Summary to date

• Computational complexity: what and how to count
  – Memory vs disk access
  – Cost of scanning vs seeks for disk (and memory)
• Probability review
  – Classification with a “density estimator”
  – Naïve Bayes as a density estimator/classifier
• How to implement Naïve Bayes
  – Time is linear in size of data (one scan!)
  – Assuming the event counters fit in memory
  – We need to count
    \[ C(Y=label), C(X=word ^ Y=label), \ldots \]
Naïve Bayes: Counts in Memory

• You have a *train* dataset and a *test* dataset
• Initialize an “event counter” (hashtable) $C$
• For each example $id, y, x_1, \ldots, x_d$ in *train*:
  – $C(“Y=ANY”) ++$; $C(“Y=y”) ++$
  – For $j$ in $1..d$:
    • $C(“Y=y \land X=x_j”) ++$
    • $C(“Y=y \land X=ANY”) ++$
• For each example $id, y, x_1, \ldots, x_d$ in *test*:
  – For each $y'$ in $\text{dom}(Y)$:
    • Compute $\log \Pr(y', x_1, \ldots, x_d) =$

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

– Return the best $y'$

where:

$q_x = 1/|V|$
$q_y = 1/|\text{dom}(Y)|$
$m=1$
SCALING TO LARGE VOCABULARIES: WHY?
Complexity of Naïve Bayes

- You have a train dataset and a test dataset
- Initialize an “event counter” (hashtable) C
- For each example id, y, x₁,…..,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, x₁,…..,x_d in test:
  - For each y’ in dom(Y):
    • Compute log Pr(y’,x₁,…..,x_d) =

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

- Return the best y’

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

Sequential reads

Complexity: O(\(|\text{dom}(Y)| \times n’\)), n’=size of test

Sequential reads

Assume hashtable holding all counts fits in memory

where:

\[q_x = 1/|V|\]
\[q_y = 1/|\text{dom}(Y)|\]
\[mq_x=1\]
The Naïve Bayes classifier – v1

• Dataset: each example has
  – A unique id \( id \)
    • Why? For debugging the feature extractor
  – \( d \) attributes \( X_1, \ldots, X_d \)
    • Each \( X_i \) takes a discrete value in \( \text{dom}(X_i) \)
  – One class label \( Y \) in \( \text{dom}(Y) \)

• You have a \textit{train} dataset and a \textit{test} dataset

• Assume:
  – the dataset doesn’t fit in memory
  – the model doesn’t either
What's next

• How to implement Naïve Bayes
  – Assuming the event counters do not fit in memory

• Why?

Micro: 0.5G memory
0.00652/hr

Standard:
S: 2Gb
$0.03/hr
XL: 8Gb
$0.104/hr
10xlarge: 160Gb
$2.34/hr
x1.32xlarge: 2Tb, 128 cores
$13.33/hr
What’s next

• How to implement Naïve Bayes
  – Assuming the event counters do not fit in memory
• Why?
  – Zipf’s law: many words that you see, you don’t see often.
<table>
<thead>
<tr>
<th>Number of Occurrences (n)</th>
<th>Predicted Proportion of Occurrences (1/n(n+1))</th>
<th>Actual Proportion occurring n times (L_n/D)</th>
<th>Actual Number of Words occurring n times</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.500</td>
<td>.402</td>
<td>204,357</td>
</tr>
<tr>
<td>2</td>
<td>.167</td>
<td>.132</td>
<td>67,082</td>
</tr>
<tr>
<td>3</td>
<td>.083</td>
<td>.069</td>
<td>35,083</td>
</tr>
<tr>
<td>4</td>
<td>.050</td>
<td>.046</td>
<td>23,271</td>
</tr>
<tr>
<td>5</td>
<td>.033</td>
<td>.032</td>
<td>16,332</td>
</tr>
<tr>
<td>6</td>
<td>.024</td>
<td>.024</td>
<td>12,421</td>
</tr>
<tr>
<td>7</td>
<td>.018</td>
<td>.019</td>
<td>9,766</td>
</tr>
<tr>
<td>8</td>
<td>.014</td>
<td>.016</td>
<td>8,200</td>
</tr>
<tr>
<td>9</td>
<td>.011</td>
<td>.014</td>
<td>6,907</td>
</tr>
<tr>
<td>10</td>
<td>.009</td>
<td>.012</td>
<td>5,893</td>
</tr>
</tbody>
</table>

