Basics of MapReduce

Shannon Quinn
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
- Other tasks for stream-and-sort
- ...MapReduce?
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*, 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*, in *test*:
  - For each *y'* in dom(*Y*):
    - Compute \( \log \Pr(y',x_1,\ldots,x_d) = \)
      \[ \left( \sum_j \log \frac{C(X=x_j \land Y=y') + mq_x}{C(Y=y') + m} \right) + \log \frac{C(Y=y') + mq_y}{C(Y=ANY) + m} \]
    - Return the best *y'*

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

Sequential reads

Assume hashtable holding all counts fits in memory
What’s next

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

• Why?
  Micro: 0.6G memory
  Standard:
  S: 1.7Gb
  L: 7.5Gb
  XL: 15Mb
  Hi Memory: XXL: 34.2
  XXXXL: 68.4
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:

\[ V = K n^\beta \]  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, …)
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>Description</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>

- L1 cache reference is approximately 10x faster than branching.
- L2 cache reference is approximately 15x faster than branching.
- Main memory reference is 100x faster than mutex lock/unlock.
- Disk seek is 40x faster than reading 1 MB sequentially from network.
- Sending a packet across the network is approximately 100,000x slower than reading from memory.
Using a database for Big ML

- We often want to do random access on big data
  - E.g., different versions of examples for q/a
  - E.g., spot-checking parameter weights to see if they are sensible
- Simplest approach:
  - Sort the data and use binary search \( \mathcal{O}(\log_2 n) \) seeks to find query row
Using a database for Big ML

• We often want to do random access on big data
  – E.g., different versions of examples for q/a
  – E.g., spot-checking parameter weights to see if they are sensible

• Almost-as-simple idea based on fact that disk seek time $\approx$ reading 1Mb
  – Let $K = \text{rows}/\text{Mb}$ (e.g., $K=1000$)
  – Scan through data once and record the seek position of every $K$-th row in an index file (or memory)
  – To find row $r$:
    • Find the $r'$, last item in the index smaller than $r$
    • Seek to $r'$ and read the next megabyte

Cost is $\approx 2$ seeks
Using a database for Big ML

• Summary: we’ve gone from \( \sim 1 \) seek (best possible) to \( \sim 2 \) seeks---plus finding \( r' \) in the index.
  – If index is \( O(1\text{Mb}) \) then finding \( r' \) is also like 1 seek
  – So we’re paying about 3 seeks per random access in a Gb
• What if the index is still large?
  – Build (the same sort of index) for the index!
  – Now we’re paying 4 seeks for each random access into a Tb
  – ….and repeat recursively if you need
• This is called a B-tree
  – It only gets complicated when we want to delete and insert.
Combining the same track on all disks makes a cylinder.
## 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>
</tbody>
</table>

Best case (data is in same sector/block)
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...

Question: What could you do to reduce this cost?
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 …..

\[ O(n*\text{scan}) \rightarrow O(n*\text{scan}^*\text{???}) \]
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
- ....

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

Machine 0

“increment C[x] by D”

Hash table 1

Machine 1

Hash table 2

Machine 2

...,

Hash table 2

Machine K

Now we have enough memory....
Distributed Counting

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

Counting logic

Hash table 1
Machine 1
Hash table 2
Machine 2
Hash table 2
Machine K

“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 (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: $\approx 10 \times$
  - Mutex lock/unlock: $\approx 15 \times$
  - Disk seek: $40 \times$
  - Read 1 MB sequentially from disk: $\approx 100,000 \times$
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

- 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….
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
Large-vocabulary Naïve Bayes

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

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

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

\[
\begin{align*}
\text{previousKey} & = \text{Null} \\
\text{sumForPreviousKey} & = 0 \\
\text{For each (event, delta) in input:} \\
\quad & \text{If event == previousKey} \\
\quad & \quad \text{sumForPreviousKey} += \text{delta} \\
\quad & \text{Else} \\
\quad & \quad \text{OutputPreviousKey()} \\
\quad & \quad \text{previousKey} = \text{event} \\
\quad & \quad \text{sumForPreviousKey} = \text{delta} \\
\quad & \text{OutputPreviousKey()} \\
\text{define OutputPreviousKey():} \\
\quad & \text{If PreviousKey != Null} \\
\quad & \quad \text{print PreviousKey, sumForPreviousKey}
\end{align*}
\]
Distributed Counting $\rightarrow$ Stream and Sort Counting

