# Naïve Bayes for Big Data

These slides adapted from William Cohen (CMU)

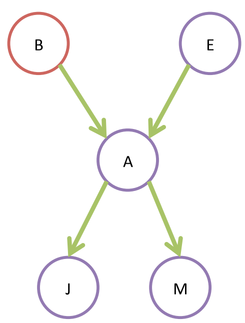
## Administrivia

- Have you met your groups yet?
- Proposals Due Tuesday!

- HW1 Due Friday
- HW2 out Friday

## Recall Bayes Nets

$$
\begin{aligned}
&= P(x_n,\ldots,x_1) \\
&= P(x_n \mid x_{n-1},\ldots,x_1)P(x_{n-1},\ldots,x_1) \\
&= P(x_n \mid x_{n-1},\ldots,x_1)P(x_{n-1} \mid x_{n-2},\ldots,x_1)P(x_{n-2},\ldots,x_1) \\
&= \ldots \\
&= \prod_{i=1}^{n} P(x_i \mid x_{i-1},\ldots,x_1) = \prod_{i=1}^{n} P(x_i \mid parents(x_i))
\end{aligned}
$$



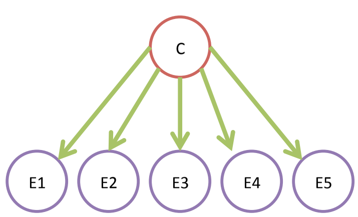

## Naive Bayes

$$P(c \mid e_1,\ldots,e_n) = \alpha P(e_1 \mid c)\ldots P(e_n \mid c)P(c)$$

- Assumption:
  Attributes are conditionally independent (given the class value)

## What would that graph look like?

$$P(c \mid e_1,\ldots,e_n) = \alpha P(e_1 \mid c)\ldots P(e_n \mid c)P(c)$$




## Implementing Naïve Bayes

$$
\begin{aligned}
&P(c \mid e_1,\ldots,e_n) = \alpha P(e_1 \mid c)\ldots P(e_n \mid c)P(c) \\
&= (C(e_1 \,\&\, c))/C(c) * (C(e_2 \,\&\, c))/C(c) * \ldots * C(c)/C(any)
\end{aligned}
$$

Assume hashtable holding all counts fits in memory

## Complexity of Naïve Bayes

- You have a *train* dataset and a *test* dataset
- Initialize an "event counter" (hashtable) C
- For each example *id, y,* $x_1,\dots,x_d$ in *train:*
  - C("*Y*=ANY") ++; C("*Y=y*") ++
  - For *j* in *1..d*:
    - C("*Y=y* ^ *X*=$x_j$") ++
    - …
- For each example *id, y,* $x_1,\dots,x_d$ in *test:*
  - For each *y'* in *dom(Y):*
    - Compute log Pr(*y'*,$x_1,\dots,x_d$) =

$$= \left(\sum_j \log \frac{C(X = x_j \wedge Y = y') + mq_x}{C(X = ANY \wedge Y = y') + m}\right) + \log \frac{C(Y = y') + mq_y}{C(Y = ANY) + m}$$

  - Return the best *y'*

where:
$q_x = 1/|V|$
$q_y = 1/|dom(Y)|$
$mq_x = 1$

Sequential reads

Complexity: O(*n*), *n*=size of *train*

Sequential reads

Complexity: O(|*dom(Y)*|**n'*), *n'*=size of *test*

7

## Complexity of Naïve Bayes

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Why?

**General Purpose - Current Generation**

| | | | | | |
|---|---|---|---|---|---|
| t2.nano | 1 | Variable | 0.5 | EBS Only | $0.0065 per Hour |
| t2.micro | 1 | Variable | 1 | EBS Only | $0.013 per Hour |
| t2.small | 1 | Variable | 2 | EBS Only | $0.026 per Hour |
| t2.medium | 2 | Variable | 4 | EBS Only | $0.052 per Hour |
| t2.large | 2 | Variable | 8 | EBS Only | $0.104 per Hour |
| m4.large | 2 | 6.5 | 8 | EBS Only | $0.12 per Hour |
| m4.xlarge | 4 | 13 | 16 | EBS Only | $0.239 per Hour |
| m4.2xlarge | 8 | 26 | 32 | EBS Only | $0.479 per Hour |
| m4.4xlarge | 16 | 53.5 | 64 | EBS Only | $0.958 per Hour |
| m4.10xlarge | 40 | 124.5 | 160 | EBS Only | $2.394 per Hour |
| m3.medium | 1 | 3 | 3.75 | 1 x 4 SSD | $0.067 per Hour |
| m3.large | 2 | 6.5 | 7.5 | 1 x 32 SSD | $0.133 per Hour |
| m3.xlarge | 4 | 13 | 15 | 2 x 40 SSD | $0.266 per Hour |
| m3.2xlarge | 8 | 26 | 30 | 2 x 80 SSD | $0.532 per Hour |

