Data-Intensive Information Processing Applications — Session #9

Language Models

Nitin Madnani
University of Maryland

Tuesday, April 6, 2010

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details
Today’s Agenda

- What are Language Models?
  - Mathematical background and motivation
  - Dealing with data sparsity (*smoothing*)
  - Evaluating language models

- Large Scale Language Models using MapReduce
N-Gram Language Models

- **What?**
  - LMs assign probabilities to sequences of tokens

- **How?**
  - Based on previous word histories
  - \( n \)-gram = consecutive sequences of tokens

- **Why?**
  - Speech recognition
  - Handwriting recognition
  - Predictive text input
  - **Statistical machine translation**
Statistical Machine Translation

Training Data

- i saw the small table
  - vi la mesa pequeña

Parallel Sentences

- he sat at the table
  - the service was good

Target-Language Text

- maria no daba una bofetada a la bruja verde

Foreign Input Sentence

- vi, i saw

Phrase Extraction

- (vi, i saw)
  - (la mesa pequeña, the small table)

Language Model

Translation Model

Decoder

English Output Sentence

- mary did not slap the green witch
Maria no dio una bofetada a la bruja verde.

Mary did not give a slap to the green witch.
N-Gram Language Models

N=1 (unigrams)

Unigrams:

This,
is,a,sentence

Sentence of length $s$, how many unigrams?
N-Gram Language Models

N=2 (bigrams)

Sentence of length \( s \), how many bigrams?
N-Gram Language Models

N=3 (trigrams)

This is a sentence

Trigrams:
This is a,
is a sentence

Sentence of length $s$, how many trigrams?
Computing Probabilities

\[ P(w_1, w_2, \ldots, w_T) \]

\[ = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2) \ldots P(w_T|w_1, \ldots, w_{T-1}) \]

[chain rule]

Is this practical?

No! Can’t keep track of all possible histories of all words!
Approximating Probabilities

**Basic idea:** limit history to fixed number of words $N$
(Markov Assumption)

$$P(w_k | w_1, \ldots, w_{k-1}) \approx P(w_k | w_{k-N+1}, \ldots, w_{k-1})$$

**N=1: Unigram Language Model**

$$P(w_k | w_1, \ldots, w_{k-1}) \approx P(w_k)$$

$$\Rightarrow P(w_1, w_2, \ldots, w_T) \approx P(w_1)P(w_2)\ldots P(w_T)$$

Relation to HMMs?
Approximating Probabilities

**Basic idea:** limit history to fixed number of words $N$ (Markov Assumption)

$$P(w_k | w_1, \ldots, w_{k-1}) \approx P(w_k | w_{k-N+1}, \ldots, w_{k-1})$$

**N=2: Bigram Language Model**

$$P(w_k | w_1, \ldots, w_{k-1}) \approx P(w_k | w_{k-1})$$

$$\Rightarrow P(w_1, w_2, \ldots, w_T) \approx P(w_1 | <S>) P(w_2 | w_1) \ldots P(w_T | w_{T-1})$$

Relation to HMMs?
Approximating Probabilities

**Basic idea:** limit history to fixed number of words N (Markov Assumption)

\[ P(w_k | w_1, \ldots, w_{k-1}) \approx P(w_k | w_{k-N+1}, \ldots, w_{k-1}) \]

**N=3: Trigram Language Model**

\[ P(w_k | w_1, \ldots, w_{k-1}) \approx P(w_k | w_{k-2}, w_{k-1}) \]

\[ \Rightarrow P(w_1, w_2, \ldots, w_T) \approx P(w_1 | <S><S>) \ldots P(w_T | w_{T-2}w_{T-1}) \]

Relation to HMMs?
Building N-Gram Language Models

- Use existing sentences to compute n-gram probability estimates (training)

