Language models for speech recognition

Bhiksha Raj and Rita Singh

The need for a “language” scaffolding

- “I’m at a meeting” or “I met a meeting” ??
- The two are acoustically nearly identical
- Need a means of deciding which of the two is correct
  - Or more likely to be correct
- This is provided by a “language” model

- The “language” model may take many forms
  - Finite state graphs
  - Context-free grammars
  - Statistical Language Models

What The Recognizer Recognizes

- The Recoginzer ALWAYS recognizes one of a set of sentences
  - Word sequences
- E.g. we may want to recognize a set of commands:
  - Open File
  - Edit File
  - Close File
  - Delete File
  - Delete All Files
  - Delete Marked Files
  - Close All Files
  - Close Marked Files
- The recognizer *explicitly* only attempts to recognize these sentences

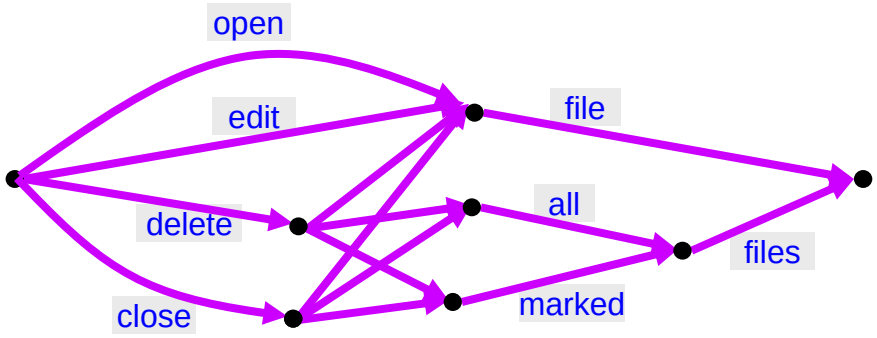
What The Recognizer Recognizes

- Simple approach: Construct HMMs for each of the following:
  - Open File
  - Edit File
  - Close File
  - Delete File
  - Delete All Files
  - Delete Marked Files
  - Close All Files
  - Close Marked Files

  - HMMs may be *composed* using word or phoneme models
- Recognize what was said, using the same technique used for word recognition

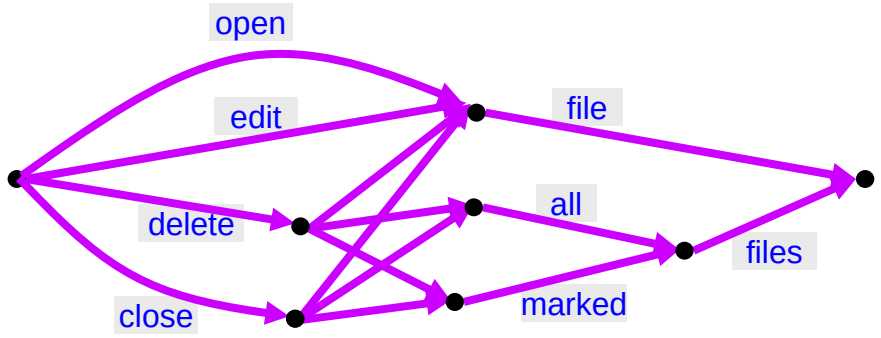
A More Compact Representation

- The following
  - Open File
  - Edit File
  - Close File
  - Delete File
  - Delete All Files
  - Delete Marked Files
  - Close All Files
  - Close Marked Files
- .. Can all be collapsed into the following graph



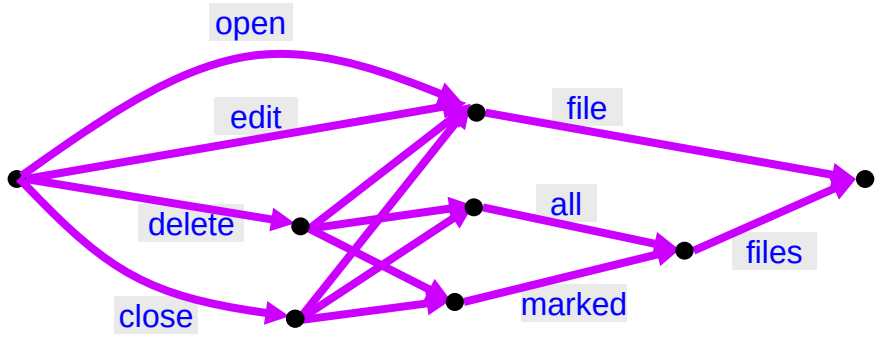
A More Compact Representation

- Build an HMM for the graph below such that each edge was replaced by the HMM for the corresponding word
- The best state sequence through the graph will automatically pass along only one of the valid paths from beginning to the end
  - We will show this later
- So simply running Viterbi on the HMM for the graph is equivalent to performing best-state-sequence-probability based recognition from the HMMs for the individual commands
  - Full probability computation based recognition wont work



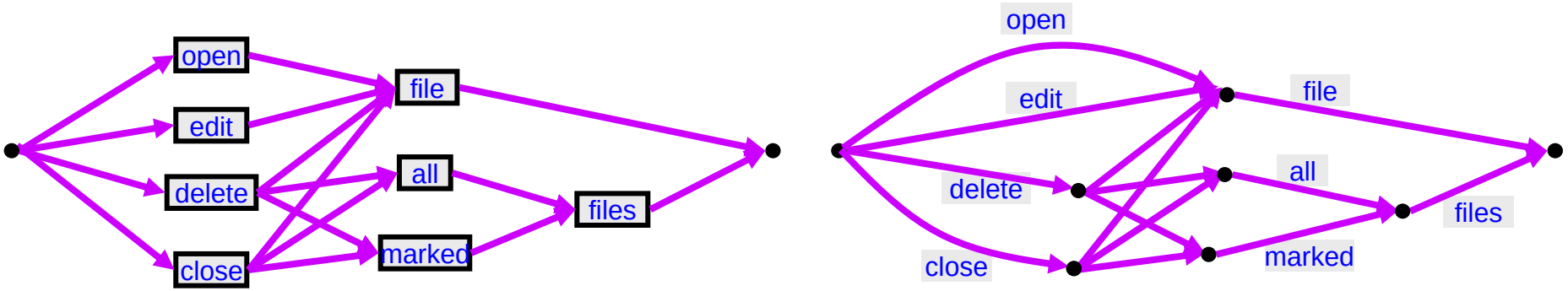
Economical Language Repersentations

- If we limit ourselves to using Viterbi based recogintion, simply representing the complete set of sentences as a graph is the most effective representation
  - The graph is directly transformed to an HMM that can be used for recognition
  - This only works when the set of all possible sentences is expressible as a small graph



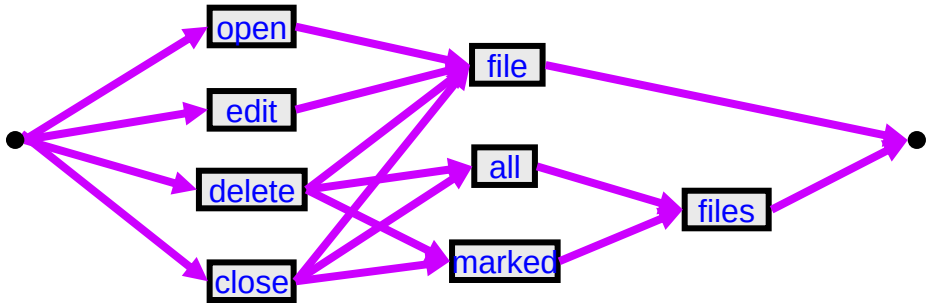
Finite State Graph

- A finite-state graph represents the set of all allowed sentences as a graph
  - Word sequences that do not belong to this "allowed" set will not be recognized
    - They will actually be mis-recognized as something from the set
- An example FSG to recognize our computer commands

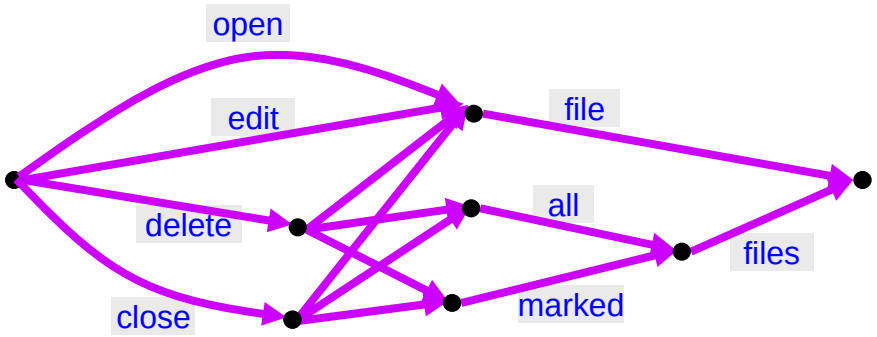


FSG Specification

- The FSG specification may use one of two equivalent ways:
  - Words at nodes and edges representing sequencing constraints



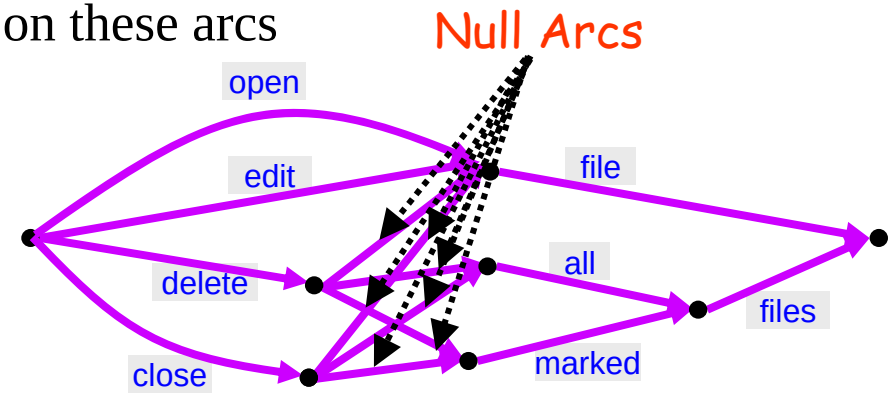
  - Words on edges; nodes represent abstract states