Frequencies from 336,310 documents in the 1GB TREC Volume 3 Corpus 125,720,891 total word occurrences; 508,209 unique words
What’s next

- 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:
  - $K \approx 1/10 \quad 1/100$
  - $\beta \approx 0.4–0.6$ (approx. square-root)
- Why?
  - Proper names, misspellings, 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, …)
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)
# Numbers (Jeff Dean says) Everyone Should Know

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>100</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>10,000</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from network</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>30,000,000</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>
A single large file can be spread out among many non-adjacent blocks/sectors...

and then you need to seek around to scan the contents of the file...
What’s next

• How to implement Naïve Bayes
  – Assuming the event counters do *not* fit in memory
• Possible approaches:
  – Use a database?
    • 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 …..
What’s next

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

• example 1
• example 2
• example 3
• ....

“increment $C[x]$ by $D$”

Hash table, database, etc

Counting logic
Counting

- example 1
- example 2
- example 3
- ....

Counting logic

"increment C[x] by D"

Hash table, database, etc

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

- example 1
- example 2
- example 3
- ....

Counting logic

```
Hash table1
```

Machine 1

```
Hash table2
```

Machine 2

```
Hash table2
```

Machine K

```
increment C[x] by D```

Now we have enough memory....
Distributed Counting

Counting logic

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

example 1
example 2
example 3
....

Machine 0

Hash table1

Machine 1

Hash table2

Machine 2

Machine K
# Numbers (Jeff Dean says) Everyone Should Know

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
<th>Approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
<td></td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
<td></td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
<td>~= 10x</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>100 ns</td>
<td>~= 15x</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
<td></td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>10,000 ns</td>
<td></td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000 ns</td>
<td></td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000 ns</td>
<td></td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000 ns</td>
<td></td>
</tr>
<tr>
<td>Disk seek</td>
<td></td>
<td>40x</td>
</tr>
<tr>
<td>Read 1 MB sequentially from network</td>
<td>10,000,000ns</td>
<td>~= 100,000x</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>30,000,000ns</td>
<td></td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000ns</td>
<td></td>
</tr>
</tbody>
</table>
What’s next

• How to implement Naïve Bayes
  – Assuming the event counters do not fit in memory
• Possible approaches:
  – 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.
What’s next

• How to implement Naïve Bayes
  – **Assuming** the event counters do *not* fit in memory
• Possible approaches:
  – 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

\[ O(n*\text{scan}) \Rightarrow O(n*\text{scan}+n*\text{send}) \]
Large-vocabulary Naïve Bayes Counting

• 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 \(i=0,\ldots,K-1\):
      – 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…. 
HOW TO ORGANIZE DATA TO ENABLE LARGE-SCALE COUNTING
Large vocabulary counting

• Another approach:
  – Start with
    • Q: “what can we do for large sets quickly”? 
    • A: sorting
      – It’s $O(n \log n)$, not much worse than linear
      – You can do it for very large datasets using a merge sort
        » sort $k$ subsets that fit in memory,
        » merge results, which can be done in linear time
Alternative visualization
ASIDE: MORE ON SORTING
Bottom-Up Merge Sort

use: input array $A[n]$; buffer array $B[n]$

• assert: $A[\ ]$ contains sorted runs of length $r=1$
• for run-length $r=1,2,4,8,…$
  • merge adjacent length-$r$ runs in $A[\ ]$, copying the result into the buffer $B[\ ]$
  • assert: $B[\ ]$ contains sorted runs of length $2*r$
• swap roles of A and B
BottomUpMerge(int A[], int iLeft, int iRight, int iEnd, int B[])
{
    int i0 = iLeft;
    int i1 = iRight;
    int j;

    /* While there are elements in the left or right lists */
    for (j = iLeft; j < iEnd; j++)
    {
        /* If left list head exists and is <= existing right list head */
        if (i0 < iRight && (i1 >= iEnd || A[i0] <= A[i1]))
        {
            B[j] = A[i0];
            i0 = i0 + 1;
        }
        else
        {
            B[j] = A[i1];
            i1 = i1 + 1;
        }
    }
}
Wikipedia on Old-School Merge Sort

Use four tape drives A,B,C,D

1. merge runs from A,B and write them alternately into C,D

2. merge runs from C,D and write them alternately into A,B

3. And so on....

Requires only constant memory.
Unix Sort

- Load as *much as you can* [actually --buffer-size=SIZE] into memory and do an *in-memory sort* [usually quicksort].

- If you have more to do, then spill this sorted buffer out on to disk, and get a another buffer’s worth of data.

- Finally, merge your spill buffers.
SORTING OUT OF MEMORY WITH PIPES

genenerate lines | sort | process lines
How Unix Pipes Work

- Processes are all started at the same time
- Data streaming thru the pipeline is held in a queue: `writer \rightarrow [...] \rightarrow reader`
- If the queue is full:
  - the *writing process* is blocked
- If the queue is empty:
  - the *reading process* is blocked
- (I think) queues are usually smallish: 64k
How stream-and-sort works

• Pipeline is stream $\rightarrow […]queue[…] \rightarrow sort$

• 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
THE STREAM-AND-SORT DESIGN PATTERN FOR NAIVE BAYES
Large-vocabulary Naïve Bayes

• Create a hashtable C
• For each example $id, y, x_1, ..., x_d$ in train:
  – $C(“Y=ANY”)++; C(“Y=y”)++$
  – For $j$ in 1..$d$:
    • $C(“Y=y \land X=x_j”)++$
Large-vocabulary Naïve Bayes

• Create a hashtable \( C \)
• For each example \( id, y, x_1, \ldots, x_d \) in \( train \):
  – \( C(\text{"Y=ANY"}) \) ++; \( C(\text{"Y=y"}) \) ++
  – Print “\( Y=\text{ANY} \) += 1”
  – Print “\( Y=y \) += 1”
  – For \( j \) in 1..d:
    • \( C(\text{"Y=y ^ X=x_j"}) \) ++
    • 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

