Data-Intensive Information Processing Applications — Session #9

Language Models

Nitin Madnani
University of Maryland

Tuesday, April 6, 2010

Source: Wikipedia (Japanese rock garden)

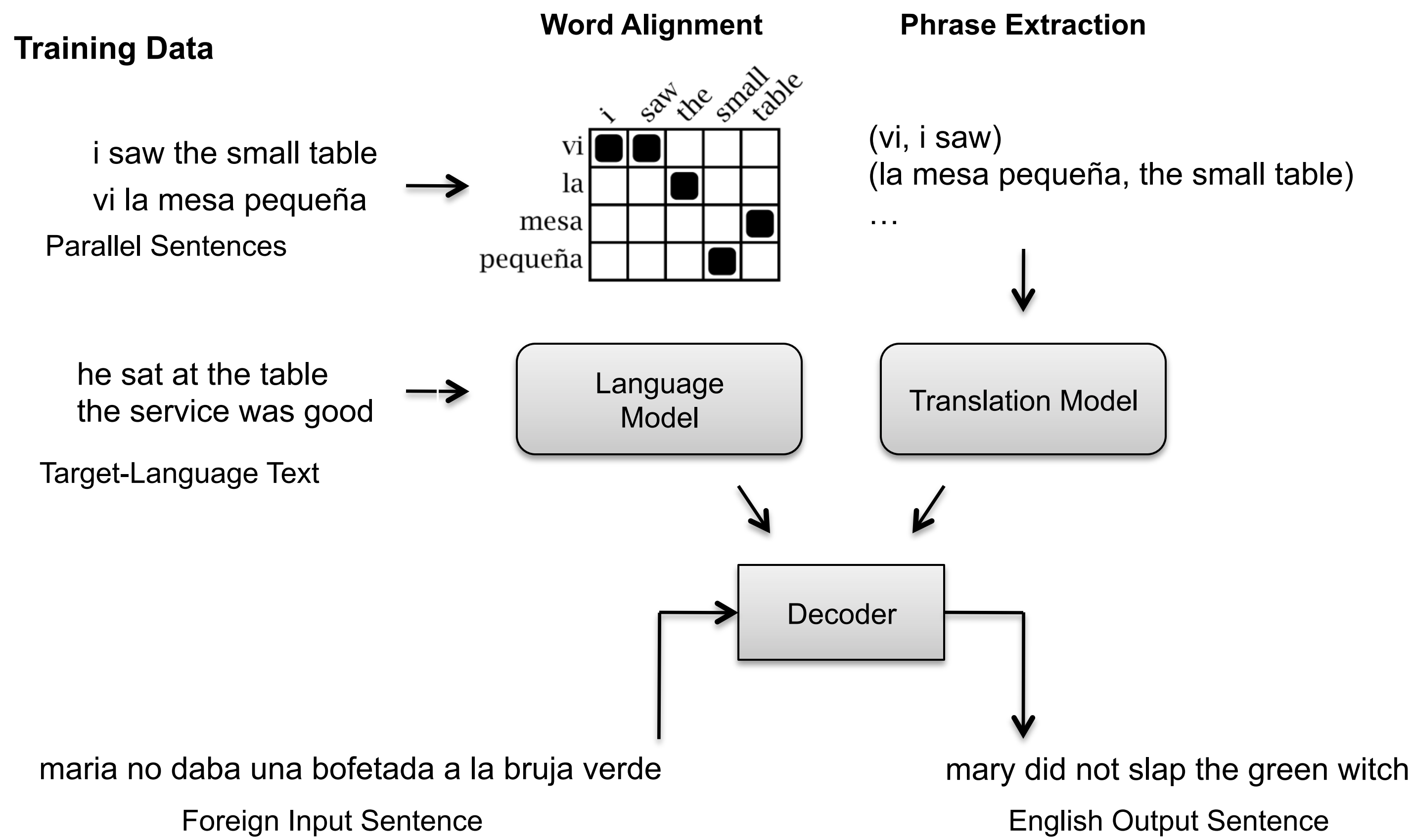
Today's Agenda

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

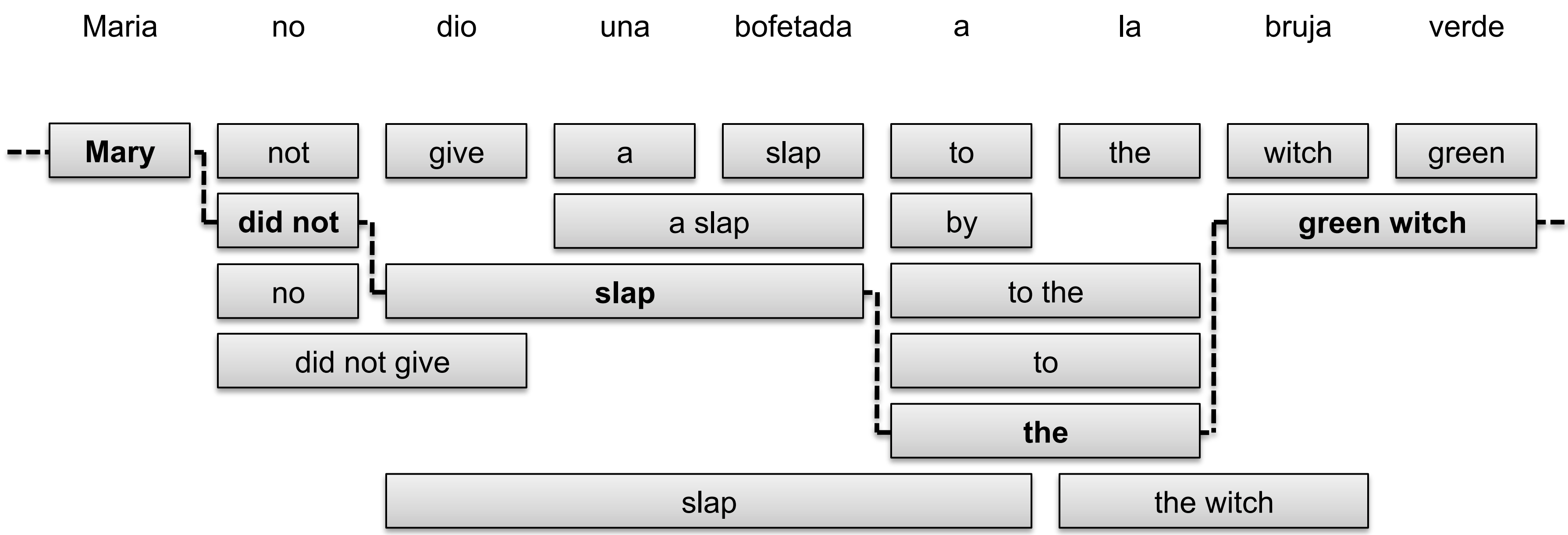
N-Gram Language Models

- 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



SMT: The role of the LM



N-Gram Language Models

N=1 (unigrams)

This is a sentence

Unigrams:
This,
is,
a,
sentence

Sentence of length *s*, how many unigrams?

N-Gram Language Models

N=2 (bigrams)

This is a sentence

Bigrams:
This is,
is a,
a sentence

Sentence of length *s*, how many bigrams?

N-Gram Language Models

N=3 (trigrams)

$\boxed{\text{This } \boxed{\text{is a}} \text{ sentence}}$

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

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

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

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

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

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

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)
  - $C(w_1,...,w_k)$ = frequency of n-gram $w_1, ..., w_k$ in training data
  - $P(w_1, ..., w_k)$ = probability estimate for n-gram $w_1 ... w_k$
  - $P(w_k|w_1, ..., w_{k-1})$ = conditional probability of producing w_k given the history $w_1, ... w_{k-1}$

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: $P(w_i, w_j) = \frac{C(w_i, w_j)}{N}$

Why not just substitute $P(w_i)$?

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

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

Example: Bigram Language Model

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

Training Corpus

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: $P(w_i, w_j) = \frac{C(w_i, w_j)}{N}$

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

Let's revisit this issue…

Why not just substitute *$P(w_i)$*?

