Word Context Entropy

Kenneth Heafield

Google Inc

January 16, 2008

1. Problem
   - Context
   - Entropy

2. Implementation
   - Streaming Entropy
   - Reducer Sorting
   - Custom Partitioner

Word Weighting

Idea

Measure how specific a word is

Applications

- Query refinement
  Results **1** - **10** of about **2,910,000** for **a of the attleboro**.
- Automatic tagging

Example

Specific	pangolin	whistle	bug	airplane	purple
	1.6	4.2	4.9	5.0	5.3
Generic	sufficiently	any	from	is	a
	6.4	8.7	9.6	9.6	9.8

Neighbors

Idea

Non-specific words appear in random contexts.

Example

- A bug in the code is worth two in the documentation.
- A complex system that works is invariably found to have evolved from a simple system that works.
- A computer scientist is someone who fixes things that aren't broken.
- I'm still waiting for the advent of the computer science groupie.
- If I'd known computer science was going to be like this, I'd never have given up being a rock 'n' roll star.

A bug, complex, from, simple, computer, being, rock

Quotes copied from fortune computers file

Neighbors

Idea

Non-specific words appear in random contexts.

Example

- A bug in the code is worth two in the documentation.
- A complex system that works is invariably found to have evolved from a simple system that works.
- A computer scientist is someone who fixes things that aren't broken.
- I'm still waiting for the advent of the computer science groupie.
- If I'd known computer science was going to be like this, I'd never have given up being a rock 'n' roll star.

A	bug, complex, from, simple, computer, being, rock
Computer	A, scientist, the, science, known, science

Quotes copied from fortune computers file

Neighbors

Idea

Non-specific words appear in random contexts.

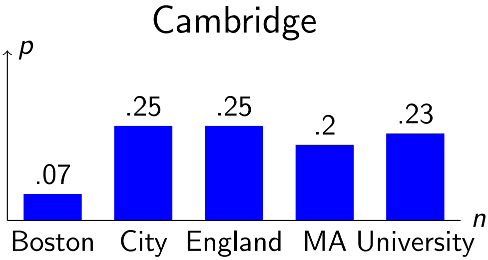
Example

- A bug in the code is worth two in the documentation.
- A complex system that works is invariably found to have evolved from a simple system that works.
- A computer scientist is someone who fixes things that aren't broken.
- I'm still waiting for the advent of the computer science groupie.
- If I'd known computer science was going to be like this, I'd never have given up being a rock 'n' roll star.

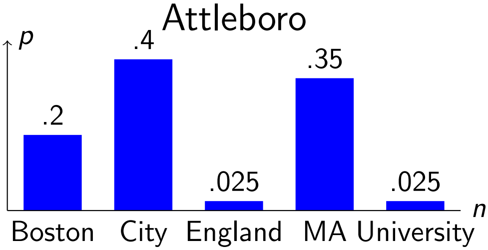
A	bug, complex, from, simple, computer, being, rock
Computer	A, scientist, the, science, known, science

Quotes copied from fortune computers file

Context Distribution



- Ambiguous
- Closer to uniform



- Just a city in MA
- Spiked

Numbers on this slide are illustrative only.

Entropy

Definition

Measures how uncertain a random variable N is:

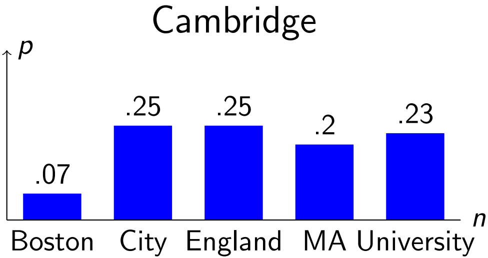
$$Entropy\left(N\right) = -\sum_{n} p\left(N = n\right)\log_2 p\left(N = n\right)$$