- Terminology:
  - \( N \) = total number of words in training data (tokens)
  - \( V \) = vocabulary size or number of unique words (types)
  - \( C(w_1, \ldots, w_k) \) = frequency of n-gram \( w_1, \ldots, w_k \) in training data
  - \( P(w_1, \ldots, w_k) \) = probability estimate for n-gram \( w_1 \ldots w_k \)
  - \( P(w_k|w_1, \ldots, w_{k-1}) \) = conditional probability of producing \( w_k \) given the history \( w_1, \ldots w_{k-1} \)

What’s the vocabulary size?
Building N-Gram Models

- Start with what’s easiest!
- Compute maximum likelihood estimates for individual n-gram probabilities
  - Unigram: \( P(w_i) = \frac{C(w_i)}{N} \)
  - Bigram: \( P(w_i, w_j) = \frac{C(w_i, w_j)}{N} \)

\[
P(w_j|w_i) = \frac{P(w_i, w_j)}{P(w_i)} = \frac{C(w_i, w_j)}{\sum_w C(w_i, w)} = \frac{C(w_i, w_j)}{C(w_i)}
\]

- Uses relative frequencies as estimates
- Maximizes the likelihood of the data given the model \( P(D|M) \)

Why not just substitute \( P(w_i) \)?
Example: Bigram Language Model

Training Corpus

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

P( I | <s> ) = 2/3 = 0.67
P( am | I ) = 2/3 = 0.67
P( </s> | Sam )= 1/2 = 0.50

P( Sam | <s> ) = 1/3 = 0.33
P( do | I ) = 1/3 = 0.33
P( Sam | am) = 1/2 = 0.50

... 

Bigram Probability Estimates

Note: We don’t ever cross sentence boundaries
Building N-Gram Models

- Start with what’s easiest!
- Compute maximum likelihood estimates for individual n-gram probabilities
  - Unigram: \[ P(w_i) = \frac{C(w_i)}{N} \]
  - Bigram: \[ P(w_i, w_j) = \frac{C(w_i, w_j)}{N} \]
    \[ P(w_j | w_i) = \frac{P(w_i, w_j)}{P(w_i)} = \frac{C(w_i, w_j)}{\sum_w C(w_i, w)} = \frac{C(w_i, w_j)}{C(w_i)} \]
- Uses relative frequencies as estimates
- Maximizes the likelihood of the data given the model \[ P(D|M) \]

Let’s revisit this issue…

Why not just substitute \( P(w_i) \)?
More Context, More Work

- Larger $N = \text{more context}$
  - Lexical co-occurrences
  - Local syntactic relations

- More context is better?

- Larger $N = \text{more complex model}$
  - For example, assume a vocabulary of 100,000
  - How many parameters for unigram LM? Bigram? Trigram?

- Larger $N$ has another more serious problem!
Data Sparsity

\[
P(I \mid <s>) = \frac{2}{3} = 0.67 \\
P(\text{am} \mid I) = \frac{2}{3} = 0.67 \\
P(<s> \mid \text{Sam}) = \frac{1}{2} = 0.50
\]

... 

Bigram Probability Estimates

\[
P(I \text{ like ham}) \\
= P(I \mid <s>) \cdot P(\text{like} \mid I) \cdot P(\text{ham} \mid \text{like}) \cdot P(<s> \mid \text{ham}) \\
= 0
\]

Why? 
Why is this bad?
Data Sparsity

- Serious problem in language modeling!
- Becomes more severe as N increases
  - What’s the tradeoff?
- Solution 1: Use larger training corpora
  - Can’t always work... Blame Zipf’s Law (Looong tail)
- Solution 2: Assign non-zero probability to unseen n-grams
  - Known as smoothing
Smoothing

- Zeros are bad for any statistical estimator
  - Need better estimators because MLEs give us a lot of zeros
  - A distribution without zeros is “smoother”

- The Robin Hood Philosophy: Take from the rich (seen n-grams) and give to the poor (unseen n-grams)
  - And thus also called discounting
  - Critical: make sure you still have a valid probability distribution!