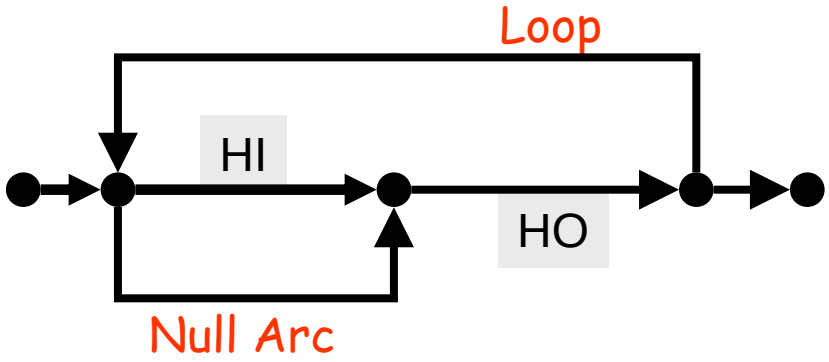
  - The latter is more common

FSG with Null Arcs and Loops

- FSGs may have "null" arcs
  - No words produced on these arcs

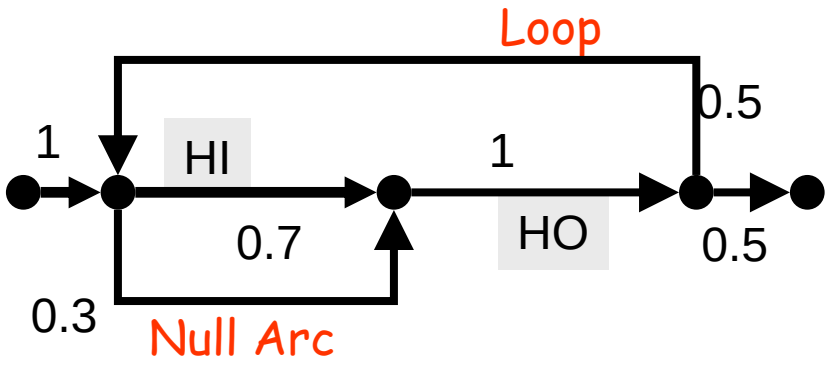


- FSGs may have loops
  - Allows for infinitely long word sequences



Probabilistic FSG

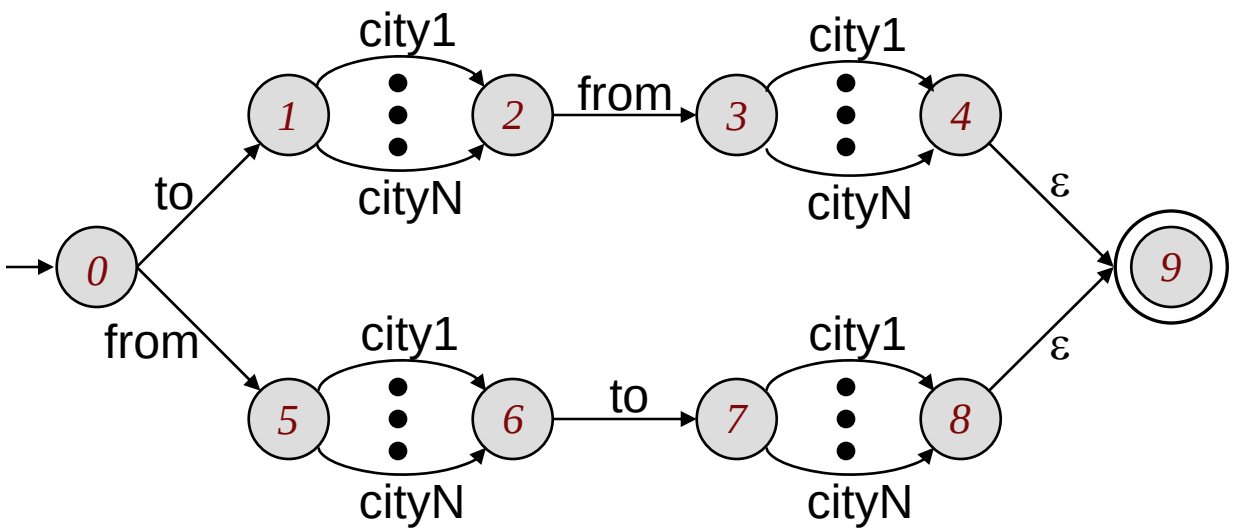
- By default, edges have no explicit weight associated with them
  - Effectively having a weight of 1 (multiplicatively)
- The edges can have probabilities associated with them
  - Specifying the probability of a particular edge being taken from a node



CMUSphinx FSG Specification

- "Assembly language" for specifying FSGs
  - Low-level
  - Most standards should compile down to this level
- Set of N states, numbered 0 .. N-1
- Transitions:
  - Emitting or non-emitting (aka null or epsilon)
  - Each emitting transition emits one word
  - Fixed probability $0 < p <= 1$.
- Words are on edges (transitions)
  - Null transitions have no words associated with them
- One start state, and one final state
  - Null transitions can effectively give you as many as needed

An FSG Example




```
FSG_BEGIN     leg

NUM_STATES   10
START_STATE   0
FINAL_STATE    9

# Transitions
T   0  1   0.5   to
T   1  2   0.1   city1 …
T   1  2   0.1   cityN
T   2  3   1.0   from
T   3  4   0.1   city1 …
T   3  4   0.1   cityN
T   4  9   1.0
T   0  5   0.5   from
T   5  6   0.1   city1 …
T   5  6   0.1   cityN
T   6  7   1.0   to
T   7  8   0.1   city1 …
T   7  8   0.1   cityN
T   8  9   1.0
FSG_END
```

Context-Free Grammars

- Context-free grammars specify production rules for the language in the following form:
  - RULE = Production
- Rules may be specified in terms of other production rules
  - RULE1 = Production1
  - RULE2 = word1 RULE1 Production2
- This is a *context-free* grammar since the production for any rule does not depend on the context that the rule occurs in
  - E.g. the production of RULE1 is the same regardless of whether it is preceded by word1 or not
- A production is a pattern of words
- The precise formal definition of CFGs is outside the scope of this talk

Context-Free Grammars for Speech Recognition

- Various forms of CFG representations have been used
- BNF: Rules are of the form <RULE> ::= PRODUCTION

- Example (from wikipedia):
  - <postal-address> ::= <name-part> <street-address> <zip-part>
  - <name-part> ::= <personal-part> <last-name> <opt-jr-part> <EOL> | <personal-part> <name-part>
  - <personal-part> ::= <first-name> | <initial> "."
  - <street-address> ::= <opt-apt-num> <house-num> <street-name> <EOL>
  - <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>
  - <opt-jr-part> ::= "Sr." | "Jr." | <roman-numeral> | “”
- The example is incomplete. To complete it we need additional rules like:
  - <personal-part> ::= “Clark” | “Lana” | “Steve”
  - <last-name> ::= “Kent” | “Lang” | “Olsen”
- Note: Production rules include sequencing (AND) and alternatives (OR)

CFG: EBNF

- Extended BNF grammar specifications allow for various shorthand
- Many variants of EBNF. The most commonly used one is the W3C definition
- Some shorthand rules introduced by W3C EBNF:
  - X? specifies that X is optional
    - E.g. Formal_Name = "Mr." "Clark"? "Kent"
    - "Clark" may or may not be said
  - Y+ specifies one or more repetitions e.g.
    - Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"
    - Integer = Digit+
  - Z* specifies zero or more repetitions e.g.
    - Alphabet = "a" | "b" | "c" | "d" | "e" | "f" | "g"
    - Registration = Alphabet+ Digit*
- The entire set of rules is available from W3C

CFG: ABNF

- Augmented BNF adds a few new rule expressions
- The key inclusion is the ability to specify a *number* of repetitions
  - N*MPattern says “Pattern” can occur a minimum of N and a maximum of M times
  - N*Pattern states “Pattern” must occur a minimum of N times
  - *MPattern specifies that “Pattern” can occur at most M times
- Some changes in syntax
  - “/” instead of “|”
  - Grouping permitted with parantheses
  - Production rule names are often indicated by “$”
  - E.g.
    - $Digit = “0” / “1” / “2” / “3” / “4” / “5” / “6” / “7” / “8” / “9” / “0”
    - $Alphabet = “a” / “b” / “c” / “d” / “e” / “f” / “g”
    - $Registration = 3*$Alphabet *5$Digit

CFG: JSGF

- JSGF is a form of ABNF designed specifically for speech applications
- Example:

```
grammar polite;

public <startPolite> = [please | kindly | could you | oh mighty
   computer];
public <endPolite> = (please | thanks | thank you) [very* much];
```

- The grammar name specifies a namespace
- "Public" rules can directly be used by a recognizer
  - E.g. if the grammar specified to the recognizer is <endPolite>, the set of "sentences" being recognized are "Please", "Thanks", "Please much", "Please very much", "Please very very much" etc.
- Private rules can also be specified. These can only be used in the composition of other rules in the grammar
  - Cannot be used in other grammars or by the recognizer