```
python MyTrainer.py train | sort | python MyCountAdder.py > model
```
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”
  
  – We’re collecting together messages about the same counter

• Scan and add the sorted messages and output the final counter values
Large-vocabulary Naïve Bayes

Scan-and-add:

\[
\begin{align*}
Y=\text{business} & + = 1 \\
Y=\text{business} & + = 1 \\
\cdots & \\
Y=\text{business} \land X=\text{aaa} & + = 1 \\
\cdots & \\
Y=\text{business} \land X=\text{zynga} & + = 1 \\
Y=\text{sports} \land X=\text{hat} & + = 1 \\
Y=\text{sports} \land X=\text{hockey} & + = 1 \\
Y=\text{sports} \land X=\text{hockey} & + = 1 \\
\cdots & \\
Y=\text{sports} \land X=\text{hoe} & + = 1 \\
\cdots & \\
Y=\text{sports} & + = 1 \\
\cdots & 
\end{align*}
\]

\[
\text{previousKey} = \text{Null} \\
\text{sumForPreviousKey} = 0
\]

For each \((\text{event}, \text{delta})\) in input:
- If \(\text{event}==\text{previousKey}\)
  - \(\text{sumForPreviousKey} += \text{delta}\)
- Else
  - \(\text{OutputPreviousKey}()\)
  - \(\text{previousKey} = \text{event}\)
  - \(\text{sumForPreviousKey} = \text{delta}\)
- \(\text{OutputPreviousKey}()\)

\[
\text{define OutputPreviousKey}():
\begin{align*}
\text{If PreviousKey}!==\text{Null} & \\
\text{print PreviousKey}, \text{sumForPreviousKey}
\end{align*}
\]

Accumulating the event counts requires constant storage … as long as the input is sorted.
Distributed Counting → Stream and Sort Counting

- example 1
- example 2
- example 3
- ....

Counting logic

Machine 0

Message-routing logic

"C[x] +=D"

Hash table1

Machine 1

Hash table2

Machine 2

... 

Hash table2

Machine K
Distributed Counting → Stream and Sort Counting

- example 1
- example 2
- example 3
- ....