**Micro:** $0.006/hr
0.6G memory
**Standard**:
S: 1.7Gb $0.03/hr
L: 7.5Gb
XL: 15Gb $0.24/hr
**Hi Memory**:
XXL: 32 GB $0.48/hr
10XL: 160GB $2.40/hr

8

## Event Counters Don't fit in RAM

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Why?
- Heaps' Law: If *V* is the size of the vocabulary and the *n* is the length of the corpus in words:

$$V = Kn^{\beta} \quad \text{with constants } K,\ 0 < \beta < 1$$

- Typical constants:
  - $K \approx$ 1/10 - 1/100
  - $\beta \approx$ 0.4-0.6 (approx. square-root)
- Why?
  - Proper names, missspellings, neologisms, …
- Summary:
  - For text classification for a corpus with O(n) words, expect to use O(sqrt(n)) storage for vocabulary.
  - Scaling might be worse for other cases (e.g., hypertext, phrases, …)

9

Some "obvious" answers

## SCALING TO LARGE VOCABULARIES: HOW?

10

## What's next

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Possible approaches:
  - Use a database? (or at least a key-value store)

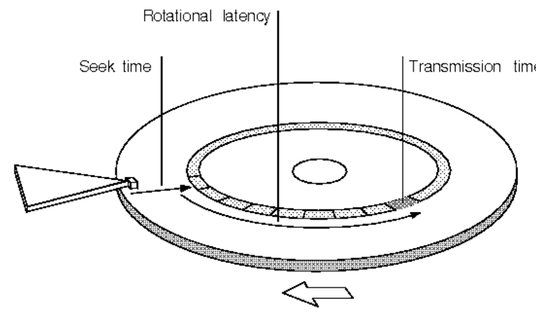
11

## Numbers (Jeff Dean says) Everyone Should Know

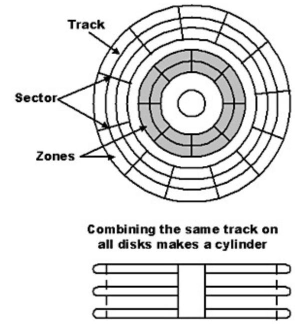
```
L1 cache reference                          0.5 ns
Branch mispredict                             5 ns
L2 cache reference                            7 ns
Mutex lock/unlock                           100 ns
Main memory reference                       100 ns
Compress 1K bytes with Zippy             10,000 ns
Send 2K bytes over 1 Gbps network        20,000 ns
Read 1 MB sequentially from memory      250,000 ns
Round trip within same datacenter       500,000 ns
Disk seek                            10,000,000 ns
Read 1 MB sequentially from network  10,000,000 ns
Read 1 MB sequentially from disk     30,000,000 ns
Send packet CA->Netherlands->CA     150,000,000 ns
```

~= 10x
~= 15x
40x
~= 100,000x

12








# Naïve Bayes for Big Data

- How to implement Naïve Bayes
  – Assuming the event counters do *not* fit in memory
- Possible approaches:
  – Use a database?

    O(n*scan) → O(n*scan*seek)
    - Counts are stored on disk, not in memory
    - …So, accessing a count might involve some seeks
      – Caveat: many DBs are good at caching frequently-used values, so seeks might be infrequent …..
  – In memory distributed database?