- Language modeling: theory vs. practice
Laplace’s Law

- Simplest and oldest smoothing technique
- Just add 1 to all n-gram counts including the unseen ones
- So, what do the revised estimates look like?
Laplace’s Law: Probabilities

**Unigrams**

\[ P_{MLE}(w_i) = \frac{C(w_i)}{N} \quad \rightarrow \quad P_{LAP}(w_i) = \frac{C(w_i) + 1}{N + V} \]

**Bigrams**

\[ P_{MLE}(w_i, w_j) = \frac{C(w_i, w_j)}{N} \quad \rightarrow \quad P_{LAP}(w_i, w_j) = \frac{C(w_i, w_j) + 1}{N + V^2} \]

Careful, don’t confuse the N’s!

\[ P_{LAP}(w_j|w_i) = \frac{P_{LAP}(w_i, w_j)}{P_{LAP}(w_i)} = \frac{C(w_i, w_j) + 1}{C(w_i) + V} \]

What if we don’t know V?
Laplace’s Law: Frequencies

Expected Frequency Estimates

\[ C_{LAP}(w_i) = P_{LAP}(w_i)N \]
\[ C_{LAP}(w_i, w_j) = P_{LAP}(w_i, w_j)N \]

Relative Discount

\[ d_1 = \frac{C_{LAP}(w_i)}{C(w_i)} \]
\[ d_2 = \frac{C_{LAP}(w_i, w_j)}{C(w_i, w_j)} \]
Laplace’s Law

- Bayesian estimator with uniform priors
- Moves too much mass over to unseen n-grams
- What if we added a fraction of 1 instead?
Lidstone’s Law of Succession

- Add $0 < \gamma < 1$ to each count instead
- The smaller $\gamma$ is, the lower the mass moved to the unseen n-grams (0=no smoothing)
- The case of $\gamma = 0.5$ is known as Jeffery-Perks Law or Expected Likelihood Estimation
- How to find the right value of $\gamma$?
Good-Turing Estimator

- Intuition: Use n-grams seen once to estimate n-grams never seen and so on
- Compute $N_r$ (frequency of frequency $r$)

$$N_r = \sum_{w_i w_j:C(w_i w_j)} 1$$

- $N_0$ is the number of items with count 0
- $N_1$ is the number of items with count 1
- ...
Good-Turing Estimator

- For each $r$, compute an expected frequency estimate (smoothed count)

$$r' = C_{GT}(w_i, w_j) = (r + 1) \frac{N_{r+1}}{N_r}$$

- Replace MLE counts of seen bigrams with the expected frequency estimates and use those for probabilities

$$P_{GT}(w_i, w_j) = \frac{C_{GT}(w_i, w_j)}{N} \quad P_{GT}(w_j | w_i) = \frac{C_{GT}(w_i, w_j)}{C(w_i)}$$
Good-Turing Estimator

- What about an unseen bigram?

\[ r' = C_{GT} = (0 + 1) \frac{N_1}{N_0} = \frac{N_1}{N_0} \]

\[ P_{GT} = \frac{C_{GT}}{N} \]

- Do we know \( N_0 \)? Can we compute it for bigrams?

\( N_0 = V^2 \) — bigrams we have seen
# Good-Turing Estimator: Example

<table>
<thead>
<tr>
<th>$r$</th>
<th>$N_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>138741</td>
</tr>
<tr>
<td>2</td>
<td>25413</td>
</tr>
<tr>
<td>3</td>
<td>10531</td>
</tr>
<tr>
<td>4</td>
<td>5997</td>
</tr>
<tr>
<td>5</td>
<td>3565</td>
</tr>
<tr>
<td>6</td>
<td>...</td>
</tr>
</tbody>
</table>

$V = 14585$

Seen bigrams = 199252

\[
N_0 = (14585)^2 - 199252
\]

\[
C_{unseen} = \frac{N_1}{N_0} = 0.00065
\]

\[
P_{unseen} = \frac{N_1}{N_0 N} = 1.06 \times 10^{-9}
\]

**Note**: Assumes mass is uniformly distributed

$C(person\ she) = 2$  \quad C_{GT}(person\ she) = (2+1)(10531/25413) = 1.243$

$C(person) = 223$  \quad P(she|person) = C_{GT}(person\ she)/223 = 0.0056$
Good-Turing Estimator