```
C[x] += D
```

Machine A

```
C[x1] += D1
C[x1] += D2
....
```

Logic to combine counter updates

Machine C

```
"C[x] += D"
```

Counting logic

Machine B
Stream and Sort Counting  →  Distributed Counting

- example 1
- example 2
- example 3
- ....

Counting logic

Machines A1, ...

"C[x] += D"

Sort

Machines B1, ...

Standardized message routing logic

- C[x1] += D1
- C[x1] += D2
- ....

Logic to combine counter updates

Machines C1, ...

Easy to parallelize!

Trivial to parallelize!
Locality is good

**Micro:**
- 0.6G memory

**Standard:**
- S: 1.7Gb
- L: 7.5Gb
- XL: 15Mb

**Hi Memory:**
- XXL: 34.2
- XXXXL: 68.4

<table>
<thead>
<tr>
<th>Region: US East (Virginia)</th>
<th>Linux/UNIX Usage</th>
<th>Windows Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard On-Demand Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Small (Default)</td>
<td>$0.085 per hour</td>
<td>$0.12 per hour</td>
</tr>
<tr>
<td>Large</td>
<td>$0.34 per hour</td>
<td>$0.48 per hour</td>
</tr>
<tr>
<td>Extra Large</td>
<td>$0.68 per hour</td>
<td>$0.96 per hour</td>
</tr>
<tr>
<td><strong>Micro On-Demand Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Micro</td>
<td>$0.02 per hour</td>
<td>$0.03 per hour</td>
</tr>
<tr>
<td><strong>Hi-Memory On-Demand Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extra Large</td>
<td>$0.50 per hour</td>
<td>$0.62 per hour</td>
</tr>
<tr>
<td>Double Extra Large</td>
<td>$1.00 per hour</td>
<td>$1.24 per hour</td>
</tr>
<tr>
<td>Quadruple Extra Large</td>
<td>$2.00 per hour</td>
<td>$2.48 per hour</td>
</tr>
<tr>
<td><strong>Hi-CPU On-Demand Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>$0.17 per hour</td>
<td>$0.29 per hour</td>
</tr>
<tr>
<td>Extra Large</td>
<td>$0.68 per hour</td>
<td>$1.16 per hour</td>
</tr>
<tr>
<td><strong>Cluster Compute Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quadruple Extra Large</td>
<td>$1.30 per hour</td>
<td>$1.61 per hour</td>
</tr>
<tr>
<td>Eight Extra Large</td>
<td>$2.40 per hour</td>
<td>$2.97 per hour</td>
</tr>
<tr>
<td><strong>Cluster GPU Instances</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quadruple Extra Large</td>
<td>$2.10 per hour</td>
<td>$2.60 per hour</td>
</tr>
</tbody>
</table>
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 $Y=y \land X=x_j ++ 1$

• Sort the event-counter update “messages”

• Scan and add the sorted messages and output the final counter values

```
python MyTrainer.py train | sort | python MyCountAdder.py > model
```

Model size: $\min(O(n), O(|V| \mid dom(Y)|))$
STREAM-AND-SORT + LOCAL PARTIAL COUNTING
Today

• Naïve Bayes with huge feature sets
  – i.e. ones that don’t fit in memory
• Pros and cons of possible approaches
  – Traditional “DB” (actually, key-value store)
  – Memory-based distributed distributed DB
  – Stream-and-sort counting
• Optimizations
• Other tasks for stream-and-sort
**Optimizations**