# Numbers (Jeff Dean says) Everyone Should Know

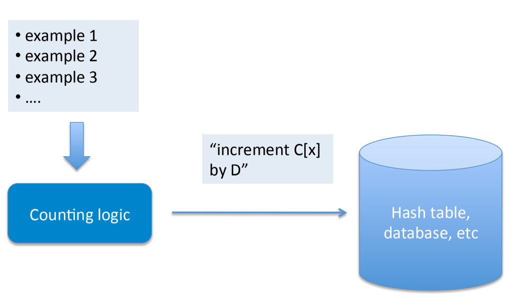
```
L1 cache reference                             0.5 ns
Branch mispredict                                5 ns
L2 cache reference                               7 ns   ~= 10x
Mutex lock/unlock                              100 ns
Main memory reference                          100 ns   ~= 15x
Compress 1K bytes with Zippy                10,000 ns
Send 2K bytes over 1 Gbps network           20,000 ns
Read 1 MB sequentially from memory         250,000 ns
Round trip within same datacenter          500,000 ns
Disk seek                               10,000,000 ns
Read 1 MB sequentially from network     10,000,000 ns
Read 1 MB sequentially from disk        30,000,000 ns   ~= 100,000x
                                           000,000 ns
```

40x

Best case (data is in same sector/block), but of course it could be scattered all over disk



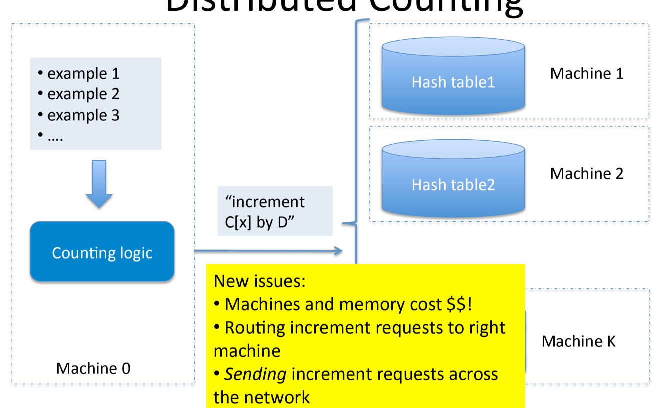
# Counting




Hashtable issue: memory is too small
Database issue: seeks are slow



# Distributed Counting




New issues:
- Machines and memory cost $$!
- Routing increment requests to right machine
- *Sending* increment requests across the network
- **Communication complexity**

## Numbers (Jeff Dean says) Everyone Should Know

| | | |
|---|---|---|
| L1 cache reference | 0.5 ns | |
| Branch mispredict | 5 ns | |
| L2 cache reference | 7 ns | ~= 10x |
| Mutex lock/unlock | 100 ns | |
| Main memory reference | 100 ns | ~= 15x |
| Compress 1K bytes with Zippy | 10,000 ns | |
| Send 2K bytes over 1 Gbps network | 20,000 ns | |
| Read 1 MB sequentially from memory | 250,000 ns | |
| Round trip within same datacenter | 500,000 ns | |
| Disk seek | 10,000,000 ns | 40x |
| Read 1 MB sequentially from network | 10,000,000 ns | |
| Read 1 MB sequentially from disk | 30,000,000 ns | ~= 100,000x |
| Send packet CA->Netherlands->CA | 150,000,000 ns | |



## Naïve Bayes for Big Data

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Possible approaches: O(n*scan) ➔ O(n*scan+n*send)
  - Use a **memory-based distributed** database?
    - Extra cost: Communication costs: O(n) … but that's "ok"
    - Extra complexity: routing requests correctly
      - Note: If the increment requests were ordered seeks would not be needed!

1) Distributing data in memory across machines is not *as cheap* as accessing memory locally because of **communication costs.**
2) The problem we're dealing with is not **size.** It's the interaction between **size** and **locality:** we have a large structure that's being accessed in a **non-local** way.



## Naïve Bayes for Big Data

- How to implement Naïve Bayes
  - Assuming the event counters do *not* fit in memory
- Possible approaches: O(n*scan) ➔ O(n*scan+n*send)
  - Use a **memory-based distributed** database?
    - Extra cost: Communication costs: O(n) … but that's "ok"
    - Extra complexity: routing requests correctly
  - Compress the counter hash table?
    - Use integers as keys instead of strings?
    - Use approximate counts?
    - Discard infrequent/unhelpful words?
  - Trade off time for space somehow?
    - Observation: if the counter updates were better-ordered we could avoid using disk



## Counting ~~Naïve Bayes~~ for Big Data

- One way trade off time for space:
  - Assume you **need** *K* times as much memory as you actually **have**
  - Method:
    - Construct a hash function *h(event)*
    - For *i=0,…,K-1*:
      - Scan thru the *train* dataset
      - Increment counters for *event* only if *h(event) mod K == i*
      - Save this counter set to disk at the end of the scan
    - After K scans you have a complete counter set
- Comment:
  - this works for *any* counting task, not just Naïve Bayes
  - What we're really doing here is organizing our "messages" to get more locality….