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

More Context, More Work

- Larger N = more context
  - Lexical co-occurrences
  - Local syntactic relations
- More context is better?
- Larger N = 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 | <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

P(I like ham)

= P(I | <s>) P(like | I) P(ham | like) P(</s> | 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

$$P_{MLE}(w_i) = \frac{C(w_i)}{N} \longrightarrow P_{LAP}(w_i) = \frac{C(w_i)+1}{N+V}$$

Bigrams

$$P_{MLE}(w_i, w_j) = \frac{C(w_i, w_j)}{N} \longrightarrow P_{LAP}(w_i, w_j) = \frac{C(w_i, w_j)+1}{N+V^2}$$

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

$$
\begin{aligned}
C_{LAP}(w_i) &= P_{LAP}(w_i)N \\
C_{LAP}(w_i, w_j) &= P_{LAP}(w_i, w_j)N
\end{aligned}
$$

Relative Discount

$$
\begin{aligned}
d_1 &= \frac{C_{LAP}(w_i)}{C(w_i)} \\
d_2 &= \frac{C_{LAP}(w_i, w_j)}{C(w_i, w_j)}
\end{aligned}
$$

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 γ 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 γ?

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

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

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?

$$N_0 = V^2 - \text{bigrams we have seen}$$

Good-Turing Estimator: Example

r	N_r
1	138741
2	25413
3	10531
4	5997
5	3565
6	...

V = 14585

Seen bigrams =199252

$$N_0 = (14585)^2 - 199252$$

$$C_{unseen} = N_1 / N_0 = 0.00065$$

$$P_{unseen} = N_1 / (N_0 N) = 1.06 \times 10^{-9}$$

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

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

What if w_i isn't observed?

Good-Turing Estimator

- Can't replace all MLE counts
- What about r_{max}?
  - $N_{r+1} = 0$ for $r = r_{max}$
- 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

$$P(w_k|w_{k-2}w_{k-1}) =$$

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

$$0 <= \lambda_i <= 1 \qquad \sum_i \lambda_i = 1$$

Linear MLE Interpolation

- λ_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 (≤ 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(Los Angeles) = C(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

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

$$P_{CONT}(w_i) = \frac{N(\bullet\ w_i)}{\sum_{w'} N(\bullet\ w')}$$

$N(\bullet\ w_i)$ = number of different contexts w_i 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

Order	Unigram	Bigram	Trigram
PP	962	170	109

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

$$P_{KN}(w_3|w_1, w_2) = \frac{C_{KN}(w_1 w_2 w_3) - D}{C_{KN}(w_1 w_2)} + \lambda(w_1 w_2) P_{KN}(w_3|w_2)$$

Questions for today

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

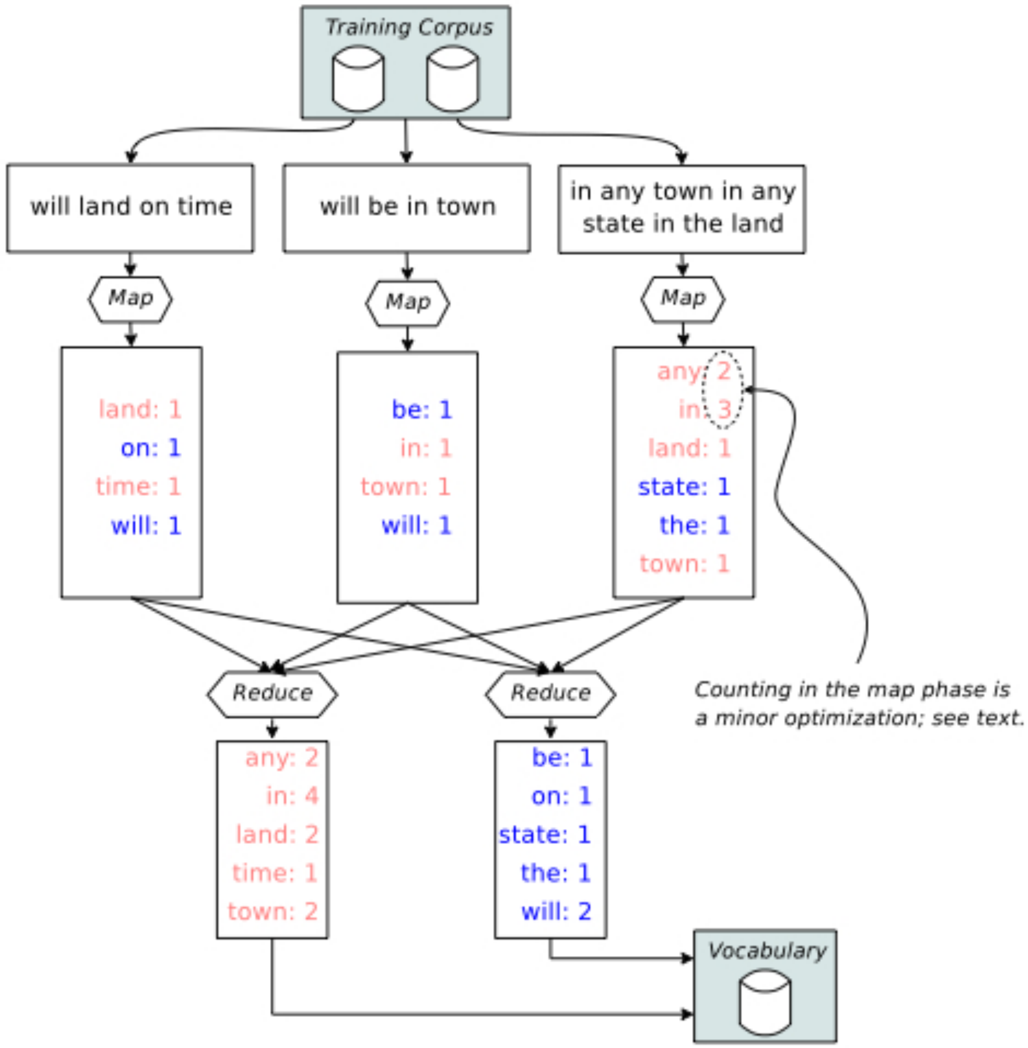
Does it really matter?