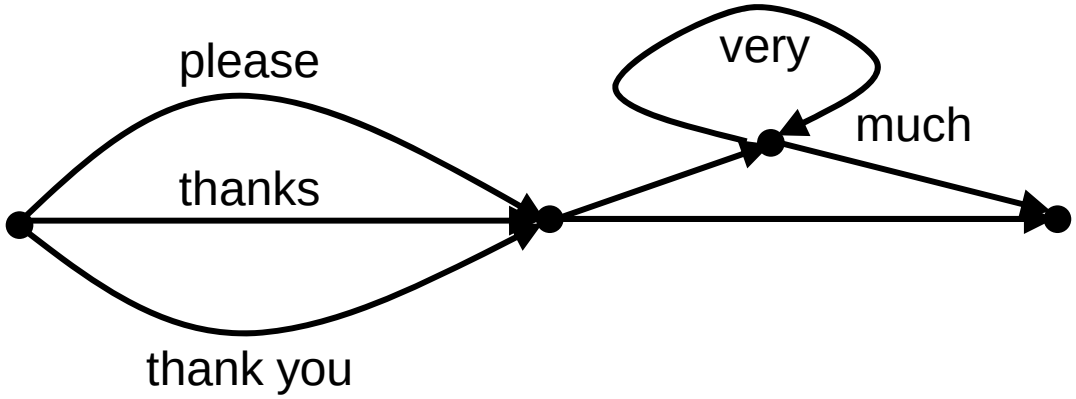
Context-Free Grammars

- CFGs can be loopy
  - $RULE = [$RULE] word (Left Recursion)
  - $RULE = word [$RULE]
  - Both above specify an arbitrarily long sequence of “word”
  - Other more complex recursions are possible

CFGs in the Recognizer

- Internally, the CFG is converted to a Finite State Graph
  - Most efficient approach
  - Alternate approach may use a "production" approach where the grammar is used to *produce* word sequences that are hypothesized
  - The latter approach can be very inefficient

- `<endPolite> = (please | thanks | thank you) [very* much];`



- Many algorithms for conversion from the one to the other

Restrictions on CFG for ASR

- CFGs must be representable as finite state machines
  - Not all CFGs are finite state
  - $RULE = word1^N $RULE word2^N | word1 word2
    - Represents the language word1^N word2^N
    - Only produces sequences of the kind:
      word1 word1 word1 (N times) word2 word (N times)

- CFGs that are not finite state are usually approximated to finite state machines for ASR
  - $RULE = word1^N $RULE word2^N | word1 word2 approximated as
  - $RULE = (word1 word2) | (word1 word1 word2 word2) | (word1 word1 word1 word2 word2 word2)

CFGs and FSGs

- CFGs and FSGs are typically used where the recognition "language" can be prespecified
  - E.g. command-and-control applications, where the system may recognize only a fixed set of things
  - Address entry for GPS: the precise set of addresses is known
    - Although the set may be very large
- Typically, the application provides the grammar
  - Although algorithms do exist to learn them from large corpora
- The problem is the rigidity of the structure: the sysem *cannot* recognize any word sequence that does not conform to the grammar
- For recognizing natural language we need models that represent *every possible word sequence*
  - For this we look to the Bayesian specification of the ASR problem

Natural Language Recognition

- In "natural" language of any kind, the number of sentences that can be said is infinitely large
  - Cannot be enumerated
  - Cannot be characterized by a simple graph or grammar

- Solved by realizing that recognition is a problem of *Bayesian Classification*
- Try to find the word sequence such that

$$word_1, word_2, ... = \arg\max_{w_1, w_2, ...} \{P(w_1, w_2, .. \mid X)\}$$

$$word_1, word_2, ... = \arg\max_{w_1, w_2, ...} \{P(X \mid w_1, w_2, ..) P(w_1, w_2, ..)\}$$

The Bayes classifier for speech recognition

- The Bayes classification rule for speech recognition:

$$word_1, word_2, ... = \arg\max_{w_1, w_2, ...} \{P(X \mid w_1, w_2, ..)P(w_1, w_2, ..)\}$$

- P(X | w_1, w_2, …) measures the likelihood that speaking the word sequence w_1, w_2 … could result in the data (feature vector sequence) *X*

- P(w_1, w_2 …) measures the probability that a person might actually utter the word sequence w_1, w_2 ….
  - This will be 0 for impossible word sequences

- In theory, the probability term on the right hand side of the equation must be computed for every possible word sequence
  - It will be 0 for impossible word sequences

- In practice this is often impossible
  - There are infinite word sequences

Speech recognition system solves

$$word_1, word_2, \ldots, word_N =$$

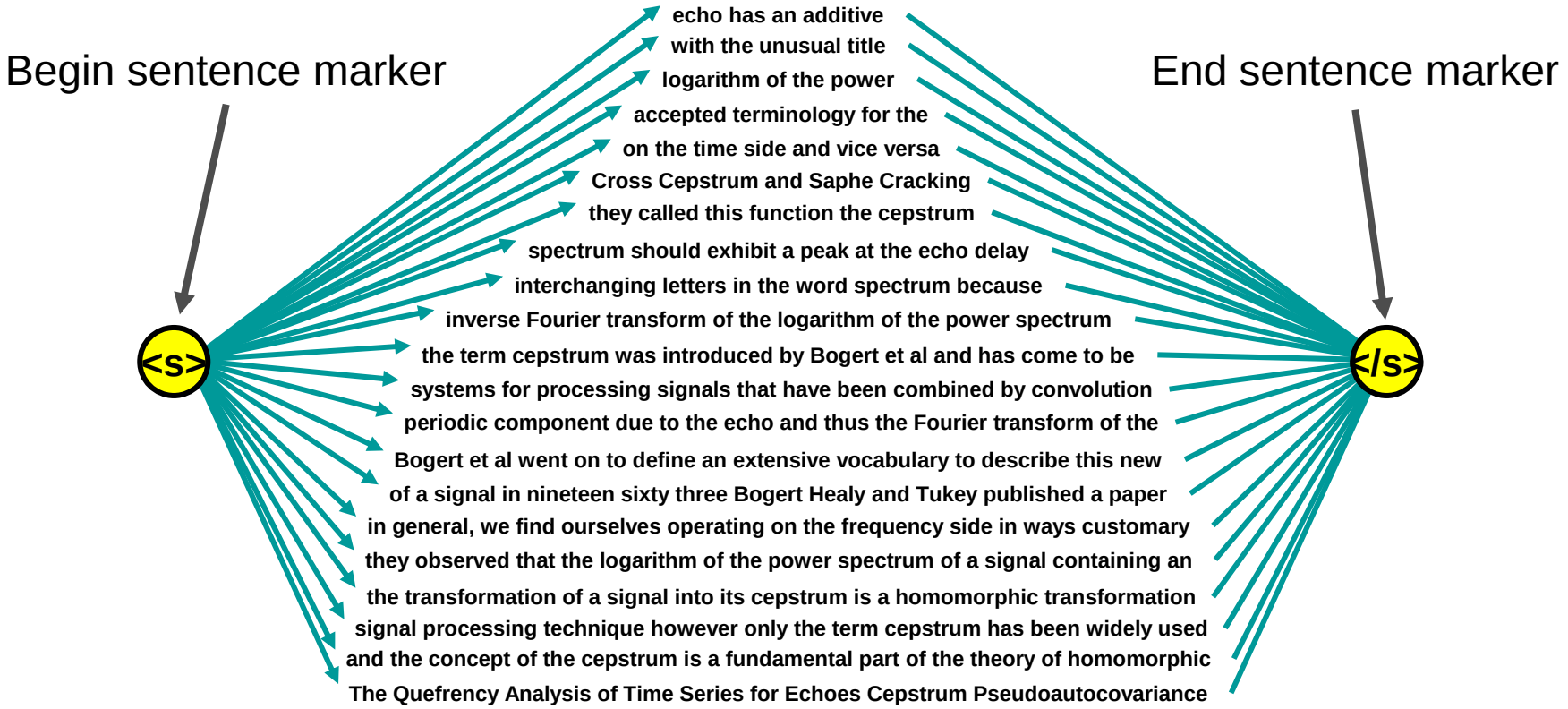
$$\arg\max_{wd_1, wd_2, \ldots, wd_N} \{P(signal|wd_1, wd_2, \ldots, wd_N) P(wd_1, wd_2, \ldots, wd_N)\}$$

Acoustic model

For HMM-based systems this is an HMM

Lanugage model

Bayes' Classification: A Graphical View



.....