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

Counting logic

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

Machine 0

Message-routing logic

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

Logic to combine counter updates

Machine C

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

Machine B

Counting logic

"C[x] += D"
Stream and Sort Counting $\rightarrow$ Distributed Counting

- Example 1
- Example 2
- Example 3
- ....

Counting logic

Machines A1, ...

Standardized message routing logic

“C[x] += D”

Sort

Logic to combine counter updates

Machines B1, ...

Machines C1, ...

Trivial to parallelize!

Easy to parallelize!
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

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

Model size: \(\max O(n), O(|V| |\text{dom}(Y)|)\)
Other stream-and-sort tasks

- “Meaningful” phrase-finding
A Language Model Approach to Keyphrase Extraction

Takashi Tomokiyo and Matthew Hurst
Applied Research Center
Intelliseek, Inc.
Pittsburgh, PA 15213
{ttomokiyo,mhurst}@intelliseek.com

ACL Workshop 2003
<table>
<thead>
<tr>
<th></th>
<th>civic hybrid</th>
<th>21</th>
<th>mustang gt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>honda civic hybrid</td>
<td>22</td>
<td>ford escape</td>
</tr>
<tr>
<td>3</td>
<td>toyota prius</td>
<td>23</td>
<td>steering wheel</td>
</tr>
<tr>
<td>4</td>
<td>electric motor</td>
<td>24</td>
<td>toyota prius today</td>
</tr>
<tr>
<td>5</td>
<td>honda civic</td>
<td>25</td>
<td>electric motors</td>
</tr>
<tr>
<td>6</td>
<td>fuel cell</td>
<td>26</td>
<td>gasoline engine</td>
</tr>
<tr>
<td>7</td>
<td>hybrid cars</td>
<td>27</td>
<td>internal combustion engine</td>
</tr>
<tr>
<td>8</td>
<td>honda insight</td>
<td>28</td>
<td>gas engine</td>
</tr>
<tr>
<td>9</td>
<td>battery pack</td>
<td>29</td>
<td>front wheels</td>
</tr>
<tr>
<td>10</td>
<td>sports car</td>
<td>30</td>
<td>key sense wire</td>
</tr>
<tr>
<td>11</td>
<td>civic si</td>
<td>31</td>
<td>civic type r</td>
</tr>
<tr>
<td>12</td>
<td>hybrid car</td>
<td>32</td>
<td>test drive</td>
</tr>
<tr>
<td>13</td>
<td>civic lx</td>
<td>33</td>
<td>street race</td>
</tr>
<tr>
<td>14</td>
<td>focus fcv</td>
<td>34</td>
<td>united states</td>
</tr>
<tr>
<td>15</td>
<td>fuel cells</td>
<td>35</td>
<td>hybrid powertrain</td>
</tr>
<tr>
<td>16</td>
<td>hybrid vehicles</td>
<td>36</td>
<td>rear bumper</td>
</tr>
<tr>
<td>17</td>
<td>tour de sol</td>
<td>37</td>
<td>ford focus</td>
</tr>
<tr>
<td>18</td>
<td>years ago</td>
<td>38</td>
<td>detroit auto show</td>
</tr>
<tr>
<td>19</td>
<td>daily driver</td>
<td>39</td>
<td>parking lot</td>
</tr>
<tr>
<td>20</td>
<td>jetta tdi</td>
<td>40</td>
<td>rear wheels</td>
</tr>
</tbody>
</table>

**Figure 1**: Top 40 keyphrases automatically extracted from messages relevant to “civic hybrid” using our system
Why phrase-finding?