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

		Step 1	**Step 2**	**Step 3**	**Step 4**
Mapper Input	Input Key	DocID	*n*-grams “a b c”	“a b c”	“a b”
	Input Value	Document	C_{total}(“a b c”)	C_{KN}(“a b c”)	_Step 3 Output_
Mapper Output Reducer Input	Intermediate Key	*n*-grams “a b c”	“a b c”	“a b” (history)	“c b a”
	Intermediate Value	C_{doc}(“a b c”)	C'_{KN}(“a b c”)	(“c”, C_{KN}(“a b c”))	(P’(“a b c”), λ(“a b”))
	Partitioning	“a b c”	“a b c”	“a b”	“c b”
Reducer Output	Output Value	C_{total}(“a b c”)	C_{KN}(“a b c”)	(“c”, P’(“a b c”), λ(“a b”))	(P_{KN}(“a b c”), λ(“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

		Step 1	Step 2	Step 3	Step 4
Mapper Input	Input Key	DocID	*n*-grams "a b c"	"a b c"	"a b"
	Input Value	Document	C_{total}("a b c")	C_{KN}("a b c")	_Step 3 Output_
Mapper Output Reducer Input					
	Intermediate Key				"c b a"
	Intermediate Value				'a b c"), λ("a b"))
	Partitioning				"c b"
Reducer Output					
	Output Value	C_{total}("a b c")	C_{KN}("a b c")	("c", P'("a b c"), λ("a b"))	(P_{KN}("a b c"), λ("a b"))
		Count n-grams	**Count contexts**	**Compute unsmoothed probs AND interp. weights**	**Compute Interp. probs**

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! [α(•) = α]

$$S(w_3|w_2,w_1) = \frac{c(w_1 w_2 w_3)}{c(w_1 w_2)} \quad \text{if } c(w_1 w_2 w_3) > 0$$