```java
MyTrainer.train | sort | MyCountAdder > model
```

- $O(n)$ Input size=n
  - Output size=n
- $O(n\log n)$ Input size=n
  - Output size=n
- $O(n)$ Input size=n
  - Output size=m
  - $m<<n \ldots$ say $O(\sqrt{n})$

A useful optimization:
- Decrease the size of the input to the sort
- Reduces the size from $O(n)$ to $O(m)$

1. Compress the output by using simpler messages
   - (“$C[event] += 1$”) $\rightarrow$ “event 1”
2. Compress the output more – e.g. string $\rightarrow$ integer code
   - Tradeoff – ease of debugging vs efficiency – are messages meaningful or meaningful in context?
Optimization: partial local counting

- For each example \( id, y, x_1, \ldots, x_d \) in \( \text{train} \):
  - Print “\( Y=y \) += 1”
  - For \( j \) in \( 1..d \):
    - Print “\( Y=y \land X=x_j \) += 1”
- Sort the event-counter update “messages”
- Scan and add the sorted messages and output the final counter values

- Initialize hashtable \( C \)
- For each example \( id, y, x_1, \ldots, x_d \) in \( \text{train} \):
  - \( C[Y=y] \) += 1
  - For \( j \) in \( 1..d \):
    - \( C[Y=y \land X=x_j] \) += 1
  - If memory is getting full: output all values from \( C \) as messages and re-initialize \( C \)
- Sort the event-counter update “messages”
- Scan and add the sorted messages

```bash
python MyTrainer.py train | sort | python MyCountAdder.py > model
```
Review: Large-vocab Naïve Bayes

- Create a hashtable C
- For each example \(id, y, x_1, \ldots, x_d\) in train:
  - \(C\.inc(\text{"Y=\textit{y}\"})\)
  - For \(j\) in \(1..d\):
    - \(C\.inc(\text{"Y=\textit{y} \wedge X=\textit{x}_j\"})\)

