Recap
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
    $C(Y=label)$, $C(X=word \land Y=label)$,…
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, x₁,…..,xₙ in train:
  - C(“Y=ANY”) ++;  C(“Y=y”) ++
  - For j in 1..d:
    - C(“Y=y ^ X=xⱼ”) ++
    - C(“Y=y ^ X=ANY”) ++
- For each example id, y, x₁,…..,xₙ in test:
  - For each y’ in dom(Y):
    - Compute log Pr(y’,x₁,…..,xₙ) =

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

- Return the best y’
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 \( \log \Pr(y’,x₁,...,x_d) = \)
      \[
      \left( \sum_j \log \frac{C(X = x_j ∧ Y = y') + mq_x}{C(X = ANY ∧ Y = y') + m} \right) + \log \frac{C(Y = y') + mq_y}{C(Y = ANY) + m}
      \]
  - Return the best y’

Assume hashtable holding all counts fits in memory

Sequential reads

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

where:

\[
q_x = \frac{1}{|V|}
\]
\[
q_y = \frac{1}{|\text{dom}(Y)|}
\]
\[
mq_x = 1
\]

Sequential reads

Complexity: O(\(|\text{dom}(Y)| \times n’\), n’=size of test
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 *train* dataset and a *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
  $0.00652/\text{hr}$

Standard:
  S: 2Gb
  $0.03/\text{hr}$
  XL: 8Gb
  $0.104/\text{hr}$
  10xlarge: 160Gb
  $2.34/\text{hr}$
  x1.32xlarge: 2Tb, 128 cores
  $13.33/\text{hr}$
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:
  – $K \approx 1/10 \ - \ 1/100$
  – $\beta \approx 0.4-0.6$ (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, …)
Alternatives….
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>Operation</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>

Approximations:
- L2 cache reference to L2 cache reference: \(~= 10x\)
- Main memory reference to Main memory reference: \(~= 15x\)
- Disk seek to Round trip within same datacenter: \(~= 40x\)
- Disk seek to Send packet CA->Netherlands->CA: \(~= 100,000x\)
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 …..

\[ O(n*\text{scan}) \Rightarrow O(n*\text{scan}*\text{seek}) \]
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 …..

\[ O(n^{\text{\scriptsize scan}}) \rightarrow O(n^{\text{\scriptsize scan}^{??}}) \]
Counting

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

"increment \( C[x] \) by \( D \)"

Counting logic

Hash table, database, etc
Counting

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

"increment C[x] by D"

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

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

Counting logic

Now we have enough memory....

```
Hash table1
Machine 1

Hash table2
Machine 2

...  

Hash table2
Machine K
```
Distributed Counting

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

Counting logic

"increment C[x] by D"

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>Relative Speed</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></td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>100 ns</td>
<td></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>10,000,000 ns</td>
<td></td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>30,000,000 ns</td>
<td></td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000 ns</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

\[ O(n*\text{scan}) \rightarrow O(n*\text{scan} + n*\text{send}) \]

Great ideas which we’ll discuss more later
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

generate 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 [...queue...] \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 $stream \rightarrow [\ldots queue\ldots] \rightarrow sort$
• 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, x₁,…..,xₖ in train:
  – C("Y=ANY") ++; C("Y=y") ++
  – For j in 1..d:
    • C("Y=y ^ X=x_{j}\) ++
Large-vocabulary Naïve Bayes

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

`java MyTrainer train | sort | java MyCountAdder > 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:

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

Accumulating the event counts requires \textit{constant} storage \ldots 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
- ....

```
\text{C}[x] += D
```

Machine A

```
\text{C}[x1] += D1
\text{C}[x1] += D2
\text{.....}
```

Logic to combine counter updates

Machine B

Machine C
Stream and Sort Counting → Distributed Counting

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

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

Machines A1, ...

Counting logic

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

Logic to combine counter updates

Machines C1, ...

Easy to parallelize!

Machines B1, ...

Sort

Standardized message routing logic

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</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 \( Y=\text{ANY} \) += 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

\[
\text{Model size: } \min(O(n), O(|V| \cdot |\text{dom}(Y)|))
\]
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 DB
  – Stream-and-sort counting
• Optimizations
• Other tasks for stream-and-sort
Optimizations

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

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(n)</td>
<td>Input size=n Output size=n</td>
</tr>
<tr>
<td>O(nlogn)</td>
<td>Input size=n Output size=n</td>
</tr>
<tr>
<td>O(n)</td>
<td>Input size=n Output size=m</td>
</tr>
</tbody>
</table>

m<<n ... say O(sqrt(n))