- There will be one path for every possible word sequence
- A priori probabilitiy for a word sequence can be applied anywhere along the path representing that word sequence.
- It is the structure and size of this graph that determines the feasibility of the recognition task

A left-to-right model for the langauge

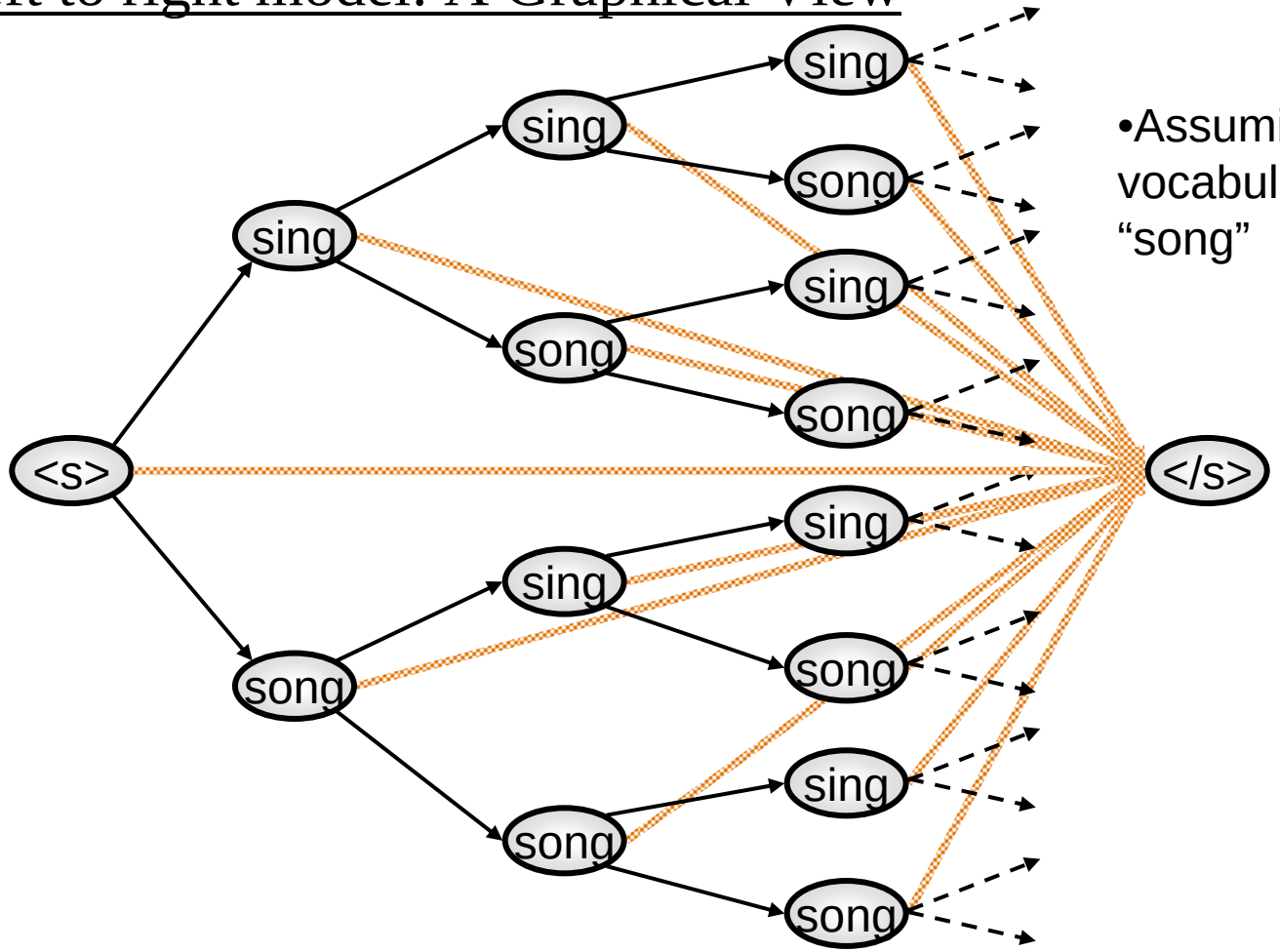
- A factored representation of the a priori probability of a word sequence

  P(<s> word1 word2 word3 word4…</s>) =
  P(<s>) P(word1 | <s>) P(word2 | <s> word1) P(word3 | <s> word1 word2)…

- This is a left-to-right factorization of the probability of the word sequence
  - The probability of a word is assumed to be dependent only on the words preceding it
  - This probability model for word sequences is as accurate as the earlier whole-word-sequence model, in theory
- It has the advantage that the probabilities of words are applied left to right – this is perfect for speech recognition
- P(word1 word2 word3 word4 …) is incrementally obtained :

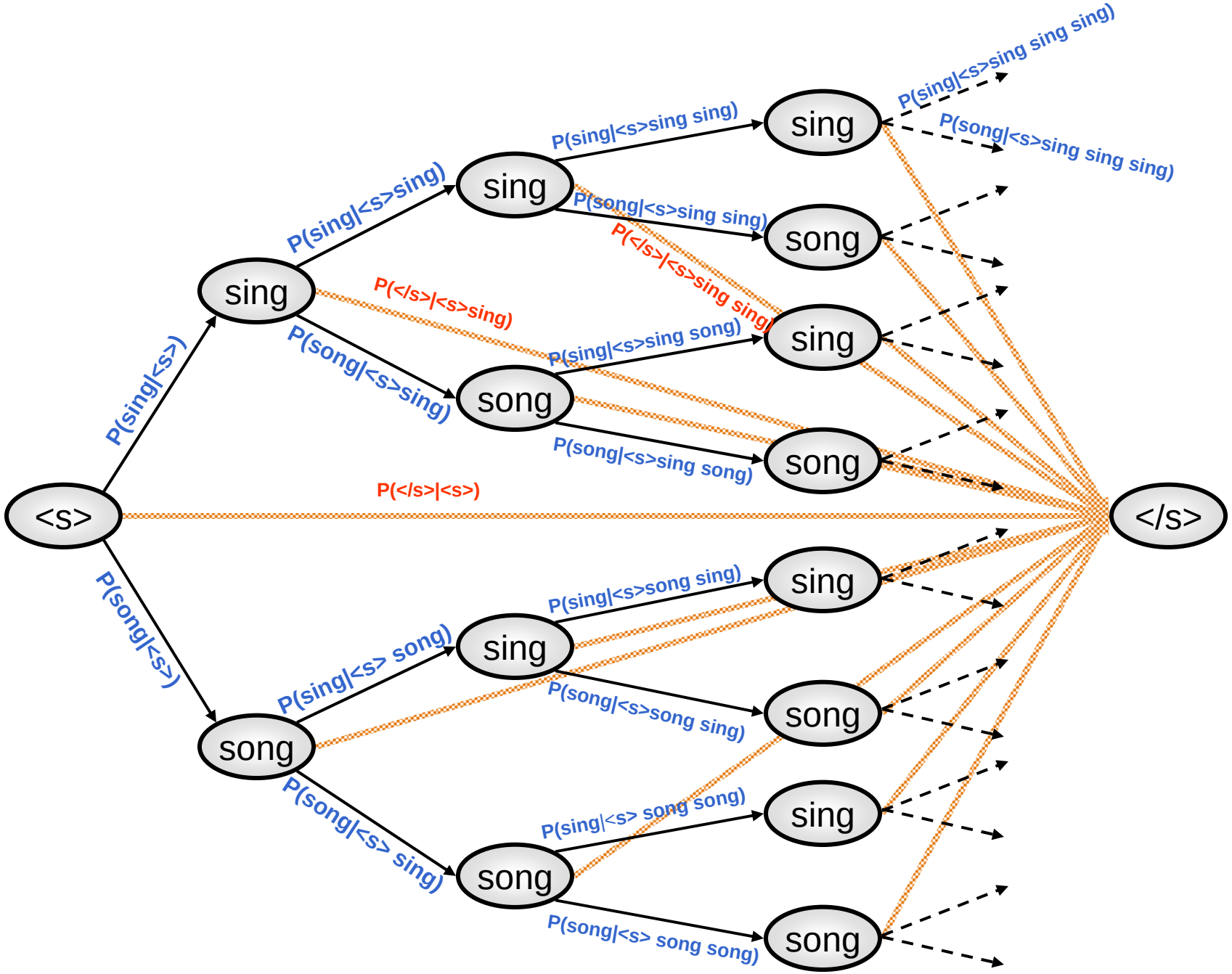
  word1
  word1 **word2**
  word1 word2 **word3**
  word1 word2 word3 **word4**
  .....

The left to right model: A Graphical View



•Assuming a two-word vocabulary: “sing” and “song”

- *A priori* probabilities for word sequences are spread through the graph
  - They are applied on every edge
- This is a much more compact representation of the language than the full graph shown earlier
  - But is still inifinitely large in size


<s>
sing
song
sing
song
sing
song
sing
song
sing
song
sing
song
</s>
P(sing|<s>)
P(song|<s>)
P(sing|<s>sing)
P(song|<s>sing)
P(sing|<s> song)
P(song|<s> sing)
P(sing|<s>sing sing)
P(song|<s>sing sing)
P(sing|<s>sing song)
P(song|<s>sing song)
P(sing|<s>song sing)
P(song|<s>song sing)
P(sing|<s> song song)
P(song|<s> song song)
P(sing|<s>sing sing sing)
P(song|<s>sing sing sing)
P(</s>|<s>)
P(</s>|<s>sing)
P(</s>|<s>sing sing)

Left-to-right language probabilities and the N-gram model

- The N-gram assumption

$$P(w_K \mid w_1, w_2, w_3, \ldots w_{K-1}) = P(w_K \mid w_{K-(N-1)}, w_{K-(N-2)}, \ldots, w_{K-1})$$

- The probability of a word is assumed to be dependent only on the past N-1 words
  - For a 4-gram model, the probability that a person will follow "two times two is" with "four" is assumed to be identical to the probability that they will follow "seven times two is" with "four".

- This is not such a poor assumption
  - Surprisingly, the words we speak (or write) at any time are largely (but not entirely) dependent on the previous 3-4 words.