$$= \alpha S(w_3|w_2) \quad \text{otherwise}$$

$$S(w_3) = \frac{c(w_3)}{N} \quad \text{(recursion ends at unigrams)}$$

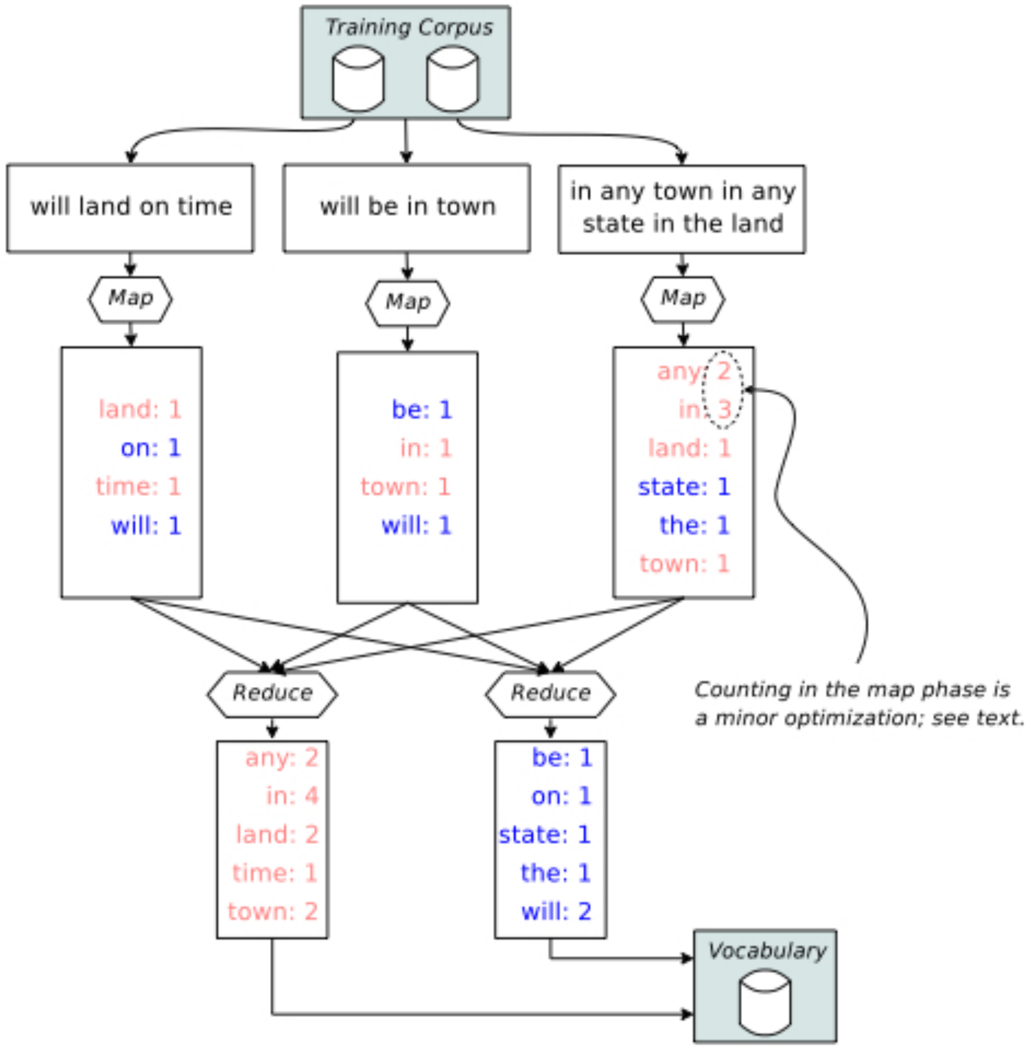
"Stupid Backoff (SB)"

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

Steps 1 & 2

		Step 1	Step 2
Mapper Input	Input Key	DocID	First two words of *n*-grams "a b c" and "a b" ("a b")
	Input Value	Document	C_{total}("a b c")
Mapper Output Reducer Input	Intermediate Key	*n*-grams "a b c"	"a b c"
	Intermediate Value	C_{doc}("a b c")	S("a b c")
	Partitioning	first two words (why?) "a b"	last two words "b c"
Reducer Output	Output Value	"a b c", C_{total}("a b c")	S("a b c") [write to disk]
		Count n-grams	**Compute LM scores**