• There are lots of phrases
• There’s not supervised data
• It’s hard to articulate
  – What makes a phrase a phrase, vs just an n-gram?
    • a phrase is independently meaningful (“test drive”, “red meat”) or not (“are interesting”, “are lots”)
  – What makes a phrase interesting?
The breakdown: what makes a good phrase

• Two properties:
  – Phraseness: “the degree to which a given word sequence is considered to be a phrase”
  – Statistics: how often words co-occur together vs separately
  – Informativeness: “how well a phrase captures or illustrates the key ideas in a set of documents” – something novel and important relative to a domain

• Background corpus and foreground corpus; how often phrases occur in each
“Phraseness” \textsubscript{1} – based on BLRT

- Binomial Ratio Likelihood Test (BLRT):
  - Draw samples:
    - \(n_1\) draws, \(k_1\) successes
    - \(n_2\) draws, \(k_2\) successes
    - Are they from one binomial (i.e., \(k_1/n_1\) and \(k_2/n_2\) were different due to chance) or from two distinct binomials?
  - Define
    - \(p_1 = k_1/n_1, \ p_2 = k_2/n_2, \ p = (k_1+k_2)/(n_1+n_2)\),
    - \(L(p,k,n) = p^k(1-p)^{n-k}\)

\[
BLRT(n_1,k_1,n_2,k_2) = \frac{L(p_1,k_1,n_1)L(p_2,k_2,n_2)}{L(p,k_1,n_1)L(p,k_2,n_2)}
\]
Phraseness” – based on BLRT

- Binomial Ratio Likelihood Test (BLRT):
  - Draw samples:
    - \(n_1\) draws, \(k_1\) successes
    - \(n_2\) draws, \(k_2\) successes
    - Are they from one binomial (i.e., \(k_1/n_1\) and \(k_2/n_2\) were different due to chance) or from two distinct binomials?
  - Define
    - \(p_i = k_i/n_i\), \(p = (k_1 + k_2)/(n_1 + n_2)\),
    - \(L(p,k,n) = p^k(1-p)^{n-k}\)

\[
BLRT(n_1,k_1,n_2,k_2) = 2 \log \frac{L(p_1,k_1,n_1)L(p_2,k_2,n_2)}{L(p,k_1,n_1)L(p,k_2,n_2)}
\]
“Phraseness” - based on BLRT

Define

\[ p_i = \frac{k_i}{n_i}, \quad p = \frac{(k_1 + k_2)}{(n_1 + n_2)}, \]

\[ L(p, k, n) = p^k(1-p)^{n-k} \]

\[ \varphi_p(n_1, k_1, n_2, k_2) = 2 \log \frac{L(p_1, k_1, n_1)L(p_2, k_2, n_2)}{L(p, k_1, n_1)L(p, k_2, n_2)} \]

<table>
<thead>
<tr>
<th>Comment</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k_1 )</td>
<td>( C(W_1 = x \land W_2 = y) )</td>
</tr>
<tr>
<td>( n_1 )</td>
<td>( C(W_1 = x) )</td>
</tr>
<tr>
<td>( k_2 )</td>
<td>( C(W_1 \neq x \land W_2 = y) )</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>( C(W_1 \neq x) )</td>
</tr>
</tbody>
</table>

Does \( y \) occur at the same frequency after \( x \) as in other positions?
“Informativeness” – based on BLRT

- Define
  - \( p_i = \frac{k_i}{n_i} \), \( p = \frac{(k_1 + k_2)}{(n_1 + n_2)} \),
  - \( L(p, k, n) = p^k (1-p)^{n-k} \)

\[
\phi_i(n_1, k_1, n_2, k_2) = 2 \log \frac{L(p_1, k_1, n_1)L(p_2, k_2, n_2)}{L(p, k_1, n_1)L(p, k_2, n_2)}
\]

<table>
<thead>
<tr>
<th></th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k_1 )</td>
<td>( C(W_1 = x \land W_2 = y) ) how often bigram ( x \ y ) occurs in corpus ( C )</td>
</tr>
<tr>
<td>( n_1 )</td>
<td>( C(W_1 = * \land W_2 = *) ) how many bigrams in corpus ( C )</td>
</tr>
<tr>
<td>( k_2 )</td>
<td>( B(W_1 = x \land W_2 = y) ) how often ( x \ y ) occurs in background corpus</td>
</tr>
<tr>
<td>( n_2 )</td>
<td>( B(W_1 = * \land W_2 = *) ) how many bigrams in background corpus</td>
</tr>
</tbody>
</table>