A useful optimization: decrease the size of the input to the sort

1. Compress the output by using simpler messages
   ("C[event] ++ 1") → "event 1"
2. Compress the output more – e.g. string → 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 \(train:\)
  - 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 \(train:\)
  - \(C[Y=y] \, += \, 1\)
  - 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

```
java MyTrainer \| sort \| java MyCountAdder > model
```
Review: Large-vocab Naïve Bayes

- Create a hashtable $C$
- For each example $id, y, x_1, ..., x_d$ in $train$:
  - $C.inc(”Y=y”)$
  - For $j$ in $1..d$:
    - $C.inc(”Y=y ^ X=x_j”)$

class EventCounter {
    void inc(String event) {
        // increment the right hashtable slot
        if (hashtable.size() > BUFFER_SIZE) {
            for (e,n) in hashtable.entries : print e + “\t” + n
            hashtable.clear();
        }
    }
}

Distributed Counting → Stream and Sort Counting

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

Counting logic

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

Machine A

Machine B

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

Logic to combine counter updates

Machine C
How much does buffering help?

small-events.txt: nb.jar
  time java -cp nb.jar com.wcohen.SmallStreamNB < RCV1.small_train.txt \\
  | sort -k1,1 \\
  | java -cp nb.jar com.wcohen.StreamSumReducer> small-events.txt

test-small: small-events.txt nb.jar
  time java -cp nb.jar com.wcohen.SmallStreamNB \\
  RCV1.small_test.txt MCAT,CCAT,GCAT,ECAT 2000 < small-events.txt \\
  | cut -f3 | sort | uniq -c

<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>
MORE STREAM-AND-SORT EXAMPLES
Some other stream and sort tasks

• Coming up: classify Wikipedia pages
  – Features:
    • words on page: $src \ w_1 \ w_2 \ldots$
    • outlinks from page: $src \ dst_1 \ dst_2 \ldots$
    • 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: $src \, w_1 \, w_2 \, \ldots$
    • outlinks from page: $src \, dst_1 \, dst_2 \, \ldots$

an inverted index for the documents

Out2In.java

\[
\begin{align*}
  w_1 \, src_{1,1} \, src_{1,2} \, src_{1,3} \, \ldots \\
  w_2 \, src_{2,1} \, \ldots \\
  \ldots
\end{align*}
\]
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

- Later on: distributional clustering of words

<table>
<thead>
<tr>
<th>duty</th>
<th>responsibility 0.21 0.21</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>role 0.12 0.11</td>
</tr>
<tr>
<td></td>
<td>action 0.11 0.10</td>
</tr>
<tr>
<td></td>
<td>change 0.24 0.08</td>
</tr>
<tr>
<td></td>
<td>rule 0.16 0.08</td>
</tr>
<tr>
<td></td>
<td>restriction 0.27 0.08</td>
</tr>
<tr>
<td></td>
<td>ban 0.30 0.08</td>
</tr>
<tr>
<td></td>
<td>sanction 0.19 0.08</td>
</tr>
<tr>
<td></td>
<td>schedule 0.11 0.07</td>
</tr>
<tr>
<td></td>
<td>regulation 0.37 0.07</td>
</tr>
<tr>
<td></td>
<td>challenge 0.13 0.07</td>
</tr>
<tr>
<td></td>
<td>issue 0.13 0.07</td>
</tr>
<tr>
<td></td>
<td>reason 0.14 0.07</td>
</tr>
<tr>
<td></td>
<td>matter 0.28 0.07</td>
</tr>
<tr>
<td></td>
<td>measure 0.22 0.07</td>
</tr>
<tr>
<td></td>
<td>obligation 0.12 0.10</td>
</tr>
<tr>
<td></td>
<td>power 0.17 0.08</td>
</tr>
<tr>
<td></td>
<td>jurisdiction 0.13 0.08</td>
</tr>
<tr>
<td></td>
<td>right 0.12 0.07</td>
</tr>
<tr>
<td></td>
<td>control 0.20 0.07</td>
</tr>
<tr>
<td></td>
<td>ground 0.08 0.07</td>
</tr>
<tr>
<td></td>
<td>accountability 0.14 0.08</td>
</tr>
<tr>
<td></td>
<td>experience 0.12 0.07</td>
</tr>
<tr>
<td></td>
<td>post 0.14 0.14</td>
</tr>
<tr>
<td></td>
<td>task 0.10 0.10</td>
</tr>
<tr>
<td></td>
<td>chore 0.11 0.07</td>
</tr>
<tr>
<td></td>
<td>operation 0.10 0.10</td>
</tr>
<tr>
<td></td>
<td>function 0.10 0.08</td>
</tr>
<tr>
<td></td>
<td>mission 0.12 0.07</td>
</tr>
<tr>
<td></td>
<td>patrol 0.07 0.07</td>
</tr>
<tr>
<td></td>
<td>staff 0.10 0.07</td>
</tr>
<tr>
<td></td>
<td>penalty 0.09 0.09</td>
</tr>
<tr>
<td></td>
<td>fee 0.17 0.08</td>
</tr>
<tr>
<td></td>
<td>tariff 0.13 0.08</td>
</tr>
<tr>
<td></td>
<td>tax 0.19 0.07</td>
</tr>
<tr>
<td></td>
<td>reservist 0.07 0.07</td>
</tr>
</tbody>
</table>
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