The validity of the N-gram assumption

- An N-gram language model is a generative model
  - One can generate word sequences randomly from it
- In a good generative model, randomly generated word sequences should be similar to word sequences that occur naturally in the language
  - Word sequences that are more common in the language should be generated more frequently
- Is an N-gram language model a good model?
  - If randomly generated word sequences are plausible in the language, it is a reasonable model
  - If more common word sequences in the language are generated more frequently it is a good model
  - If the relative frequency of generated word sequences is exactly that in the language, it is a perfect model
- Thought exercise: how would you generate word sequences from an N-gram LM ?
  - Clue: Remember that N-gram LMs include the probability of a sentence end marker

Examples of sentences synthesized with N-gram LMs

- 1-gram LM:
  - The and the figure a of interval compared and
  - Involved the a at if states next a a the of producing of too
  - In out the digits right the the to of or parameters endpoint to right
  - Finding likelihood with find a we see values distribution can the a is
- 2-gram LM:
  - Give an indication of figure shows the source and human
  - Process of most papers deal with an HMM based on the next
  - Eight hundred and other data show that in order for simplicity
  - From this paper we observe that is not a technique applies to model
- 3-gram LM:
  - Because in the next experiment shows that a statistical model
  - Models have recently been shown that a small amount
  - Finding an upper bound on the data on the other experiments have been
  - Exact Hessian is not used in the distribution with the sample values

N-gram LMs

- N-gram models are reasonably good models for the language at higher N
  - As N increases, they become better models
- For lower N (N=1, N=2), they are not so good as generative models
- Nevertheless, they are quite effective for analyzing the relative validity of word sequences
  - Which of a given set of word sequences is more likely to be valid
  - They usually assign higher probabilities to plausible word sequences than to implausible ones
- This, and the fact that they are left-to-right (Markov) models makes them very popular in speech recognition
  - They have found to be the most effective language models for large vocabulary speech recognition

Estimating N-gram probabilities

- N-gram probabilities must be estimated from data
- Probabilities can be estimated simply by counting words in training text
- E.g. the training corpus has 1000 words in 50 sentences, of which 400 are "sing" and 600 are "song"
  - count(sing)=400; count(song)=600; count(</s>)=50
  - There are a total of 1050 tokens, including the 50 "end-of-sentence" markers
- UNIGRAM MODEL:
  - P(sing) = 400/1050; P(song) = 600/1050; P(</s>) = 50/1050
- BIGRAM MODEL: finer counting is needed. For example:
  - 30 sentences begin with sing, 20 with song
    - We have 50 counts of <s>
    - P(sing | <s>) = 30/50; P(song|<s>) = 20/50
  - 10 sentences end with sing, 40 with song
    - P(</s> | sing) = 10/400; P(</s>|song) = 40/600
  - 300 instances of sing are followed by sing, 90 are followed by song
    - P(sing | sing) = 300/400; P(song | sing) = 90/400;
  - 500 instances of song are followed by song, 60 by sing
    - P(song | song) = 500/600; P(sing|song) = 60/600

Estimating N-gram probabilities

- Note that "</s>" is considered to be equivalent to a word. The probability for "</s>" are counted exactly like that of other words
- For N-gram probabilities, we count not only words, but also word sequences of length N
  - E.g. we count word sequences of length 2 for bigram LMs, and word sequences of length 3 for trigram LMs
- For N-gram probabilities of order N>1, we also count word sequences that include the word beginning and word end markers
  - E.g. counts of sequences of the kind "<s> w_a w_b" and "w_c w_d </s>"
- The N-gram probability of a word w_d given a context "w_a w_b w_c" is computed as
  - $P(w_d \mid w_a\ w_b\ w_c) = \text{Count}(w_a\ w_b\ w_c\ w_d) / \text{Count}(w_a\ w_b\ w_c)$
  - For unigram probabilities the count in the denominator is simply the count of all word tokens (except the beginning of sentence marker <s>). We do not explicitly compute the probability of P(<s>).

Estimating N-gram probabilities

- Direct estimation by counting is however not possible in all cases
- If we had only a 1000 words in our vocabulary, there are 1001*1001 possible bigrams (including the <s> and </s> markers)
- We are unlikely to encounter all 1002001 word pairs in any given corpus of training data
  - i.e. many of the corresponding bigrams will have 0 count
- However, this does not mean that the bigrams will never occur during recognition
  - E.g., we may never see “sing sing” in the training corpus
  - P(sing | sing) will be estimated as 0
  - If a speaker says “sing sing” as part of any word sequence, at least the “sing sing” portion of it will never be recognized
- The problem gets worse as the order (N) of the N-gram model increases
  - For the 1000 word vocabulary there are more than 10^9 possible trigrams
  - Most of them will never been seen in any training corpus
  - Yet they may actually be spoken during recognition

Discounting

- We must assign a small non-zero probability to all N-grams that were never seen in the training data
- However, this means we will have to reduce the probability of other terms, to compensate
  - Example: We see 100 instances of sing, 90 of which are followed by sing, and 10 by </s> (the sentence end marker).
  - The bigram probabilities computed directly are P(sing|sing) = 90/100, P(<s/>|sing) = 10/100
  - We never observed sing followed by song.
  - Let us attribute a small probability X (X > 0) to P(song|sing)
  - But 90/100 + 10/100 + X > 1.0
  - To compensate we subtract a value Y from P(sing|sing) and some value Z from P(</s>|sing) such that
    - P(sing | sing) = 90 / 100 – Y
    - P(</s> | sing) = 10 / 100 – Z
    - P(sing | sing) + P(</s> | sing) + P(song | sing) = 90/100-Y+10/100-Z+X=1

Discounting and smoothing

- The reduction of the probability estimates for seen Ngrams, in order to assign non-zero probabilities to unseen Ngrams is called discounting
  - The process of modifying probability estimates to be more generalizable is called smoothing
- Discounting and smoothing techniques:
  - Absolute discounting
  - Jelinek-Mercer smoothing
  - Good Turing discounting
  - Other methods
    - Kneser-Ney..
- All discounting techniques follow the same basic principle: they modify the *counts* of Ngrams that are seen in the training data
  - The modification usually reduces the counts of seen Ngrams
  - The withdrawn counts are reallocated to unseen Ngrams
- Probabilities of seen Ngrams are computed from the modified counts
  - The resulting Ngram probabilities are *discounted* probability estimates
  - Non-zero probability estimates are derived for unseen Ngrams, from the counts that are reallocated to unseen Ngrams

Absolute Discounting

- Subtract a constant from all counts
- E.g., we have a vocabulary of K words, w_1, w_2,w_3…w_K
- Unigram:
  - Count of word w_i = C(i)
  - Count of end-of-sentence markers (</s>) = C_{end}
  - Total count $C_{total} = \Sigma_i C(i) + C_{end}$
- Discounted Unigram Counts
  - Cdiscount(i) = C(i) – ε
  - $Cdiscount_{end} = C_{end} – \varepsilon$
- Discounted probability for seen words
  - P(i) = Cdiscount(i) / C_{total}
  - Note that the denominator is the total of the ***undiscounted*** counts
- If K_o words are seen in the training corpus, K – K_o words are unseen
  - A total count of $K_o x \varepsilon$, representing a probability $K_o x \varepsilon / C_{total}$ remains unaccounted for
  - This is distributed among the K – K_o words that were never seen in training
    - We will discuss how this distribution is performed later

Absolute Discounting: Higher order N-grams

- Bigrams: We now have counts of the kind
  - Contexts: Count(w1), Count(w2), … , Count(<s>)
    - Note <s> is also counted; but it is used ***only*** as a context
    - Context does ***not*** incoroporate </s>
  - Word pairs: Count (<s> w_1), Count(<s>,w_2),…,Count(<s> </s>),…, Count(w_1 w_1), …,Count(w_1 </s>) … Count(w_K w_K), Count(w_K </s>)
    - Word pairs ending in </s> are also counted
- Discounted counts:
  - DiscountedCount(w_i w_j) = Count(w_i w_j) – ε
- Discounted probability:
  - $P(w_j \mid w_i)$ = DiscountedCount(w_i w_j) / Count(w_i)
  - Note that the discounted count is used only in the numerator
- For each context w_i, the probability $K_o(w_i)$xε / Count(w_i) is left over
  - $K_o(w_i)$ is the number of words that were seen following w_i in the training corpus
  - $K_o(w_i)$xε / Count(w_i) will be distributed over bigrams $P(w_j \mid w_i)$, for words w_j such that the word pair w_i w_j was never seen in the training data

Absolute Discounting

- Trigrams: Word triplets and word pair contexts are counted
  - Context Counts: Count(<s> w_1), Count(<s> w_2), …
  - Word triplets: Count (<s> $w_1 w_1$),…, Count(w_K w_K, </s>)
- DiscountedCount(w_i w_j w_k) = Count(w_i w_j w_k) – ε
- Trigram probabilities are computed as the ratio of discounted word triplet counts and undiscounted context counts
- The same procedure can be extended to estimate higher-order N-grams
- **The value of ε:** The most common value for ε is 1
  - However, when the training text is small, this can lead to allocation of a disproportionately large fraction of the probability to unseen events
  - In these cases, ε is set to be smaller than 1.0, e.g. 0.5 or 0.1
- The optimal value of ε can also be derived from data
  - Via K-fold cross validation

K-fold cross validation for estimating ε

- Split training data into K equal parts
- Create K different groupings of the K parts by holding out one of the K parts and merging the rest of the K-1 parts together. The held out part is a validation set, and the merged parts form a training set
  - This gives us K different partitions of the training data into training and validation sets
- For several values of ε
  - Compute K different language models with each of the K training sets
  - Compute the total probability Pvalidation(i) of the i^{th} validation set on the LM trained from the i^{th} training set
  - Compute the total probability $Pvalidation_{\varepsilon}$ = Pvalidation(1)*Pvalidation(2)*..*Pvalidation(K)
- Select the ε for which $Pvalidation_{\varepsilon}$ is maximum
- Retrain the LM using the *entire* training data, using the chosen value of ε

The Jelinek Mercer Smoothing Technique

- Jelinek-Mercer smoothing returns the probability of an N-gram as a weighted combination of maximum likelihood N-gram and smoothed N-1 gram probabilities

$$P_{smooth}(word \mid wa\ wb\ wc...) = \lambda(wa\ wb\ wc...)P_{ML}(word \mid wa\ wb\ wc...) + (1.0 - \lambda(wa\ wb\ wc...))P_{smooth}(word \mid wb\ wc...)$$