Properties

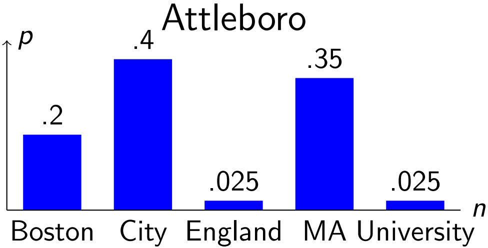
Minimized at 0 when only one outcome is possible

Maximized at $\log_2 k$ when k outcomes are equally probable

Context Distribution



n	p	$-p\log_2 p$
Boston	0.07	0.269
City	0.25	0.5
England	0.25	0.5
MA	0.2	0.464
University	0.23	0.487
Entropy		2.221



n	p	$-p\log_2 p$
Boston	0.2	0.464
City	0.4	0.529
England	0.025	0.133
MA	0.35	0.53
University	0.025	0.133
Entropy		1.789

Numbers on this slide are illustrative only.

Summary

Goal

Measure how specific a word is

Approach

1. Count the surrounding words
2. Normalize to make a probability distribution
3. Evaluate entropy

Frequentist statistics are used for simplicity.

All At Once

Implementation

Mapper outputs key *word* and value *neighbor*.

Reducer
1. Counts each *neighbor* using a hash table.
2. Normalizes *counts*.
3. Computes entropy and outputs key *word*, value *entropy*.

Example Reduce

Neighbors	City, Boston, City, MA, England, City, England						
Hash Table	City→3		Boston→1		MA→1		England→2
Normalized	$\frac{3}{7}$		$\frac{1}{7}$		$\frac{1}{7}$		$\frac{2}{7}$
Entropy	.524	+	.401	+	.401	+	.517

All At Once

Implementation

Mapper outputs key *word* and value *neighbor*.

Reducer
1. Counts each *neighbor* using a hash table.
2. Normalizes *counts*.
3. Computes entropy and outputs key *word*, value *entropy*.

Example Reduce

Neighbors	City, Boston, City, MA, England, City, England						
Hash Table	City$\rightarrow$3		Boston$\rightarrow$1		MA$\rightarrow$1		England$\rightarrow$2
Normalized	$\frac{3}{7}$		$\frac{1}{7}$		$\frac{1}{7}$		$\frac{2}{7}$
Entropy	.524	$+$	.401	$+$	.401	$+$	.517

Issues

- Too many neighbors of "the" to fit in memory.

Two Phases

Implementation

1. Count

   Mapper outputs key ($word$, $neighbor$) and empty value.

   Reducer counts values.

   Then it outputs key *word* and value *count*.

Two Phases

Implementation

1. Count

   Mapper outputs key ($word, neighbor$) and empty value.
   Reducer counts values.
   Then it outputs key *word* and value *count*.

2. Entropy

   Mapper is Identity. All counts for *word* go to one Reducer.
   Reducer buffers counts, normalizes, and computes entropy.

Issues

+ Entropy Reducer needs only counts in memory.
- There can still be a lot of counts.

An Observation about Entropy

Claim

$$Entropy = \log_2 total - \frac{partial}{total}$$

n	A neighbor of the word
$count(n)$	How many times neighbor n appeared near the word
$total$	The total number of neighbors word has: $\sum_n count(n)$
$partial$	Partial entropy: $\sum_n count(n) \log_2 count(n)$

Moral

Reducers can compute *Entropy* by accumulating two sums, *total* and *partial*, using constant memory.

Proof of Streaming Entropy

Proof

$$Entropy = -\sum_n p(N=n)\log_2 p(N=n) \tag{1}$$

$$= -\sum_n \frac{count(n)}{total}\left(\log_2 count(n) - \log_2 total\right) \tag{2}$$

$$= \log_2 total - \sum_n \left(\frac{count(n)}{total}\log_2 count(n)\right) \tag{3}$$

$$= \log_2 total - \frac{1}{total}\sum_n \left(count(n)\log_2 count(n)\right) \tag{4}$$

$$Entropy = \log_2 total - \frac{partial}{total} \tag{5}$$

Two Phases with Streaming Entropy

Implementation

1. Count

   Mapper outputs key (*word*, *neighbor*) and empty value.
   Reducer counts values.
   Then it outputs key *word* and value *count*.