Phrase \( x \ y \): \( W_1 = x \land W_2 = y \) and two corpora, \( C \) and \( B \)

Does \( x \ y \) occur at the same frequency in both corpora?
<table>
<thead>
<tr>
<th>1</th>
<th>message news</th>
<th>16</th>
<th>sixth sense</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>minority report</td>
<td>17</td>
<td>hey kids</td>
</tr>
<tr>
<td>3</td>
<td>star wars</td>
<td>18</td>
<td>gaza man</td>
</tr>
<tr>
<td>4</td>
<td>john harkness</td>
<td>19</td>
<td>lee harrison</td>
</tr>
<tr>
<td>5</td>
<td>derek janssen</td>
<td>20</td>
<td>years ago</td>
</tr>
<tr>
<td>6</td>
<td>robert frenchu</td>
<td>21</td>
<td>julia roberts</td>
</tr>
<tr>
<td>7</td>
<td>sean o’hara</td>
<td>22</td>
<td>national guard</td>
</tr>
<tr>
<td>8</td>
<td>box office</td>
<td>23</td>
<td>bourne identity</td>
</tr>
<tr>
<td>9</td>
<td>dawn taylor</td>
<td>24</td>
<td>metrotoday <a href="http://www.zap2it.com">www.zap2it.com</a></td>
</tr>
<tr>
<td>10</td>
<td>anthony gaza</td>
<td>25</td>
<td>starweek magazine</td>
</tr>
<tr>
<td>11</td>
<td>star trek</td>
<td>26</td>
<td>eric chomko</td>
</tr>
<tr>
<td>12</td>
<td>ancient race</td>
<td>27</td>
<td>wilner starweek</td>
</tr>
<tr>
<td>13</td>
<td>scooby doo</td>
<td>28</td>
<td>tim gueguen</td>
</tr>
<tr>
<td>14</td>
<td>austin powers</td>
<td>29</td>
<td>jodie foster</td>
</tr>
<tr>
<td>15</td>
<td>home.attbi.com hey</td>
<td>30</td>
<td>johnnie kendricks</td>
</tr>
</tbody>
</table>
The breakdown: what makes a good phrase

• Two properties:
  – Phraseness: “the degree to which a given word sequence is considered to be a phrase”
    • Statistics: how often words co-occur together vs separately
  – Informativeness: “how well a phrase captures or illustrates the key ideas in a set of documents” – something novel and important relative to a domain
    • Background corpus and foreground corpus; how often phrases occur in each
  – Another intuition: our goal is to compare distributions and see how different they are:
    • Phraseness: estimate $x \ y$ with bigram model or unigram model
    • Informativeness: estimate with foreground vs background corpus
The breakdown: what makes a good phrase

– Another intuition: our goal is to compare distributions and see how different they are:
  • Phraseness: estimate $x y$ with bigram model or unigram model
  • Informativeness: estimate with foreground vs background corpus
– To compare distributions, use KL-divergence

$$D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

“Pointwise KL divergence”

$$\delta_w(p \parallel q) \overset{\text{def}}{=} p(w) \log \frac{p(w)}{q(w)}$$
The breakdown: what makes a good phrase

– To compare distributions, use KL-divergence

$$D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

“Pointwise KL divergence”

$$\delta_w(p \parallel q) \overset{\text{def}}{=} p(w) \log \frac{p(w)}{q(w)}$$

Phraseness: difference between bigram and unigram language model in foreground

$$\delta_w(LM^N_{fg} \parallel LM^1_{fg})$$

Bigram model: $P(x \ y) = P(x)P(y \mid x)$

Unigram model: $P(x \ y) = P(x)P(y)$
The breakdown: what makes a good phrase

– To compare distributions, use KL-divergence

\[ D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)} \]