- P_{smooth}(word | wa wb wc..) is the N-gram probability used during recognition
  - The higher order (N-gram) term on the right hand side, P_{ML}(word | wa wb wc..) is simply a maximum likelihood (counting-based) estimate of P(word | wa wb wc..)
  - The lower order ((N-1)-gram term) P_{smooth}(word | wb wc..) is recursively obtained by interpolation between the ML estimate P_{ML}(word | wb wc..) and the smoothed estimate for the (N-2)-gram P_{smooth}(word | wc..)
  - All λ values lie between 0 and 1
  - Unigram probabilities are interpolated with a uniform probability distribution

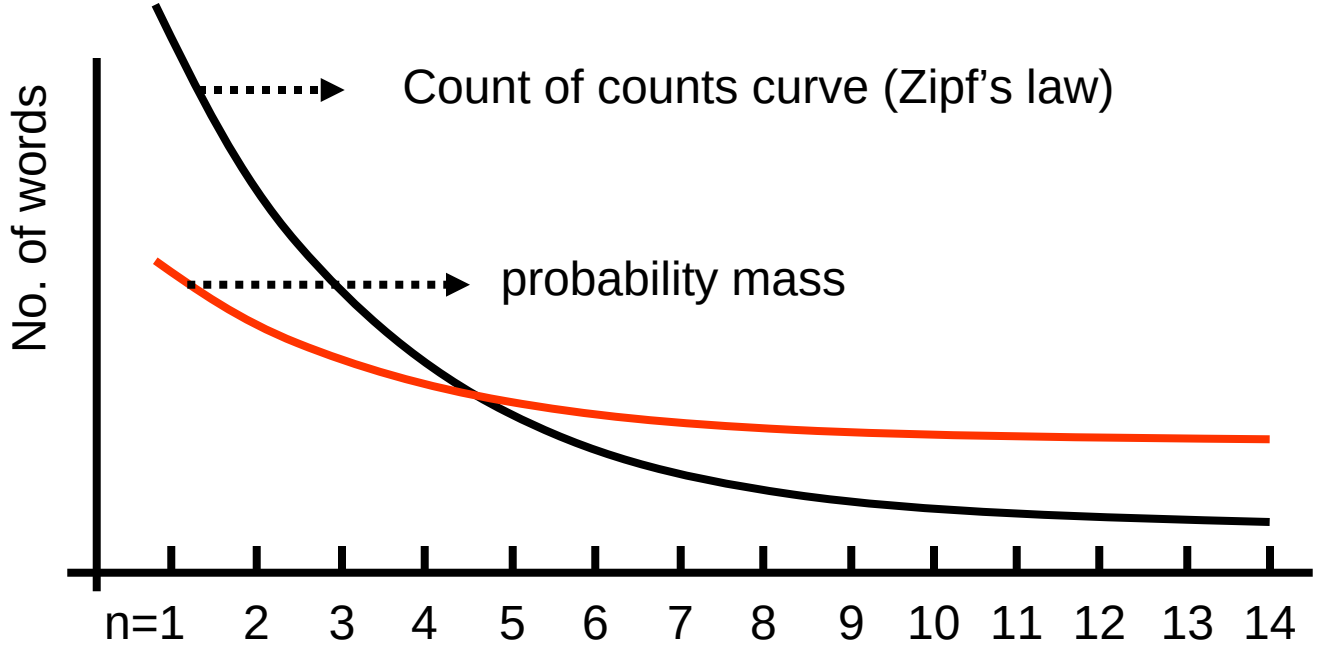
- The λ values must be estimated using held-out data
  - A combination of K-fold cross validation and the expectation maximization algorithms must be used
  - We will not present the details of the learning algorithm in this talk
  - Often, an arbitrarily chosen value of λ, such as λ = 0.5 is also very effective

Good-Turing discounting: Zipf's law

- Zipf's law: The number of events that occur often is small, but the number of events that occur very rarely is very large.
- If *n* represents the number of times an event occurs in a unit interval, the number of events that occur *n* times per unit time is proportional to $1/n^{\alpha}$, where α is greater than 1
  - George Kingsley Zipf originally postulated that $\alpha = 1$.
  - Later studies have shown that α is $1 + \varepsilon$, where ε is slightly greater than 0
- Zipf's law is true for words in a language: the probability of occurrence of words starts high and tapers off. A few words occur very often while many others occur rarely.

Good-Turing discounting

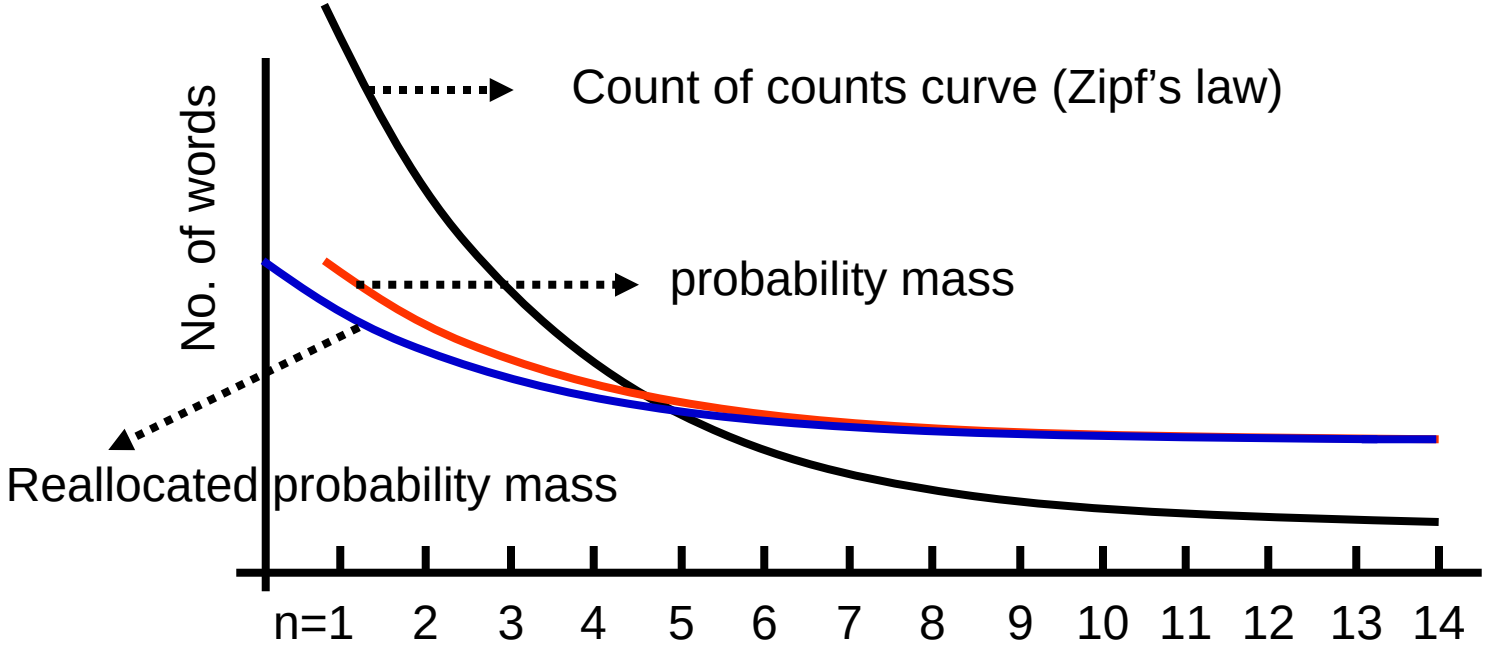
- A plot of the count of counts of words in a training corpus typically looks like this:



- In keeping with Zipf's law, the number of words that occur n times in the training corpus is typically more than the number of words that occur n+1 times
  - The total probability mass of words that occur n times falls slowly
  - Surprisingly, the total probability mass of rare words is greater than the total probability mass of common words, because of the large number of rare words

Good-Turing discounting

- A plot of the count of counts of words in a training corpus typically looks like this:



- Good Turing discounting reallocates probabilities
  - The total probability mass of all words that occurred n times is assigned to words that occurred n-1 times
  - The total probability mass of words that occurred once is reallocated to words that were never observed in training

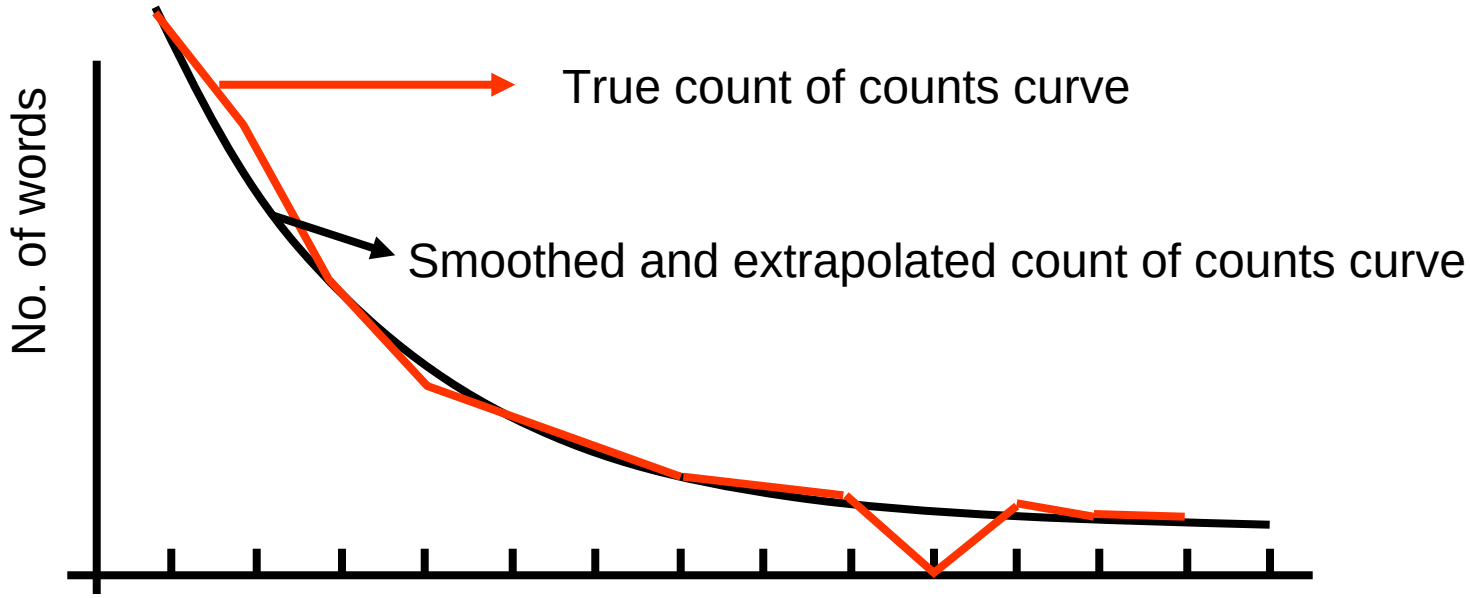
Good-Turing discounting

- The probability mass curve cannot simply be shifted left directly due to two potential problems