- For each $r$, compute an expected frequency estimate (smoothed count)

  $$r' = C_{GT}(w_i, w_j) = (r + 1) \frac{N_{r+1}}{N_r}$$

- Replace MLE counts of seen bigrams with the expected frequency estimates and use those for probabilities

  $$P_{GT}(w_i, w_j) = \frac{C_{GT}(w_i, w_j)}{N}$$
  $$P_{GT}(w_j | w_i) = \frac{C_{GT}(w_i, w_j)}{C(w_i)}$$

What if $w_i$ isn’t observed?
Good-Turing Estimator

- Can’t replace all MLE counts
- What about $r_{\text{max}}$?
  - $N_{r+1} = 0$ for $r = r_{\text{max}}$
- Solution 1: Only replace counts for $r < k (~10)$
- Solution 2: Fit a curve $S$ through the observed $(r, N_r)$ values and use $S(r)$ instead
- For both solutions, remember to do what?
- Bottom line: the Good-Turing estimator is not used by itself but in combination with other techniques
Combining Estimators

- Better models come from:
  - Combining n-gram probability estimates from different models
  - Leveraging different sources of information for prediction

- Three major combination techniques:
  - Simple Linear Interpolation of MLEs
  - Katz Backoff
  - Kneser-Ney Smoothing
Linear MLE Interpolation

- Mix a trigram model with bigram and unigram models to offset sparsity
- Mix = Weighted Linear Combination

\[
P(w_k|w_{k-2}w_{k-1}) = \\
\lambda_1 P(w_k|w_{k-2}w_{k-1}) + \lambda_2 P(w_k|w_{k-1}) + \lambda_3 P(w_k)
\]

\[
0 \leq \lambda_i \leq 1 \quad \sum_i \lambda_i = 1
\]
Linear MLE Interpolation

- $\lambda_i$ are estimated on some held-out data set (not training, not test)

- Estimation is usually done via an EM variant or other numerical algorithms (e.g. Powell)
Backoff Models

- Consult different models in order depending on specificity (instead of all at the same time)
- The most detailed model for current context first and, if that doesn’t work, back off to a lower model
- Continue backing off until you reach a model that has some counts
Backoff Models

- Important: need to incorporate discounting as an integral part of the algorithm… Why?
- MLE estimates are well-formed…
- But, if we back off to a lower order model without taking something from the higher order MLEs, we are adding extra mass!
- Katz backoff
  - Starting point: GT estimator assumes uniform distribution over unseen events… can we do better?
  - Use lower order models!
Katz Backoff

Given a trigram “x y z”

\[ P_{katz}(z|x, y) = \begin{cases} 
    P_{GT}(z|x, y), & \text{if } C(x, y, z) > 0 \\
    \alpha(x, y)P_{katz}(z|y), & \text{otherwise}
\end{cases} \]

\[ P_{katz}(z|y) = \begin{cases} 
    P_{GT}(z|y), & \text{if } C(y, z) > 0 \\
    \alpha(y)P_{GT}(z), & \text{otherwise}
\end{cases} \]
Kneser-Ney Smoothing

- Observation:
  - Average Good-Turing discount for $r \geq 3$ is largely constant over $r$
  - So, why not simply subtract a fixed discount $D (\leq 1)$ from non-zero counts?