```python
class EventCounter(object):
    def __init__(self):
        self._ctr = {}
    def inc(self, event):
        // increment the counter for 'event'
        if (len(self._ctr) > BUFFER_SIZE):
            for (e,n) in self._ctr.items(): print '%s\t%d' % (e,n)
        // clear self._ctr
```

Distributed Counting → Stream and Sort Counting

Machine A

• example 1
• example 2
• example 3
• ....

Counting logic

“C[x] += D”

Machine B

BUFFER

Machine C

• C[x1] += D1
• C[x1] += D2
• ....

Logic to combine counter updates
How much does buffering help?

<table>
<thead>
<tr>
<th>BUFFER_SIZE</th>
<th>Time</th>
<th>Message Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td>1.7M words</td>
</tr>
<tr>
<td>100</td>
<td>47s</td>
<td>1.2M</td>
</tr>
<tr>
<td>1,000</td>
<td>42s</td>
<td>1.0M</td>
</tr>
<tr>
<td>10,000</td>
<td>30s</td>
<td>0.7M</td>
</tr>
<tr>
<td>100,000</td>
<td>16s</td>
<td>0.24M</td>
</tr>
<tr>
<td>1,000,000</td>
<td>13s</td>
<td>0.16M</td>
</tr>
<tr>
<td>limit</td>
<td></td>
<td>0.05M</td>
</tr>
</tbody>
</table>
CONFESSION:
THIS NAÏVE BAYES HAS A PROBLEM....
Today

• Naïve Bayes with huge feature sets
  – i.e. ones that don’t fit in memory
• Pros and cons of possible approaches
  – Traditional “DB” (actually, key-value store)
  – Memory-based distributed DB
  – Stream-and-sort counting
• Optimizations
• Other tasks for stream-and-sort
• Finally: A “detail” about large-vocabulary Naïve Bayes.....
Complexity of Naïve Bayes

- You have a train dataset and a test dataset
- Initialize an “event counter” (hashtable) $C$
- For each example $id, y, x_1, ..., x_d$ in train:
  - $C(\text{"Y=y"})$ ++
  - For $j$ in $1..d$:
    - $C(\text{"Y=y \land X=x_j"})$ ++
    - ....
- For each example $id, y, x_1, ..., x_d$ in test:
  - For each $y'$ in $\text{dom}(Y)$:
    - Compute $\log \text{Pr}(y', x_1, ..., x_d) =$
      
      \[
      \left( \sum_j \log \frac{C(X = x_j \land Y = y') + mq_x}{C(X = \text{ANY} \land Y = y') + m} \right) + \log \frac{C(Y = y') + mq_y}{C(Y = \text{ANY}) + m}
      \]
  - Return the best $y'$

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

Sequential reads

Complexity: $O(|\text{dom}(Y)| * n')$, $n'=$ size of test

where:
- $q_j = 1/|V|
- q_y = 1/|\text{dom}(Y)|$
- $mq_x=1$
Using Large-vocabulary Naïve Bayes

- For each example \( id, y, x_1, \ldots, x_d \) in \( \text{train} \):
  - Sort the event-counter update “messages”
  - Scan and add the sorted messages and output the final counter values

Model size: \( \max O(n), O(|V| |\text{dom}(Y)|) \)

- For each example \( id, y, x_1, \ldots, x_d \) in \( \text{test} \):
  - For each \( y' \) in \( \text{dom}(Y) \):
    - Compute \( \log \Pr(y', x_1, \ldots, x_d) = \) 
      \[
      \left( \sum_j \log \frac{C(X = x_j \land Y = y') + mq_x}{C(Y = y') + m} \right) + \log \frac{C(Y = y') + mq_y}{C(Y = \text{ANY}) + m}
      \]
Using Large-vocabulary Naïve Bayes

- For each example id, y, x₁,…..,xₐ in train:
  - Sort the event-counter update “messages”
  - Scan and add the sorted messages and output the final counter values

- Initialize a HashSet NEEDED and a hashtable C
- For each example id, y, x₁,…..,xₐ in test:
  - Add x₁,…..,xₐ to NEEDED

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

- For each example id, y, x₁,…..,xₐ in test:
  - For each y’ in dom(Y):
    - Compute log Pr(y’,x₁,…..,xₐ) = ....

Model size: O(|V|)

Time: O(n²), size of test
Memory: same
Large-Vocabulary Naïve Bayes

**Learning/Counting**

- Counts on disk with a key-value store
- Counts as messages to a set of distributed processes
- Repeated scans to build up partial counts
- Counts as messages in a stream-and-sort system
- **Assignment:** Counts as messages but buffered in memory

**Using Counts**

- **Assignment:**
  - Scan through counts to find those needed for test set
  - Classify with counts in memory
- Put counts in a database
- Use partial counts and repeated scans of the test data?
- Re-organize the counts and test set so that you can classify in a stream
MORE STREAM-AND-SORT EXAMPLES
Some other stream and sort tasks

• Coming up: classify Wikipedia pages
  – Features:
    • words on page: $src \, w_1 \, w_2 \, ...$
    • outlinks from page: $src \, dst_1 \, dst_2 \, ...$
    • how about inlinks to the page?
Some other stream and sort tasks

- outlinks from page: $src \ dst_1 \ dst_2 \ ...$

  - Algorithm:
    - For each input line $src \ dst_1 \ dst_2 \ ... \ dst_n$ print out
      - $dst_1$ inlinks. = $src$
      - $dst_2$ inlinks. = $src$
      - ...
      - $dst_n$ inlinks. = $src$
    - Sort this output
    - Collect the messages and group to get