“Pointwise KL divergence”

\[ \delta_w(p \parallel q) \overset{\text{def}}{=} p(w) \log \frac{p(w)}{q(w)} \]

Informativeness: difference between foreground and background models

\[ \delta_w(LM_{fg}^N \parallel LM_{bg}^N), \text{ or } \delta_w(LM_{fg}^1 \parallel LM_{bg}^1) \]

Bigram model: \( P(x y) = P(x)P(y \mid x) \)

Unigram model: \( P(x y) = P(x)P(y) \)
The breakdown: what makes a good phrase

– To compare distributions, use KL-divergence

\[ D(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)} \]

“Pointwise KL divergence”

\[ \delta_w(p \parallel q) \overset{\text{def}}{=} p(w) \log \frac{p(w)}{q(w)} \]

Bigram model: \( P(x \; y) = P(x)P(y \mid x) \)

Unigram model: \( P(x \; y) = P(x)P(y) \)

Combined: difference between foreground bigram model and background unigram model

\[ \delta_w(LM^N_{\text{fg}} \parallel LM^1_{\text{bg}}) \]
The breakdown: what makes a good phrase

– To compare distributions, use KL-divergence

Subtle advantages:

• BLRT scores “more frequent in foreground” and “more frequent in background” symmetrically, pointwise KL does not.

• Phrasiness and informativeness scores are more comparable – straightforward combination w/o a classifier is reasonable.

• Language modeling is well-studied:
  • extensions to n-grams, smoothing methods, …
  • we can build on this work in a modular way

Combined: difference between foreground bigram model and background unigram model

\[ \delta_w(LM^N_{fg} \parallel LM^1_{bg}) \]
<table>
<thead>
<tr>
<th></th>
<th>message news</th>
<th></th>
<th>hey kids</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>minority report</td>
<td></td>
<td>years ago</td>
</tr>
<tr>
<td>3</td>
<td>star wars</td>
<td></td>
<td>gaza man</td>
</tr>
<tr>
<td>4</td>
<td>john harkness</td>
<td></td>
<td>sixth sense</td>
</tr>
<tr>
<td>5</td>
<td>robert frenchu</td>
<td></td>
<td>lee harrison</td>
</tr>
<tr>
<td>6</td>
<td>derek janssen</td>
<td></td>
<td>julia roberts</td>
</tr>
<tr>
<td>7</td>
<td>box office</td>
<td></td>
<td>national guard</td>
</tr>
<tr>
<td>8</td>
<td>sean o’hara</td>
<td></td>
<td>bourne identity</td>
</tr>
<tr>
<td>9</td>
<td>dawn taylor</td>
<td></td>
<td>metrotoday <a href="http://www.zap2it.com">www.zap2it.com</a></td>
</tr>
<tr>
<td>10</td>
<td>anthony gaza</td>
<td></td>
<td>starweek magazine</td>
</tr>
<tr>
<td>11</td>
<td>star trek</td>
<td></td>
<td>eric chomko</td>
</tr>
<tr>
<td>12</td>
<td>ancient race</td>
<td></td>
<td>wilner starweek</td>
</tr>
<tr>
<td>13</td>
<td>home.attbibi.com hey</td>
<td></td>
<td>tim gueguen</td>
</tr>
<tr>
<td>14</td>
<td>scooby doo</td>
<td></td>
<td>jodie foster</td>
</tr>
<tr>
<td>15</td>
<td>austin powers</td>
<td></td>
<td>kevin filmnutboy</td>
</tr>
</tbody>
</table>
Why phrase-finding?

- Phrases are where the standard supervised “bag of words” representation starts to break.
- There’s not supervised data, so it’s hard to see what’s “right” and why
- It’s a nice example of using unsupervised signals to solve a task that could be formulated as supervised learning
- It’s a nice level of complexity, if you want to do it in a scalable way.
Implementation

• Request-and-answer pattern
  – Main data structure: tables of key-value pairs
    • $key$ is a phrase $x\ y$
    • $value$ is a mapping from attribute names (like $\text{phraseness}$, $\text{freq-in-B}$, ...
      ) to numeric values.
  – Keys and values are just strings
  – We’ll operate mostly by sending messages to this data structure and getting results back, or else streaming thru the whole table
  – For really big data: we’d also need tables where $key$ is a word and $val$ is set of attributes of the word ($\text{freq-in-B}$, $\text{freq-in-C}$, ...