- Absolute Discounting: discounted bigram model, back off to MLE unigram model

- Kneser-Ney: Interpolate discounted model with a special “continuation” unigram model
Kneser-Ney Smoothing

Intuition

- Lower order model important only when higher order model is sparse
- Should be optimized to perform in such situations

Example

- $C(\text{Los Angeles}) = C(\text{Angeles}) = M$; $M$ is very large
- “Angeles” always and only occurs after “Los”
- Unigram MLE for “Angeles” will be high and a normal backoff algorithm will likely pick it in any context
- It shouldn’t, because “Angeles” occurs with only a single context in the entire training data
Kneser-Ney Smoothing

- Kneser-Ney: Interpolate discounted model with a special “continuation” unigram model
  - Based on appearance of unigrams in different contexts
  - Excellent performance, state of the art

\[
P_{KN}(w_k|w_{k-1}) = \frac{C(w_{k-1}w_k) - D}{C(w_{k-1})} + \beta(w_k)P_{CONT}(w_k)
\]

\[
P_{CONT}(w) = \frac{N(\bullet w)}{\sum_{w'} N(\bullet w')}
\]

\[N(\bullet w_i) = \text{number of different contexts } w_i \text{ has appeared in}\]

- Why interpolation, not backoff?
Explicitly Modeling OOV

- Fix vocabulary at some reasonable number of words
- During training:
  - Consider any words that don’t occur in this list as unknown or out of vocabulary (OOV) words
  - Replace all OOVs with the special word <UNK>
  - Treat <UNK> as any other word and count and estimate probabilities
- During testing:
  - Replace unknown words with <UNK> and use LM
  - Test set characterized by OOV rate (percentage of OOVs)
Evaluating Language Models

- Information theoretic criteria used
- Most common: Perplexity assigned by the trained LM to a test set
- Perplexity: How surprised are you on average by what comes next?
  - If the LM is good at knowing what comes next in a sentence ⇒ Low perplexity (lower is better)
  - Relation to weighted average branching factor
Computing Perplexity

- Given test set $W$ with words $w_1, \ldots, w_N$
- Treat entire test set as one word sequence
- Perplexity is defined as the probability of the entire test set normalized by the number of words
  
  $$PP(T) = P(w_1, \ldots, w_N)^{-1/N}$$

- Using the probability chain rule and (say) a bigram LM, we can write this as
  
  $$PP(T) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}}$$

- A lot easier to do with logprobs!
Practical Evaluation

- Use <s> and </s> both in probability computation
- Count </s> but not <s> in \( N \)
- Typical range of perplexities on English text is 50-1000
- Closed vocabulary testing yields much lower perplexities
- Testing across genres yields higher perplexities
- Can only compare perplexities if the LMs use the same vocabulary

<table>
<thead>
<tr>
<th>Order</th>
<th>Unigram</th>
<th>Bigram</th>
<th>Trigram</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP</td>
<td>962</td>
<td>170</td>
<td>109</td>
</tr>
</tbody>
</table>

Training: N=38 million, V~20000, open vocabulary, Katz backoff where applicable
Test: 1.5 million words, same genre as training
Typical “State of the Art” LMs

- **Training**
  - N = 10 billion words, V = 300k words
  - 4-gram model with Kneser-Ney smoothing

- **Testing**
  - 25 million words, OOV rate 3.8%
  - Perplexity ~50
Take-Away Messages

- LMs assign probabilities to sequences of tokens
- N-gram language models: consider only limited histories
- Data sparsity is an issue: smoothing to the rescue
  - Variations on a theme: different techniques for redistributing probability mass
  - Important: make sure you still have a valid probability distribution!
Scaling Language Models with MapReduce
Language Modeling Recap

- **Interpolation**: Consult all models at the same time to compute an interpolated probability estimate.

- **Backoff**: Consult the highest order model first and backoff to lower order model only if there are no higher order counts.

- **Interpolated Kneser Ney** (state-of-the-art)
  - Use absolute discounting to save some probability mass for lower order models.
  - Use a novel form of lower order models (count unique single word contexts instead of occurrences)
  - Combine models into a true probability model using interpolation

\[ P_{KN}(w_3|w_1, w_2) = \frac{C_{KN}(w_1w_2w_3) - D}{C_{KN}(w_1w_2)} + \lambda(w_1w_2)P_{KN}(w_3|w_2) \]
Questions for today

Can we efficiently train an IKN LM with terabytes of data?