## Project Announcements

- Join a team on connex (GroupProject <num>)
  - everyone from your group should join the same connex group
- Only one person from a group needs to submit the proposal
- You need to join a group to see the project proposal assignment on connex
- Friday: Olav Krigolson will talk about some data his EEG lab is creating – a possible project idea!?

## THE STREAM-AND-SORT DESIGN PATTERN FOR NAIVE BAYES

## Large-vocabulary Naïve Bayes

- Create a hashtable C
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - C("*Y*=ANY") ++;  C("*Y=y*") ++
  - For *j* in *1..d*:
    - C("*Y=y ^ X=*$x_j$") ++

25

## Large-vocabulary Naïve Bayes

- ~~Create a hashtable C~~
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - ~~C("*Y*=ANY") ++;  C("*Y=y*") ++~~
  - Print "Y=ANY += 1"
  - Print "Y=*y* += 1"
  - For *j* in *1..d*:
    - ~~C("*Y=y ^ X=*$x_j$") ++~~
    - Print "*Y=y ^ X=*$x_j$ += 1"
- Sort the event-counter update "messages"
- Scan the sorted messages and compute and output the final counter values

Think of these as "messages" to another component to increment the counters

26

## Large-vocabulary Naïve Bayes

- ~~Create a hashtable C~~
- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - ~~C("*Y*=ANY") ++;  C("*Y=y*") ++~~
  - Print "Y=ANY += 1"
  - Print "Y=*y* += 1"
  - For *j* in *1..d*:
    - ~~C("*Y=y ^ X=*$x_j$") ++~~
    - Print "*Y=y ^ X=*$x_j$ += 1"
- Think of these as update "messages"
  - We will collect together messages about the same counter

27

## Example

- Document with label "business"

  The stock market fell today.
- Will produce counts

```
Y= any                    += 1
Y=business                +=  1
Y=business ^ X =the       +=  1
Y=business ^ X =stock     +=  1
Y=business ^ X =market    +=  1
Y=business ^ X =fell      +=  1
Y=business ^ X =today     +=  1
```

## Processing the Output

- Produce counts for all docs
- Sort the counts

```
Y=business               +=   1
Y=business               +=   1
…
Y=business ^ X =aaa     +=   1
…
Y=business ^ X=zynga    +=   1
Y=sports ^ X=hat        +=   1
Y=sports ^ X=hockey     +=   1
Y=sports ^ X=hockey     +=   1
Y=sports ^ X=hockey     +=   1
…
Y=sports ^ X=hoe        +=   1
…
Y=sports                +=   1
…
```

## Large-vocabulary Naïve Bayes

Scan-and-add:

```
Y=business               +=  1
Y=business               +=  1
…
Y=business ^ X =aaa      +=  1
…
Y=business ^ X=zynga     +=  1
Y=sports ^ X=hat         +=  1
Y=sports ^ X=hockey      +=  1
Y=sports ^ X=hockey      +=  1
Y=sports ^ X=hockey      +=  1
…
Y=sports ^ X=hoe         +=  1
…
Y=sports                 +=  1
…
```

```
Y=sports ^ X=hockey    =3
```

streaming

- previousKey = Null
- sumForPreviousKey = 0
- For each (*event,delta*) in input:
  - If *event*==previousKey
    - sumForPreviousKey += delta
  - Else
    - OutputPreviousKey()
    - previousKey = *event*
    - sumForPreviousKey = *delta*
- OutputPreviousKey()

define OutputPreviousKey():
- If PreviousKey!=Null
  - print PreviousKey,sumForPreviousKey

Accumulating the event counts requires *constant* storage **… as long as the input is sorted.**

30

## How Unix Pipes Work

- Processes are all started at the same time
- Data streaming thru the pipeline is held in a queue: *writer* → *[…queue…]* → *reader*
- If the queue is *full*:
  - the *writing process* is blocked
- If the queue is *empty*:
  - the *reading process* is blocked
- Queues are usually smallish: 64k default