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>old man</td>
<td>freq(B)=10,freq(C)=13,informativeness=1.3,phrasiness=740</td>
</tr>
<tr>
<td>bad service</td>
<td>freq(B)=8,freq(C)=25,informativeness=560,phrasiness=254</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Generating and scoring phrases: 1

• Stream through foreground corpus and count events “\(W_1=x \land W_2=y\)” the same way we do in training naive Bayes: stream-and-sort and accumulate deltas (a “sum-reduce”)
  – Don’t bother generating boring phrases (e.g., crossing a sentence, contain a stopword, …)
• Then stream through the output and convert to phrase, attributes-of-phrase records with one attribute: \(freq\)-in-C\(=n\)
• Stream through foreground corpus and count events “\(W_1=x\)” in a (memory-based) hashtable....
• This is enough* to compute phrasiness:
  – \(\psi_p(x \ y) = f(freq\)-in-C\(x\), \(freq\)-in-C\(y\), \(freq\)-in-C\(x \ y\))

• …so you can do that with a scan through the phrase table that adds an extra attribute (holding word frequencies in memory).

* actually you also need total # words and total # phrases....
Generating and scoring phrases: 2

• Stream through **background** corpus and count events “$W_1=x \land W_2=y$” and convert to phrase, attributes-of-phrase records with one attribute: $freq-in-B=n$

• Sort the two phrase-tables: $freq-in-B$ and $freq-in-C$ and run the output through another “reducer” that
  – **appends** together all the attributes associated with the same key, so we now have elements like

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>old man</td>
<td>$freq(B)=10,freq(C)=13,phrasiness=740$</td>
</tr>
<tr>
<td>bad service</td>
<td>$freq(B)=8,freq(C)=25,phrasiness=254$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Generating and scoring phrases: 3

- Scan the through the phrase table one more time and add the informativeness attribute and the overall quality attribute

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>old man</td>
<td>freq(B)=10, freq(C)=13, informativeness=1.3, phrasiness=740</td>
</tr>
<tr>
<td>bad service</td>
<td>freq(B)=8, freq(C)=25, informativeness=560, phrasiness=254</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Summary, assuming word vocabulary $n_W$ is small:
- Scan foreground corpus $C$ for phrases: $O(n_C)$ producing $m_C$ phrase records – of course $m_C << n_C$
- Compute phrasiness: $O(m_C)$  
  Assumes word counts fit in memory
- Scan background corpus $B$ for phrases: $O(n_B)$ producing $m_B$
- Sort together and combine records: $O(m \log m)$, $m=m_B + m_C$
- Compute informativeness and combined quality: $O(m)$
Ramping it up – keeping word counts out of memory


• Assume I have built built phrase tables and word tables….how do I incorporate the word attributes into the phrase records?

• For each phrase $xy$, request necessary word frequencies:
  – Print “$x$ ~request=$freq$-in-$C$,from=$xy$”
  – Print “$y$ ~request=$freq$-in-$C$,from=$xy$”

• Sort all the word requests in with the word tables

• Scan through the result and generate the answers: for each word $w$, $a_1=n_1,a_2=n_2,…$
  – Print “$xy$ ~request=$freq$-in-$C$,from=$w$”

• Sort the answers in with the $xy$ records

• Scan through and augment the $xy$ records appropriately
Generating and scoring phrases: 3

Summary
1. Scan foreground corpus C for phrases, words: O(n_C) producing m_C phrase records, v_C word records
2. Scan phrase records producing word-freq requests: O(m_C) producing 2m_C requests
3. Sort requests with word records: O((2m_C + v_C)log(2m_C + v_C)) = O(m_Clog m_C) since v_C < m_C
4. Scan through and answer requests: O(m_C)
5. Sort answers with phrase records: O(m_Clog m_C)
6. Repeat 1-5 for background corpus: O(n_B + m_Blogm_B)
7. Combine the two phrase tables: O(m log m), m = m_B + m_C
8. Compute all the statistics: O(m)
Outline

• Even more on stream-and-sort and naïve Bayes
  – Request-answer pattern
• Another problem: “meaningful” phrase finding
  – Statistics for identifying phrases (or more generally correlations and differences)
  – Also using foreground and background corpora
• Implementing “phrase finding” efficiently
  – Using request-answer
• Some other phrase-related problems
  – Semantic orientation
  – Complex named entity recognition
Basically...