Does it really matter?
Using MapReduce to Train IKN

- Step 0: Count words [MR]
- Step 0.5: Assign IDs to words [vocabulary generation] (more frequent → smaller IDs)
- Step 1: Compute $n$-gram counts [MR]
- Step 2: Compute lower order context counts [MR]
- Step 3: Compute unsmoothed probabilities and interpolation weights [MR]
- Step 4: Compute interpolated probabilities [MR]

[MR] = MapReduce job
Steps 0 & 0.5

Step 0

Step 0.5
## Steps 1-4

<table>
<thead>
<tr>
<th>Mapper Input</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Key</td>
<td>DocID</td>
<td>( n )-grams (&quot;a b c&quot;)</td>
<td>(&quot;a b c&quot;)</td>
<td>(&quot;a b&quot;)</td>
</tr>
<tr>
<td>Input Value</td>
<td>Document</td>
<td>( C_{\text{total}}(&quot;a b c&quot;) )</td>
<td>( C_{\text{KN}}(&quot;a b c&quot;) )</td>
<td><em>Step 3 Output</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mapper Output</th>
<th>Reducer Input</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intermediate Key</td>
<td>( n )-grams (&quot;a b c&quot;)</td>
<td>(&quot;a b c&quot;)</td>
<td>(&quot;a b&quot;) (history)</td>
<td>(&quot;c b a&quot;)</td>
</tr>
<tr>
<td>Intermediate Value</td>
<td>( C_{\text{doc}}(&quot;a b c&quot;) )</td>
<td>( C'_{\text{KN}}(&quot;a b c&quot;) )</td>
<td>(&quot;c&quot;, ( C_{\text{KN}}(&quot;a b c&quot;) ))</td>
<td>( (P'(&quot;a b c&quot;), \lambda(&quot;a b&quot;)) )</td>
</tr>
</tbody>
</table>

| Mapper Output | | | | |
| Partitioning | \("a b c"\) | \("a b c"\) | \("a b"\) | \("c b"\) |

| Reducer Output | | | | |
| Output Value | \( C_{\text{total}}("a b c") \) | \( C_{\text{KN}}("a b c") \) | \("c", \( P'("a b c"), \lambda("a b")) \) | \( (P_{\text{KN}}("a b c"), \lambda("a b")) \) |

### Count n-grams

### Count contexts

### Compute unsmoothed probs AND interp. weights

### Compute Interp. probs

All output keys are always the same as the intermediate keys.

I only show trigrams here but the steps operate on bigrams and unigrams as well.
### Steps 1-4

<table>
<thead>
<tr>
<th>Mapper Input</th>
<th>Mapper Output</th>
<th>Reducer Input</th>
<th>Reducer Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Key</strong></td>
<td>DocID</td>
<td>n-grams “a b c”</td>
<td>“a b c”</td>
</tr>
<tr>
<td><strong>Input Value</strong></td>
<td>Document</td>
<td>C\text{total}(“a b c”)</td>
<td>C\text{KN}(“a b c”)</td>
</tr>
<tr>
<td><strong>Intermediate Key</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Intermediate Value</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Partitioning</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Output Value</strong></td>
<td>C\text{total}(“a b c”)</td>
<td>C\text{KN}(“a b c”)</td>
<td>(“c”, P(“a b c”), λ(“a b”))</td>
</tr>
<tr>
<td><strong>Count n-grams</strong></td>
<td><strong>Count contexts</strong></td>
<td><strong>Compute unsmoothed probs AND interp. weights</strong></td>
<td><strong>Compute Interp. probs</strong></td>
</tr>
</tbody>
</table>

Details are not important!

5 MR jobs to train IKN (expensive)!

IKN LMs are big!
(interpolation weights are context dependent)

Can we do something that has better behavior at scale in terms of time and space?

All output keys are always the same as the intermediate keys
I only show trigrams here but the steps operate on bigrams and unigrams as well
Let’s try something stupid!

- Simplify backoff as much as possible!

- Forget about trying to make the LM be a true probability distribution!

- Don’t do any discounting of higher order models!