- Directly shifting the probability mass curve assigns 0 probability to the most frequently occurring words
  - Let the words that occurred most frequently have occurred M times
  - When probability mass is reassigned, the total probability of words that occurred M times is reassigned to words that occurred M-1 times
  - Words that occurred M times are reassigned the probability mass of words that occurred M+1 times = 0.
  - i.e. the words that repeated most often in the training data (M times) are assigned 0 probability!

- The count of counts curve is often not continuous
  - We may have words that occurred L times, and words that occurred L+2 times, but none that occurred L+1 times
  - By simply reassigning probability masses backward, words that occurred L times are assigned the total probability of words that occurred L+1 times = 0!

Good-Turing discounting



- The count of counts curve is smoothed and extrapolated
  - Smoothing fills in "holes" – intermediate counts for which the curve went to 0
  - Smoothing may also vary the counts of events that were observed
  - Extrapolation extends the curve to one step beyond the maximum count observed in the data
- Smoothing and extrapolation can be done by linear interpolation and extrapolation, or by fitting polynomials or splines
- Probability masses are computed from the smoothed count-of-counts and reassigned

Good-Turing discounting

- Let r'(i) be the smoothed count of the number of words that occurred i times.
- The total smoothed count of all words that occurred i times is r'(i) * i.

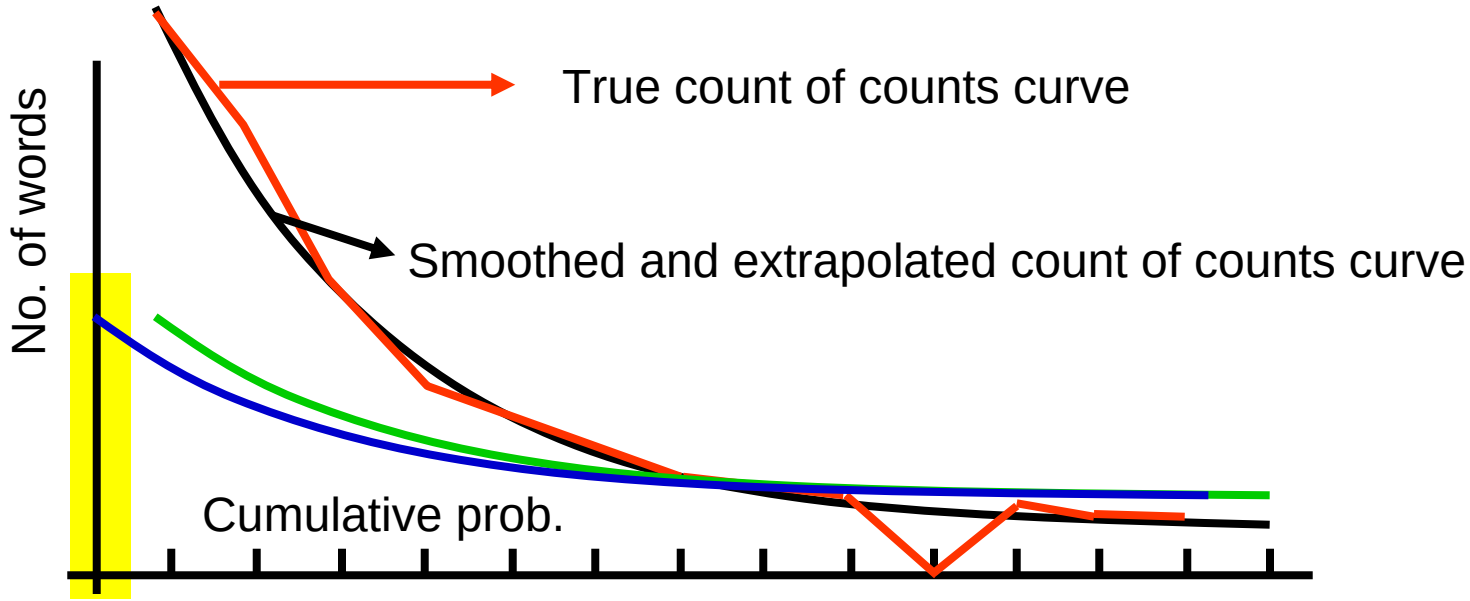
- When we reassign probabilities, we assign the total counts r'(i)*i to words that occurred i-1 times. There are r'(i-1) such words (using smoothed counts). So effectively, every word that occurred i-1 times is reassigned a count of
  - reassignedcount(i-1) = r'(i)*i / r'(i-1)

- The total reassigned count of all words in the training data is totalreassignedcount = Σ_i r'(i+1)*(i+1)
  where the summation goes over all i such that there is at least one word that occurs i times in the training data (this includes i = 0)

- A word *w* with count *i* is assigned probability
  P(*w*| *context*) = reassignedcount(i) / totalreassignedcount

- A probability mass r'(1) / totalreassignedcount is left over
  - The left-over probability mass is reassigned to words that were not seen in the training corpus

Good-Turing discounting



- Discounting effectively "moves" the green line backwards
  - I.e. cumulative probabilities that should have been assigned to count N are assigned to count N-1
  - This now assigns "counts" to events that were never seen
  - We can now compute probabilities for these terms

Good-Turing estimation of LM probabilities

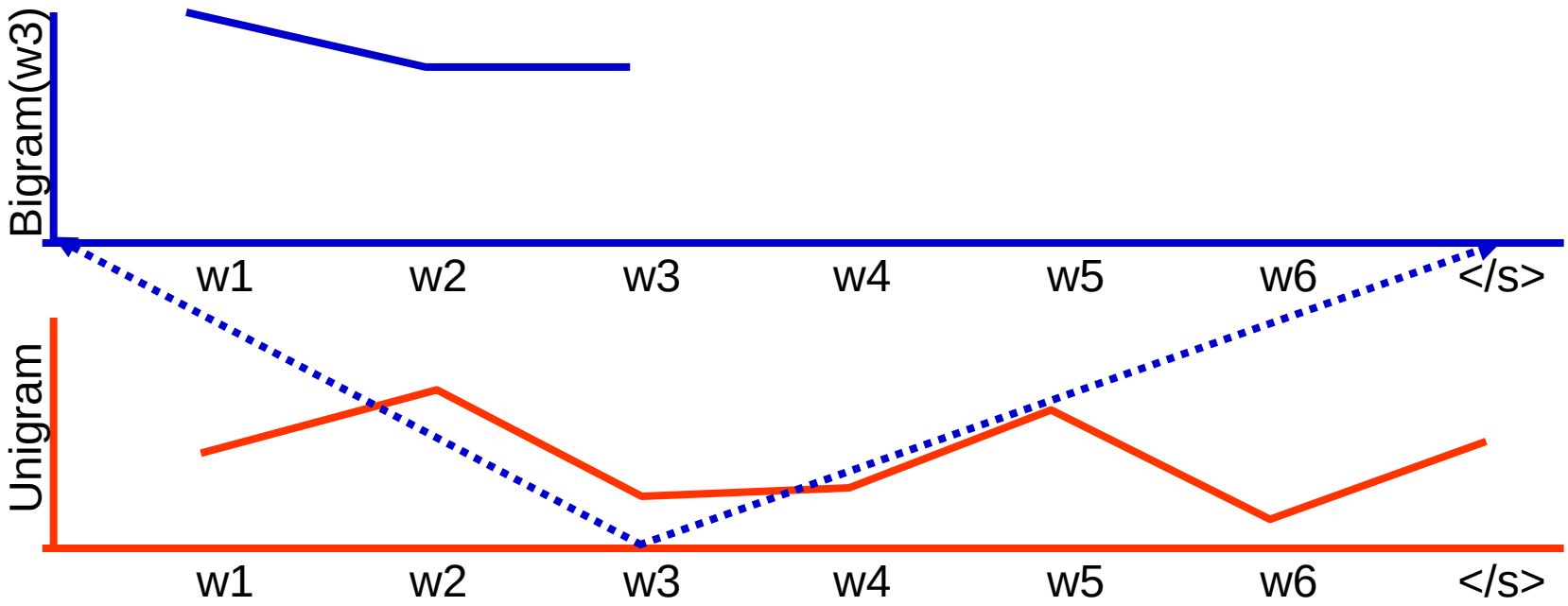
- UNIGRAMS:
  - The count-of-counts curve is derived by counting the words (including </s>) in the training corpus
  - The count-of-counts curve is smoothed and extrapolated
  - Word probabilities are computed for observed words are computed from the smoothed, reassigned counts
  - The left-over probability is reassigned to unseen words
- BIGRAMS:
  - For each word context W, (where W can also be <s>), the same procedure given above is followed: the count-of-counts for all words that occur immediately after W is obtained, smoothed and extrapolated, and bigram probabilities for words seen after W are computed.
  - The left-over probability is reassigned to the bigram probabilities of words that were never seen following W in the training corpus
- Higher order N-grams: The same procedure is followed for every word context W_1 W_2… W_{N-1}