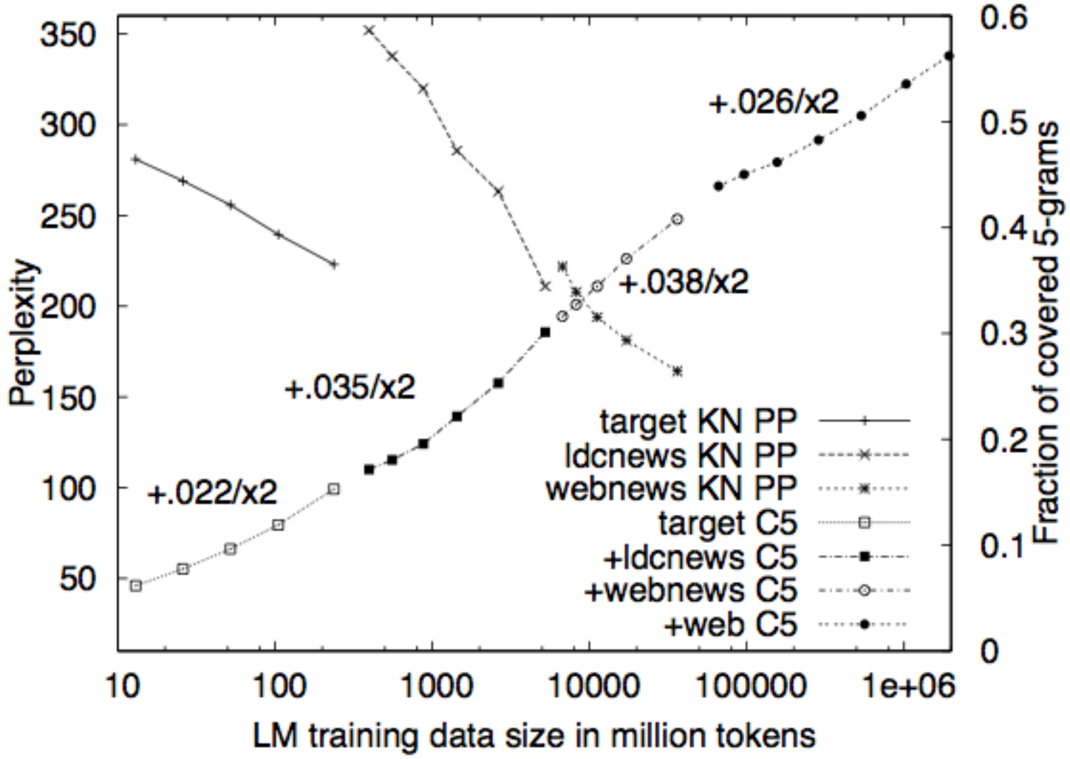
- **All unigram counts are replicated in all partitions in both steps**
- **The clever partitioning in Step 2 is the key to efficient use at runtime!**
- **The trained LM model is composed of partitions written to disk**

Which one wins?

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

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

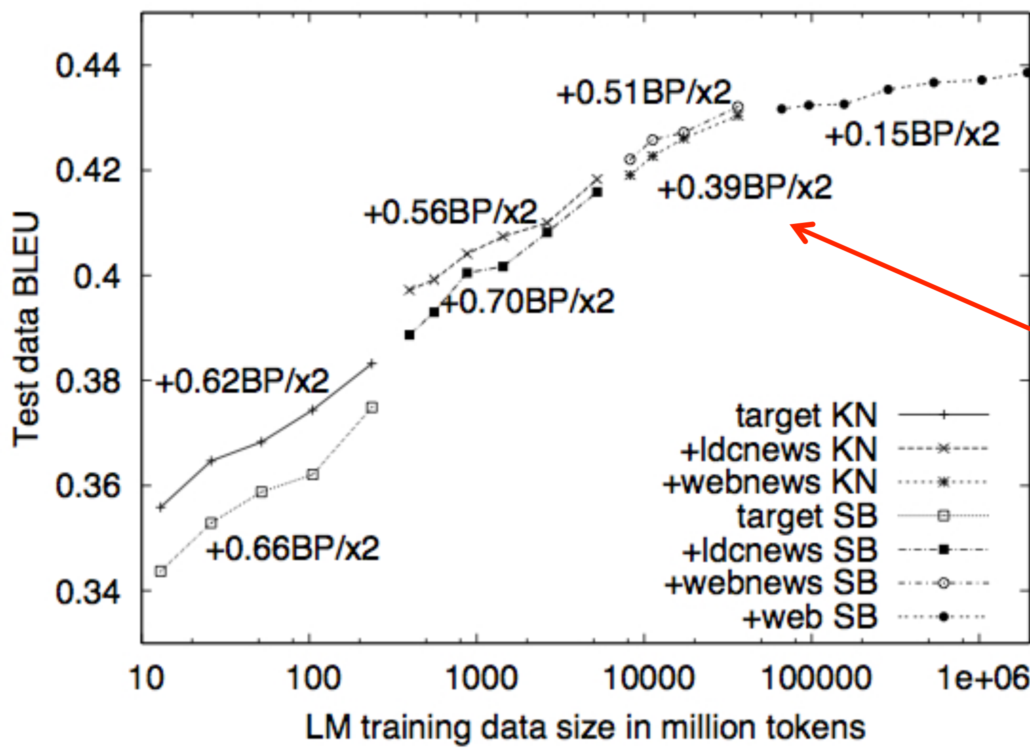
Which one wins?



Can't compute perplexity for SB. Why?

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

Which one wins?



SB overtakes IKN

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

Source: Wikipedia (Japanese rock garden)