- Have a single backoff weight independent of context! \([\alpha(\cdot) = \alpha]\)

\[
S(w_3|w_2, w_1) = \frac{c(w_1w_2w_3)}{c(w_1w_2)} \quad \text{if } c(w_1w_2w_3) > 0
\]
\[
= \alpha S(w_3|w_2) \quad \text{otherwise}
\]
\[
S(w_3) = \frac{c(w_3)}{N} \quad \text{(recursion ends at unigrams)}
\]

“Stupid Backoff (SB)”
Using MapReduce to Train SB

- Step 0: Count words [MR]

- Step 0.5: Assign IDs to words [vocabulary generation]
  (more frequent → smaller IDs)

- Step 1: Compute $n$-gram counts [MR]

- Step 2: Generate final LM “scores” [MR]

[MR] = MapReduce job
Steps 0 & 0.5

Step 0

Step 0.5

Counting in the map phase is a minor optimization; see text.
## Steps 1 & 2

<table>
<thead>
<tr>
<th>Mapper Input</th>
<th>Step 1</th>
<th>Step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Key</td>
<td>DocID</td>
<td>First two words of (n)-grams “a b c” and “a b” (“a b”)</td>
</tr>
<tr>
<td>Input Value</td>
<td>Document</td>
<td>(C_{\text{total}}(“a b c”))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mapper Output</th>
<th>Reducer Input</th>
<th>Reducer Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intermediate Key</td>
<td>(n)-grams “a b c”</td>
<td>“a b c”</td>
</tr>
<tr>
<td>Intermediate Value</td>
<td>(C_{\text{doc}}(“a b c”))</td>
<td>(S(“a b c”))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partitioning</th>
<th>Output Value</th>
<th>Count (n)-grams</th>
<th>Compute LM scores</th>
</tr>
</thead>
<tbody>
<tr>
<td>first two words (why?) “a b”</td>
<td>“a b c”, (C_{\text{total}}(“a b c”))</td>
<td>(S(“a b c”)) [write to disk]</td>
<td></td>
</tr>
<tr>
<td>last two words “b c”</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- All unigram counts are replicated in all partitions in both steps
- The clever partitioning in Step 2 is the key to efficient use at runtime!
- The trained LM model is composed of partitions written to disk
Which one wins?

<table>
<thead>
<tr>
<th></th>
<th>target</th>
<th>webnews</th>
<th>web</th>
</tr>
</thead>
<tbody>
<tr>
<td># tokens</td>
<td>237M</td>
<td>31G</td>
<td>1.8T</td>
</tr>
<tr>
<td>vocab size</td>
<td>200k</td>
<td>5M</td>
<td>16M</td>
</tr>
<tr>
<td># n-grams</td>
<td>257M</td>
<td>21G</td>
<td>300G</td>
</tr>
<tr>
<td>LM size (SB)</td>
<td>2G</td>
<td>89G</td>
<td>1.8T</td>
</tr>
<tr>
<td>time (SB)</td>
<td>20 min</td>
<td>8 hours</td>
<td>1 day</td>
</tr>
<tr>
<td>time (KN)</td>
<td>2.5 hours</td>
<td>2 days</td>
<td>–</td>
</tr>
<tr>
<td># machines</td>
<td>100</td>
<td>400</td>
<td>1500</td>
</tr>
</tbody>
</table>

Table 2: Sizes and approximate training times for 3 language models with Stupid Backoff (SB) and Kneser-Ney Smoothing (KN).
Which one wins?

Can’t compute perplexity for SB. Why?

Why do we care about 5-gram coverage for a test set?
Which one wins?

BLEU is a measure of MT performance.

Not as stupid as you thought, huh?
Take away

- The MapReduce paradigm and infrastructure make it simple to scale algorithms to web scale data.
- At Terabyte scale, efficiency becomes really important!
- When you have a lot of data, a more scalable technique (in terms of speed and memory consumption) can do better than the state-of-the-art even if it’s stupider!

“The difference between genius and stupidity is that genius has its limits.”
- Oscar Wilde

“The dumb shall inherit the cluster”
- Nitin Madnani
Questions?