- Stream-and-sort == ?
MapReduce!

- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, Map and Reduce change to fit the problem

MapReduce: The **Map** Step

**Input key-value pairs**

```
\[ (k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n) \]
```

**Intermediate key-value pairs**

```
\[ ([k_1, v_1], \ldots, [k_n, v_n]) \]
```

MapReduce: The Reduce Step

Intermediate key-value pairs

Key-value groups

Output key-value pairs

Group by key

reduce

reduce

reduce

More Specifically

- **Input**: a set of key-value pairs
- Programmer specifies two methods:
  - **Map**(k, v) → <k’, v’>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every (k,v) pair
  - **Reduce**(k’, <v’>*) → <k’, v’”>*
    - All values v’ with same key k’ are reduced together and processed in v’ order
    - There is one Reduce function call per unique key k’
Large-scale Computing

• Large-scale computing for data mining problems on commodity hardware

• Challenges:
  – How do you distribute computation?
  – How can we make it easy to write distributed programs?
  – Machines fail:
    • One server may stay up 3 years (1,000 days)
    • If you have 1,000 servers, expect to lose 1/day
    • People estimated Google had ~1M machines in 2011
      – 1,000 machines fail every day!

Idea and Solution

• **Issue:** Copying data over a network takes time

• **Idea:**
  – Bring computation close to the data
  – Store files multiple times for reliability

• **Map-reduce addresses these problems**
  – Google’s computational/data manipulation model
  – Elegant way to work with big data
  – **Storage Infrastructure – File system**
    • Google: GFS. Hadoop: HDFS

  – **Programming model**
    • Map-Reduce
Storage Infrastructure

• **Problem:**
  – If nodes fail, how to store data persistently?

• **Answer:**
  – **Distributed File System:**
    • Provides global file namespace
    • Google GFS; Hadoop HDFS;

• **Typical usage pattern**
  – Huge files (100s of GB to TB)
  – Data is rarely updated in place
  – Reads and appends are common
Distributed File System

• **Chunk servers**
  – File is split into contiguous chunks
  – Typically each chunk is 16-64MB
  – Each chunk replicated (usually 2x or 3x)
  – Try to keep replicas in different racks

• **Master node**
  – a.k.a. Name Node in Hadoop’s HDFS
  – Stores metadata about where files are stored
  – Might be replicated

• **Client library for file access**
  – Talks to master to find chunk servers
  – Connects directly to chunk servers to access data
Distributed File System

- Reliable distributed file system
- Data kept in “chunks” spread across machines
- Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure

Bring computation directly to the data!

Chunk servers also serve as compute servers
Programming Model: MapReduce

Warm-up task:

• We have a huge text document

• Count the number of times each distinct word appears in the file

• Sample application:
  – Analyze web server logs to find popular URLs
Task: Word Count

Case 1:
– File too large for memory, but all \(<\text{word}, \text{count}>\) pairs fit in memory

Case 2:
• Count occurrences of words:
  \[\text{words(doc.txt)} \mid \text{sort} \mid \text{uniq -c}\]
    • where \text{words} takes a file and outputs the words in it, one per a line

• Case 2 captures the essence of MapReduce
  – Great thing is that it is naturally parallelizable
Data Flow

• **Input and final output** are stored on a distributed file system (FS):
  – Scheduler tries to schedule map tasks “close” to physical storage location of input data