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

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

• prevKey = Null
• ctxOfPrevKey = [ ]
• For each \((w \ c.= \ w_1,\ldots,\ w_k)\) in input:
  • If \(dst==\text{prevKey}\)
    • ctxOfPrevKey.append(\(w_1,\ldots,\ w_k\))
  • Else
    • OutputPrevKey()
    • prevKey = \text{w}
    • ctxOfPrevKey=[\text{w}_1,\ldots,\text{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
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 += 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
• locOfPrevKey = Gaussian()
• For each (place 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()
LOOKING AHEAD: PARALLELIZING STREAM AND SORT
Stream and Sort Counting → Distributed Counting

Standardized message routing logic

Machines A1,…

“C[x] +=D”

Sort

Machines B1,…,

Machines C1,…

• C[x1] += D1
• C[x1] += D2
• ….

Logic to combine counter updates

Trivial to parallelize!

Easy to parallelize!
Stream and Sort Counting → Distributed Counting

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

Counting logic

“C[x] += D”

Sort

Machines A1, ...

Spill 1

Spill 2

Spill 3

Merge Spill Files

Logic to combine counter updates

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

Machines C1, ..,
Stream and Sort Counting → Distributed Counting

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

Counting logic

“C[x] += D”

Counter Machine

Sort

Spill 1

Spill 2

Spill 3

Merge Spill Files

Logic to combine counter updates

Combiner Machine

- C[x1] += D1
- C[x1] += D2
- ....
Stream and Sort Counting → Distributed Counting

**Counter Machine 1**
- Example 1
- Example 2
- Example 3
- ...

**Counter Machine 2**
- Example 1
- Example 2
- Example 3
- ...

**Combiner Machine 1**
- \( C[x1] += D1 \)
- \( C[x1] += D2 \)
- ...

**Combiner Machine 2**
- \( C[x1] += D1 \)
- \( C[x1] += D2 \)
- ...

Logic to combine counter updates

Counter logic

Partition/Sort

Spill 1

Spill 2

Spill 3

...

Spill n

Merge Spill Files

Combine counter updates
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 \( id, y, x_1, \ldots, x_d \) in *train*:
  - \( C(\"Y=y\") ++ \)
  - For \( j \) in 1..\( d \):
    - \( C(\"Y=y \land X=x_j\") ++ \)
    - ....
- For each example \( id, y, x_1, \ldots, x_d \) in *test*:
  - For each \( y' \) in \( \text{dom}(Y) \):
    - Compute \( \log \Pr(y',x_1,\ldots,x_d) = \)
      \[
      \sum_j \log \frac{C(X = x_j \land Y = y') + mq_x}{C(X = \text{ANY} \land Y = y')} + \log \frac{C(Y = y') + mq_y}{C(Y = \text{ANY}) + m}
      \]
  - Return the best \( y' \)

Assume hashtable holding all counts fits in memory

Sequential reads

Complexity: \( O(n) \), \( n = \text{size of train} \)

where:
- \( q_j = 1/|V| \)
- \( q_y = 1/|\text{dom}(Y)| \)
- \( mq_x = 1 \)

Sequential reads

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

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

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

- Initialize a HashSet \( \text{NEEDED} \) and a hashtable \( C \)
- For each example \( id, y, x_1, \ldots, x_d \) in \( test \):
  - Add \( x_1, \ldots, x_d \) to \( \text{NEEDED} \)
- For each event, \( C(\text{event}) \) in the summed counters
  - If event involves a \( \text{NEEDED} \) term \( x \) read it into \( C \)
- For each example \( id, y, x_1, \ldots, x_d \) in \( test \):
  - For each \( y' \) in \( \text{dom}(Y) \):
    - Compute \( \log \Pr(y', x_1, \ldots, x_d) = \ldots \)

Model size: \( O(|V|) \)

Time: \( O(n^2) \), size of test
Memory: same

Time: \( O(n^2) \)
Memory: same

Time: \( O(n^2) \)
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
- Assignment: Counts as messages in a stream-and-sort system

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

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