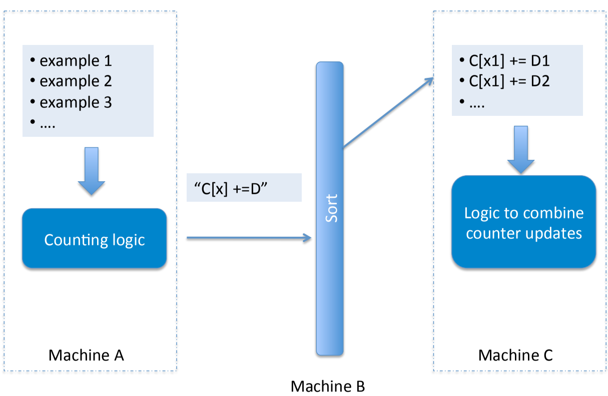


## How stream-and-sort works

- Pipeline is *stream* → *[…queue…]* → *sort*
- Algorithm you get:
  - *sort* reads *--buffer-size* lines in, sorts them, spills them to disk
  - *sort* merges spill files after *stream* closes
  - *stream* is blocked when sort falls behind
  - and *sort* is blocked if it gets ahead

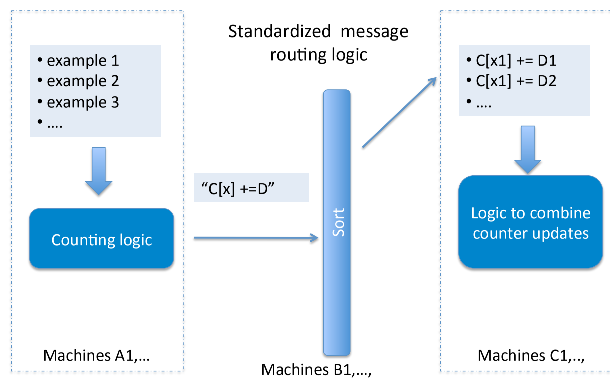


Distributed Counting → Stream and Sort Counting






Stream and Sort Counting → Distributed Counting




Trivial to parallelize!

Easy to parallelize!



## Large-vocabulary Naïve Bayes

- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
  - Print Y=ANY += 1
  - Print Y=*y* += 1
  - For *j* in *1..d*:
    - Print *Y=y ^ X=*$x_j$ += 1
- Sort the event-counter update "messages"
- Scan and add the sorted messages and output the final counter values

Complexity: O(*n*), *n*=size of *train*

(Assuming a constant number of labels apply to each document)

Complexity: O(*nlogn*)

Complexity: O(*n*)

Model size: min( *O(n), O(|V||dom(Y)|)*)



## Using Large-vocabulary Naïve Bayes

- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
- Sort the event-counter update "messages"
- Scan and add the sorted messages and output the final counter values

Model size: max *O(n), O(|V||dom(Y)|)*

- For each example *id, y,* $x_1,\ldots,x_d$ in *test:*
  - For each *y'* in *dom(Y):*
    - Compute log Pr($y', x_1,\ldots,x_d$) =

$$= \left(\sum_j \log \frac{C(X = x_j \wedge Y = y') + mq_x}{C(Y = y') + m}\right) + \log \frac{C(Y = y') + mq_y}{C(Y = ANY) + m}$$

# Using Large-vocabulary Naïve Bayes

- For each example *id, y,* $x_1,\ldots,x_d$ in *train:*
- Sort the event-counter update "messages"
- Scan and add the sorted messages and output the final counter values

Model size: $O(|V|)$

---

- Initialize a HashSet NEEDED and a hashtable C
- For each example *id, y,* $x_1,\ldots,x_d$ in *test:*
  - Add $x_1,\ldots,x_d$ to NEEDED

Time: $O(n_2)$, *size of test*
Memory: same

- For each *event, C(event)* in the summed counters
  - If *event* involves a NEEDED term *x* read it into C

Time: $O(n_2)$
Memory: same

---

- For each example *id, y,* $x_1,\ldots,x_d$ in *test:*
  - For each *y'* in *dom(Y):*
    - Compute log $\Pr(y', x_1,\ldots,x_d) = \ldots$

Time: $O(n_2)$
Memory: same