_k + \mathbf{F}(\mathbf{x}_i,\mathbf{y}) - \mathbf{F}(\mathbf{x}_i,\mathbf{y}^*)$

Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms

Michael Collins
AT&T Labs-Research, Florham Park, New Jersey.
mcollins@research.att.com

EMNLP 2002

Some background…

- Collins' parser: generative model…
- …New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron, Collins and Duffy, ACL 2002.
- …Ranking Algorithms for Named-Entity Extraction: Boosting and the Voted Perceptron, Collins, ACL 2002.
  - Propose entities using a MaxEnt tagger (as in MXPOST)
  - Use beam search to get *multiple* taggings for each document (20)
  - Learn to *rerank* the candidates to push correct ones to the top, using some new candidate-specific features:
    - Value of the "whole entity" (e.g., "Professor_Cohen")
    - Capitalization features for the whole entity (e.g., "Xx+_Xx+")
    - Last word in entity, and capitalization features of last word
    - Bigrams/Trigrams of words and capitalization features before and after the entity

Some background…

	P	R	F
Max-Ent	84.4	86.3	85.3
Boosting	87.3(18.6)	87.9(11.6)	87.6(15.6)
Voted Perceptron	87.3(18.6)	88.6(16.8)	87.9(17.7)

Figure 5: Results for the three tagging methods. P = precision, R = recall, F = F-measure. Figures in parantheses are relative improvements in error rate over the maximum-entropy model. All figures are percentages.

And back to the paper…..

Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms

Michael Collins
AT&T Labs-Research, Florham Park, New Jersey.
mcollins@research.att.com

EMNLP 2002, Best paper

Collins' Experiments

- POS tagging
- NP Chunking (words and POS tags from Brill's tagger as features) and BIO output tags
- Compared Maxent Tagging/MEMM's (with iterative scaling) and "Voted Perceptron trained HMM's"
  - With and w/o averaging
  - With and w/o feature selection (count>5)

Collins' results

NP Chunking Results

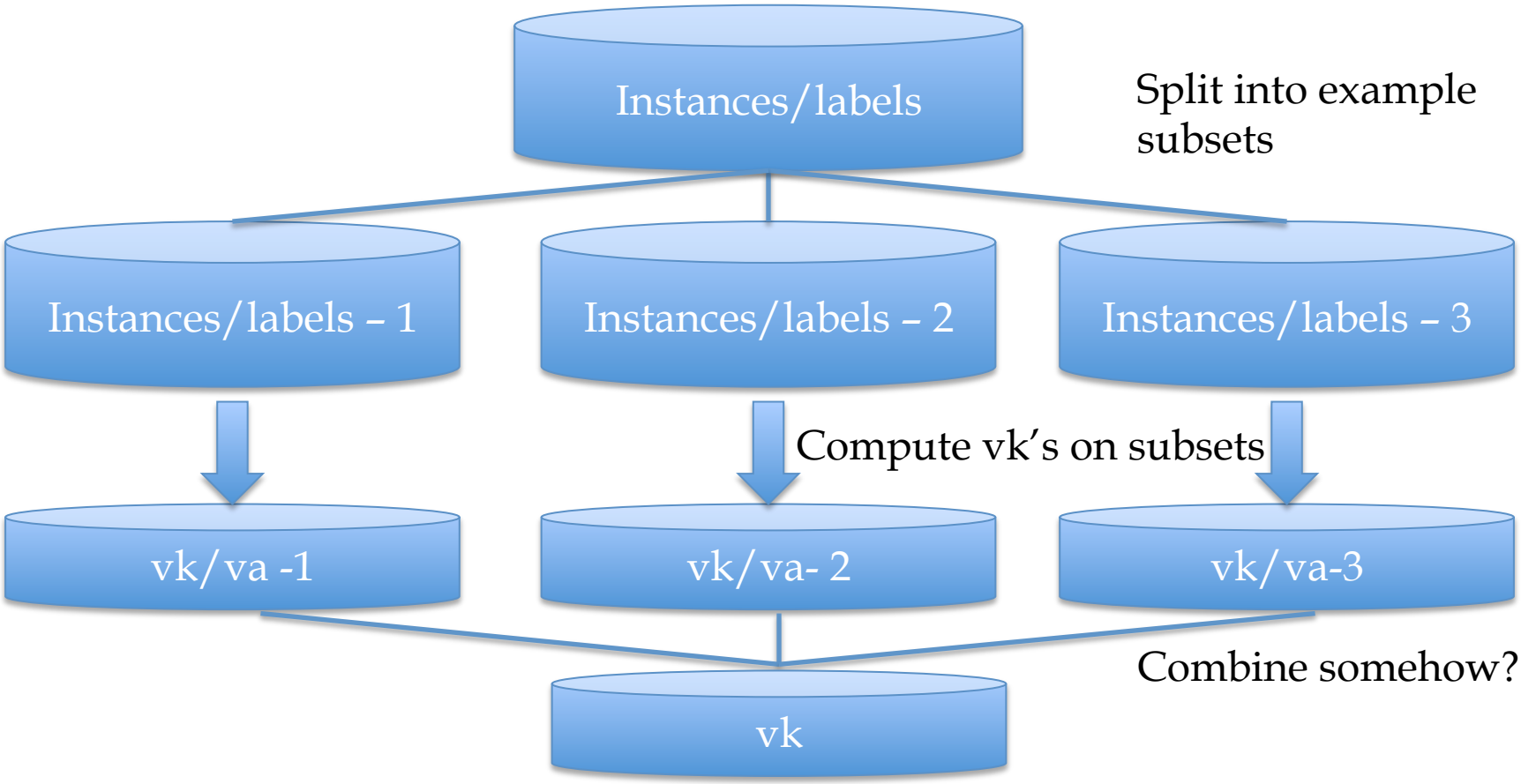
Method	F-Measure	Numits
Perc, avg, cc=0	93.53	13
Perc, noavg, cc=0	93.04	35
Perc, avg, cc=5	93.33	9
Perc, noavg, cc=5	91.88	39
ME, cc=0	92.34	900
ME, cc=5	92.65	200