• **Intermediate results** are stored on local FS of Map and Reduce workers

• **Output** is often input to another MapReduce task
Coordination: Master

• **Master node takes care of coordination:**
  – **Task status**: (idle, in-progress, completed)
  – **Idle tasks** get scheduled as workers become available
  – When a map task completes, it sends the master the location and sizes of its $R$ intermediate files, one for each reducer
  – Master pushes this info to reducers

• **Master pings workers periodically to detect failures**
Dealing with Failures

• **Map worker failure**
  – Map tasks completed or in-progress at worker are reset to idle
  – Reduce workers are notified when task is rescheduled on another worker

• **Reduce worker failure**
  – Only in-progress tasks are reset to idle
  – Reduce task is restarted

• **Master failure**
  – MapReduce task is aborted and client is notified
How many Map and Reduce jobs?

• $M$ map tasks, $R$ reduce tasks
• **Rule of a thumb:**
  - Make $M$ much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
• Usually $R$ is smaller than $M$
  - Because output is spread across $R$ files
Task Granularity & Pipelining

- **Fine granularity tasks:** map tasks $\rightarrow$ machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map

---

Refinement: Combiners

• Often a Map task will produce many pairs of the form $(k, v_1), (k, v_2), \ldots$ for the same key $k$
  – E.g., popular words in the word count example

• Can save network time by pre-aggregating values in the mapper:
  – $\text{combine}(k, \text{list}(v_1)) \rightarrow v_2$
  – Combiner is usually same as the reduce function

• Works only if reduce function is commutative and associative
Refinement: Combiners

- Back to our word counting example:
  - Combiner combines the values of all keys of a single mapper (single machine):

  - Much less data needs to be copied and shuffled!
Refinement: Partition Function

• **Want to control how keys get partitioned**
  – Inputs to map tasks are created by contiguous splits of input file
  – Reduce needs to ensure that records with the same intermediate key end up at the same worker

• **System uses a default partition function:**
  – $\text{hash(key)} \mod R$

• **Sometimes useful to override the hash function:**
  – E.g., $\text{hash(hostname(URL)) \mod R}$ ensures URLs from a host end up in the same output file
Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using
  1. *Communication cost* = total I/O of all processes
  2. *Elapsed communication cost* = max of I/O along any path
  3. *(Elapsed) computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)
Example: Cost Measures

- For a map-reduce algorithm:
  - Communication cost = input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - Elapsed communication cost is the sum of the largest input + output for any map process, plus the same for any reduce process.
What Cost Measures Mean

• Either the I/O (communication) or processing (computation) cost dominates
  – Ignore one or the other

• Total cost tells what you pay in rent from your friendly neighborhood cloud

• Elapsed cost is wall-clock time using parallelism
Cost of Map-Reduce Join

- **Total communication cost**
  \[ O(|R| + |S| + |R \bowtie S|) \]

- **Elapsed communication cost**
  \[ O(s) \]
  - We’re going to pick \( k \) and the number of Map processes so that the I/O limit \( s \) is respected.
  - We put a limit \( s \) on the amount of input or output that any one process can have. \( s \) could be:
    - What fits in main memory
    - What fits on local disk

- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like comm. cost
Performance

• IMPORTANT
  – You may not have room for all reduce values in memory
    • In fact you should PLAN not to have memory for all values
    • Remember, small machines are much cheaper
      – you have a limited budget
Implementations

• Google
  – Not available outside Google

• Hadoop
  – An open-source implementation in Java
  – Uses HDFS for stable storage

• Spark
  – An open-source implementation in Scala
  – Uses several distributed filesystems
  – Download: http://spark.apache.org/

• Others

Reading

• Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters

• Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
Further Reading

• Programming model inspired by functional language primitives
• Partitioning/shuffling similar to many large-scale sorting systems
  – NOW-Sort ['97]
• Re-execution for fault tolerance
  – BAD-FS ['04] and TACC ['97]
• Locality optimization has parallels with Active Disks/Diamond work
  – Active Disks ['01], Diamond ['04]
• Backup tasks similar to Eager Scheduling in Charlotte system
  – Charlotte ['96]
• Dynamic load balancing solves similar problem as River's distributed queues
  – River ['99]