      - $dst \ src_1 \ src_2 \ ... \ src_n$
Some other stream and sort tasks

- prevKey = Null
- sumForPrevKey = 0
- For each (event += delta) in input:
  - If event==prevKey
    - sumForPrevKey += delta
  - Else
    - OutputPrevKey()
    - prevKey = event
    - sumForPrevKey = delta
- OutputPrevKey()

define OutputPrevKey():
- If PrevKey!=Null
  - print PrevKey,sumForPrevKey

- prevKey = Null
- linksToPrevKey = []
- For each (dst inlinks.= src) in input:
  - If dst==prevKey
    - linksPrevKey.append(src)
  - Else
    - OutputPrevKey()
    - prevKey = dst
    - linksToPrevKey=[src]
- OutputPrevKey()

define OutputPrevKey():
- If PrevKey!=Null
  - print PrevKey, linksToPrevKey
Some other stream and sort tasks

• What if we run this same program on the words on a page?
  – Features:
    • words on page: src $w_1$ $w_2$ ….
    • outlinks from page: src $dst_1$ $dst_2$ …

Out2In.java

an inverted index for the documents
Some other stream and sort tasks

- outlinks from page:  \texttt{src dst}_1 \texttt{dst}_2 \ldots

  - Algorithm:
    - For each input line \texttt{src dst}_1 \texttt{dst}_2 \ldots \texttt{dst}_n print out
      - \texttt{dst}_1 \text{ inlinks} = \texttt{src}
      - \texttt{dst}_2 \text{ inlinks} = \texttt{src}
      - \ldots
      - \texttt{dst}_n \text{ inlinks} = \texttt{src}
    - Sort this output
    - Collect the messages and group to get
      - \texttt{dst \ src}_1 \texttt{src}_2 \ldots \texttt{src}_n
Some other stream and sort tasks

• Later on: distributional clustering of words
Some other stream and sort tasks

• Later on: distributional clustering of words

Algorithm:
• For each word $w$ in a corpus print $w$ and the words in a window around it
  – Print “$w_i$ context $= (w_{i-k}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+k})$”
• Sort the messages and collect all contexts for each $w$ – thus creating an instance associated with $w$
• Cluster the dataset
  – Or train a classifier and classify it
Some other stream and sort tasks

- \( \text{prevKey} = \text{Null} \)
- \( \text{sumForPrevKey} = 0 \)
- For each \((\text{event} += \text{delta})\) in input:
  - If \(\text{event} == \text{prevKey}\)
    - \(\text{sumForPrevKey} += \text{delta}\)
  - Else
    - OutputPrevKey()
    - \(\text{prevKey} = \text{event}\)
    - \(\text{sumForPrevKey} = \text{delta}\)
    - OutputPrevKey()

Define OutputPrevKey():
- If PrevKey!=Null
  - Print PrevKey, sumForPrevKey

- \( \text{prevKey} = \text{Null} \)
- \( \text{ctxOfPrevKey} = [\ ] \)
- For each \((w \ c. = w_1, \ldots, w_k)\) in input:
  - If \(\text{dst} == \text{prevKey}\)
    - \(\text{ctxOfPrevKey}.append(\ w_1, \ldots, w_k)\)
  - Else
    - OutputPrevKey()
    - \(\text{prevKey} = w\)
    - \(\text{ctxOfPrevKey} = [w_1, \ldots, w_k]\)
    - OutputPrevKey()

Define OutputPrevKey():
- If PrevKey!=Null
  - Print PrevKey, ctxOfPrevKey
Some other stream and sort tasks

- Finding unambiguous geographical names
- GeoNames.org: for each place in its database, stores
  - Several alternative names
  - Latitude/Longitude
  - ...
- Lets you put places on a map (e.g., Google Maps)
- Problem: many names are ambiguous, especially if you allow an approximate match
  - Paris, London, ... even Carnegie Mellon
Point Park
(College | University)

Carnegie Mellon
[University [School]]
Some other stream and sort tasks

- Finding almost unambiguous geographical names
- GeoNames.org: for each place in the database
  - print all plausible soft-match substrings in each alternative name, paired with the lat/long, e.g.
    - Carnegie Mellon University at lat1,lon1
    - Carnegie Mellon at lat1,lon1
    - Mellon University at lat1,lon1
    - Carnegie Mellon School at lat2,lon2
    - Carnegie Mellon at lat2,lon2
    - Mellon School at lat2,lon2
    - ...
  - Sort and collect... and filter
Some other stream and sort tasks

- prevKey = Null
- sumForPrevKey = 0
- For each \((event += \text{delta})\) in input:
  - If \(event==\text{prevKey}\)
    - sumForPrevKey += delta
  - Else
    - OutputPrevKey()
    - prevKey = event
    - sumForPrevKey = delta

- OutputPrevKey()

define OutputPrevKey():
- If PrevKey!=Null
  - print PrevKey,sumForPrevKey

- prevKey = Null
- locOfPrevKey = Gaussian()
- For each \((place \text{ at lat,lon})\) in input:
  - If dst==prevKey
    - locOfPrevKey.observe(lat, lon)
  - Else
    - OutputPrevKey()
    - prevKey = place
    - locOfPrevKey = Gaussian()
    - locOfPrevKey.observe(lat, lon)
  - OutputPrevKey()

define OutputPrevKey():
- If PrevKey!=Null and locOfPrevKey.stdDev() < 1 mile
  - print PrevKey, locOfPrevKey.avg()