2. Entropy

   Mapper is Identity. All counts for *word* go to one Reducer.
   Reducer computes streaming entropy.

Issues

+ Constant memory Reducers.

Two Phases with Streaming Entropy

Implementation

1. Count

   Mapper outputs key (*word*, *neighbor*) and empty value.
   Reducer counts values.
   Then it outputs key *word* and value *count*.

2. Entropy

   Mapper is Identity. All counts for *word* go to one Reducer.
   Reducer computes streaming entropy.

Issues

\+ Constant memory Reducers.

\- Not enough disk to store counts thrice on HDFS.

Counting Reducer

Word	Neighbor
A	Plane
Qux	Bar
A	Bird
A	Plane
A	The
Qux	Foo
A	Plane
A	The

Counting Reducer Detail

Word	Neighbor
A	Plane
Qux	Bar
A	Bird
A	Plane
A	The
Qux	Foo
A	Plane
A	The

Sort $\longrightarrow$

Word	Neighbor	Call	Output
A	Bird	→Reduce	→A:1
A	Plane		
A	Plane	→Reduce	→A:3
A	Plane		
A	The	→Reduce	→A:2
A	The		
Qux	Bar	→Reduce	→Qux:1
Qux	Foo	→Reduce	→Qux:1

Stateful Counting Reducer

Word	Neighbor
A	Plane
Qux	Bar
A	Bird
A	Plane
A	The
Qux	Foo
A	Plane
A	The

Sort →

Word	Neighbor	Call	Output
A	Bird	→Reduce	
		↓A:1	
A	Plane		
A	Plane	→Reduce	
A	Plane		
		↓A:1,3	
A	The	→Reduce	
A	The		
		↓A:1,3,2	
Qux	Bar	→Reduce	→A:1,3,2
		↓Qux:1	
Qux	Foo	→Reduce	
		↓Qux:1,1	
		→Close	→Qux:1,1

Using the Sort

Implementation

1. Count

   Mapper outputs key (*word*, *neighbor*) and empty value.
   Sorter sorts by *word* then by *neighbor*.
   Reducer counts neighbors of a word in its part.
   Outputs one key per *word* with value a list of counts.

2. Entropy

   Mapper is Identity. All counts for *word* go to one Reducer.
   Reducer computes streaming entropy.

Issues

+ Less key duplication.
- Still storing all counts.

Local Streaming Entropy

Recall

n	A neighbor of the word
$count(n)$	How many times neighbor n appeared near the word
$total$	$\sum_n count(n)$
$partial$	$\sum_n count(n) \log_2 count(n)$

Observe

Streaming entropy allows summarization of counts in each part:

$$total = total\ (\text{Part1}) + total\ (\text{Part2}) + \cdots + total\ (\text{Part}k)$$

$$partial = partial(\text{Part1}) + partial(\text{Part2}) + \cdots + partial(\text{Part}k)$$

Stateful Counting Reducer

Word	Neighbor	Call	Output
A	Bird	→Reduce	
		↓A:1	
A	Plane		
A	Plane	→Reduce	
A	Plane		
		↓A:1,3	
A	The	→Reduce	
A	The		
		↓A:1,3,2	
Qux	Bar	→Reduce	→A:1,3,2
		↓Qux:1	
Qux	Foo	→Reduce	
		↓Qux:1,1	
		→Close	→Qux:1,1