POS Tagging Results

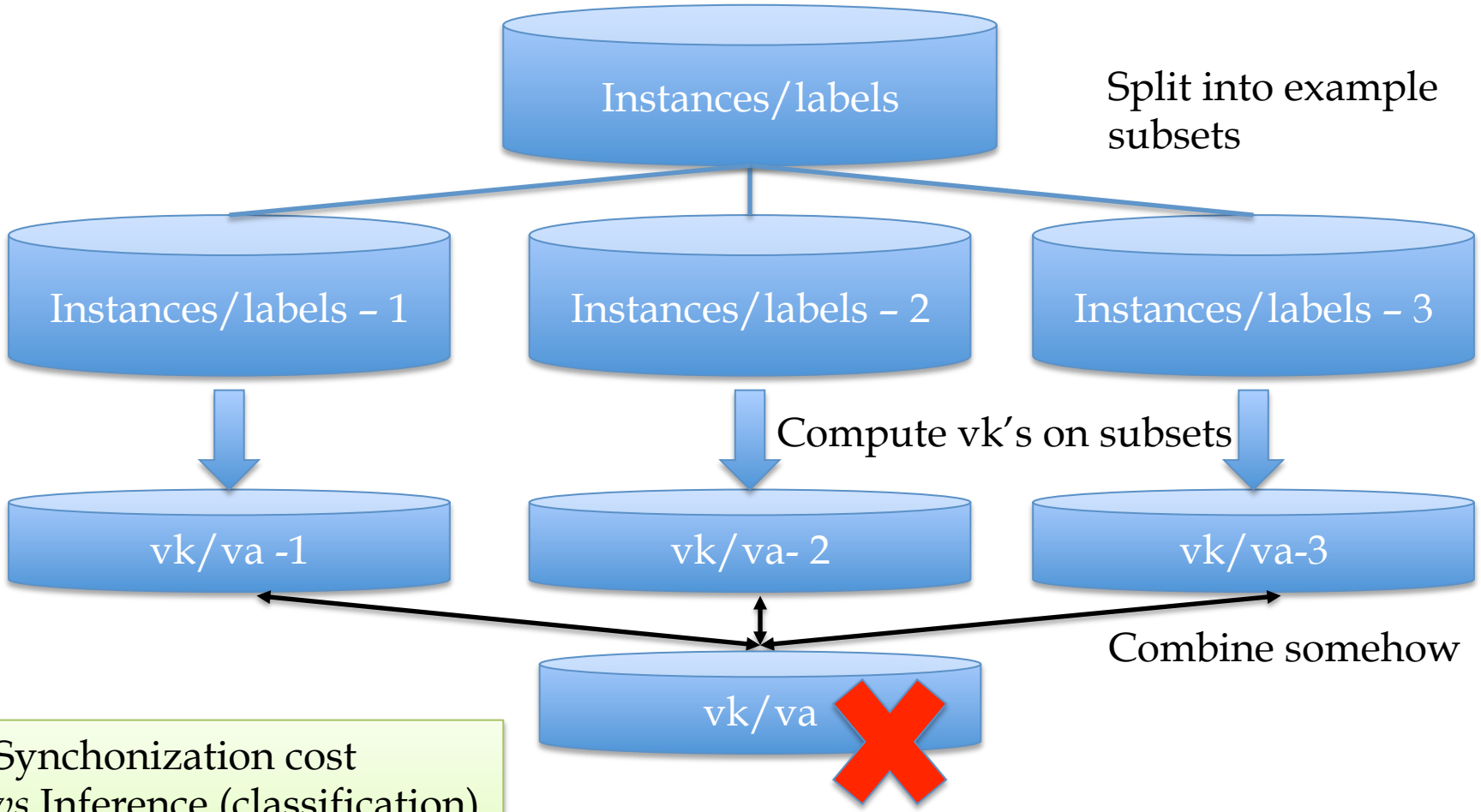
Method	Error rate/%	Numits
Perc, avg, cc=0	2.93	10
Perc, noavg, cc=0	3.68	20
Perc, avg, cc=5	3.03	6
Perc, noavg, cc=5	4.04	17
ME, cc=0	3.4	100
ME, cc=5	3.28	200

Figure 4: Results for various methods on the part-of-speech tagging and chunking tasks on development data. All scores are error percentages. Numits is the number of training iterations at which the best score is achieved. Perc is the perceptron algorithm, ME is the maximum entropy method. Avg/noavg is the perceptron with or without averaged parameter vectors. cc=5 means only features occurring 5 times or more in training are included, cc=0 means all features in training are included.

Parallelizing perceptrons



Parallelizing perceptrons



Synchonization cost
vs Inference (classification) cost