Local Streaming Entropy Reducer

Word	Neighbor	Call	Output
A	Bird	→Reduce	
		↓A: $total = 1$, $partial = 0$	
A A A	Plane Plane Plane	→Reduce	
		↓A: $total = 4$, $partial = 4.8$	
A A	The The	→Reduce	
		↓A: $total = 6$, $partial = 6.8$	
Qux	Bar	→Reduce	→A: $total = 6$, $partial = 6.8$
		↓Qux: $total = 1$, $partial = 0$	
Qux	Foo	→Reduce	
		↓Qux: $total = 2$, $partial = 0$	
		→Close	→Qux: $total = 2$, $partial = 0$

Local Streaming Entropy

Implementation

1. Local Entropy

   Mapper outputs key (*word*, *neighbor*) and empty value.
   Sorter sorts by *word* then by *neighbor*.
   Reducer computes streaming entropy within its part.
   Outputs one key per *word* with value (*total*, *partial*).

2. Entropy

   Mapper sends (*total*, *partial*) pairs for *word* to one Reducer.
   Reducer sums *total* and *partial* before computing entropy.

Issues

+ Constant memory Reducers and less intermediate data.

Local Streaming Entropy

Implementation

1. Local Entropy
   - **Mapper** outputs key (*word*, *neighbor*) and empty value.
   - **Sorter** sorts by *word* then by *neighbor*.
   - **Reducer** computes streaming entropy within its part. Outputs one key per *word* with value (*total*, *partial*).
2. Entropy
   - **Mapper** sends (*total*, *partial*) pairs for *word* to one Reducer.
   - **Reducer** sums *total* and *partial* before computing entropy.

Issues

+ Constant memory Reducers and less intermediate data.
- Local entropy is useful if neighbors are in the same part.

Balance Versus Local Entropy

Partition Function

$(word, neighbor) \to$ Part $Hash(word, neighbor)$ % 3

Example Reduce Parts

Part 0		Part 1		Part 2	
Word	Neighbor	Word	Neighbor	Word	Neighbor
A	Bird	A	Engine	A	Circus
A	Plane	A	Lift	A	Circus
A	Plane	A	What	A	Flying
A	Plane	A	What	A	Flying
A	The	Qux	Baz	A	Flying
A	The	Quz	Baz	Qux	Corge
Qux	Bar			Qux	Corge
Qux	Foo				

Balance Versus Local Entropy

Partition Function

$(word, neighbor) \rightarrow \text{Part } Hash(word, Hash(neighbor) \% 2) \% 3$

Example Reduce Parts

Part 0		Part 1		Part 2	
Word	Neighbor	Word	Neighbor	Word	Neighbor
A	Bird	Qux	Baz	A	Circus
A	Plane	Qux	Baz	A	Circus
A	Plane	Qux	Corge	A	Engine
A	Plane	Qux	Corge	A	Flying
A	The	Qux	Foo	A	Flying
A	The			A	Flying
A	Lift			A	What
A	Lift			Qux	Bar

Tuning Partitioner

Partition Function

$(word, neighbor) \rightarrow \text{Part } Hash(word, Hash(neighbor) \ \% \ Sub) \ \% \ Parts$

Parts Number of reducers

Sub Number of reducers processing *word*

Effect of *Sub*

Large *Sub* Spread neighbors evenly over Reducers

Small *Sub* Less intermediate output since local entropy is effective

Conclusion

Implementation

1. Local Entropy
   - **Mapper** outputs key (*word*, *neighbor*) and empty value.
   - **Partitioner** puts neighbors of *word* into a few parts.
   - **Sorter** sorts by *word* then by *neighbor*.
   - **Reducer** streaming entropy for *word* within its part.
2. Entropy
   - **Mapper** sends (*total*, *partial*) pairs for *word* to one Reducer.
   - **Reducer** sums *total* and *partial* before computing entropy.

Results

Specific	pangolin	whistle	bug	airplane	purple
	1.6	4.2	4.9	5.0	5.3
Generic	sufficiently	any	from	is	a
	6.4	8.7	9.6	9.6	9.8