Reassigning left-over probability to unseen words

- All discounting techniques result in a some left-over probability to reassign to unseen words and N-grams
- For unigrams, this probability is uniformly distributed over all unseen words
  - The vocabulary for the LM must be prespecified
  - The probability will be reassigned uniformly to words from this vocabulary that were not seen in the training corpus
- For higher-order N-grams, the reassignment is done differently
  - Based on lower-order N-gram, i.e. (N-1)-gram probabilities
  - The process by which probabilities for unseen N-grams is computed from (N-1)-gram probabilities is referred to as “backoff”

N-gram LM: Backoff

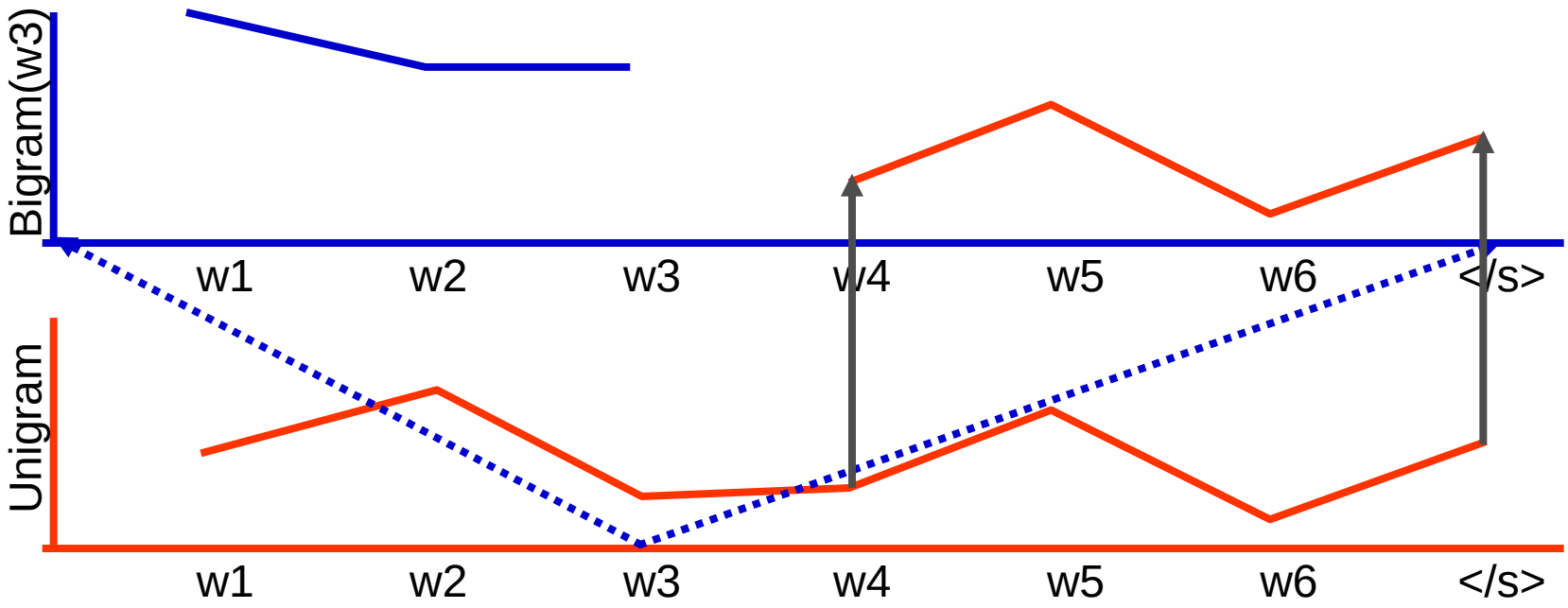
- Explanation with a bigram example



- Unigram probabilities are computed and known before bigram probabilities are computed
- Bigrams for P(w1 | w3), P(w2 | w3) and P(w3 | w3) were computed from discounted counts. w4, w5, w6 and </s> were never seen after w3 in the training corpus

N-gram LM: Backoff

- Explanation with a bigram example



- The probabilities P(w4|w3), P(w5|w3), P(w6|w3) and P(</s>|w3) are assumed to follow the same pattern as the unigram probabilities P(w4), P(w5), P(w6) and P(</s>)
- They must, however be scaled such that
  P(w1|w3) + P(w2|w3) + P(w3|w3) + scale*(P(w4)+P(w5)+P(w6)+P(</s>)) = 1.0
- The *backoff* bigram probability for the unseen bigram P(w4 | w3) = scale*P(w4)

N-gram LM (Katz Models): Backoff from N-gram to (N-1)-gram

- Assumption: When estimating N-gram probabilities, we already have access to all N-1 gram probabilities
- Let $w_1 \ldots w_K$ be the words in the vocabulary (includes </s>)
- Let "wa wb wc…" be the context for which we are trying to estimate N-gram probabilities
  - i.e we wish to compute all probabilities P(word | wa wb wc ..)
- Let $w_1 \ldots w_L$ be the words that were seen in the context "wa wb wc.." in the training data. We compute the N-gram probabilities for these words after discounting. We are left over with an unaccounted for probability mass

$$P_{leftover}(wa\ wb\ wc...) = 1.0 - \sum_{i=1}^{L} P(w_i \mid wa\ wb\ wc...)$$

- We must assign the left over probability mass $P_{leftover}$(wa wb wc ...) to the words $w_{L+1}, w_{L+2}, \ldots w_K$, in the context "wa wb wc ..."
  - *i.e.* we want to assign them to P(w_{L+1} | wa wb wc ..), P(w_{L+2} | wa wb wc …), etc.

N-gram LM: Learning the Backoff scaling term

- The backoff assumption for *unseen* N-grams:
  - $P(w_i \mid wa\ wb\ wc\ ..) = \beta(wa\ wb\ wc\ \ldots) * P(w_i \mid wb\ wc\ \ldots)$
    - The scaling constant $\beta(wa\ wb\ wc\ \ldots)$ is specific to the context of the Ngram
  - i.e. the N-gram probability is proportional to the N-1 gram probability
  - In the backoff LM estimation procedure, N-1 gram probabilities are assumed to be already known, when we estimate Ngram probabilities, so $P(w_i \mid wb\ wc\ \ldots)$ is available for all w_i

- $\beta(wa\ wb\ wc\ \ldots)$ must be set such that

$$\beta(wa\ wb\ wc...) \sum_{i \in \{unseen\}} P(w_i \mid wb\ wc...) = P_{leftover}(wa\ wb\ wc...)$$

$$\beta(wa\ wb\ wc...) = \frac{P_{leftover}(wa\ wb\ wc...)}{\sum_{i \in \{unseen\}} P(w_i \mid wb\ wc...)}$$

- Note that $\beta(wa\ wb\ wc\ \ldots)$ is *specific* to the context "wa wb wc …"
  - $\beta(wa\ wb\ wc\ \ldots)$ is known as the backoff weight of the context "wa wb wc…"
- Once $\beta(wa\ wb\ wc\ \ldots)$ has been computed, we can derive Ngram probabilities for unseen Ngram from the corresponding N-1 grams

Backoff is recursive

- In order to estimate the backoff weight needed to compute N-gram probabilities for unseen N-grams, the corresponding N-1 grams are required
  - The corresponding N-1 grams might also not have been seen in the training data
- If the backoff N-1 grams are also unseen, they must in turn be computed by backing off to N-2 grams
  - The backoff weight for the unseen N-1 gram must also be known
  - i.e. it must also have been computed already
- The procedure is recursive – unseen N-2 grams are computed by backing off to N-3 grams, and so on
- All lower order N-gram parameters (including probabilities and backoff weights) must be computed before higher-order N-gram parameters can be estimated

Learning Backoff Ngram models

- First compute Unigrams
  - Count words, perform discounting, estimate discounted probabilities for all seen words
  - Uniformly distribute the left-over probability over unseen unigrams
- Next, compute bigrams. For each word W seen in the training data:
  - Count words that follow that W. Estimate discounted probabilities P(word | W) for all words that were seen after W.
  - Compute the backoff weight β(W) for the context W.
  - The set of explicity estimated P(word | W) terms, and the backoff weight β(W) together permit us to compute all bigram probabilities of the kind: P(word | W)
- Next, compute trigrams: For each word pair "wa wb" seen in the training data:
  - Count words that follow that "wa wb". Estimate discounted probabilities P(word | wa wb) for all words that were seen after "wa wb".
  - Compute the backoff weight β(wa wb) for the context "wa wb".
- The process can be continued to compute higher order N-gram probabilities.

The contents of a completely trained N-gram language model

- An N-gram backoff language model contains
  - Unigram probabilities for all words in the vocabulary
  - Backoff weights for all words in the vocabulary
  - Bigram probabilities for some, but not all bigrams
    - i.e. for all bigrams that were seen in the training data
  - If N>2, then: backoff weights for all seen word pairs
    - If the word pair was never seen in the training corpus, it will not have a backoff weight. The backoff weight for all word pairs that were not seen in the training corpus is implicitly set to 1
  - …
  - N-gram probabilities for some, but not all N-grams
    - N-grams seen in training data
  - Note that backoff weights are not required for N-length word sequences in an N-gram LM
    - Since backoff weights for N-length word sequences are only useful to compute backed off N+1 gram probabilities

An Example Backoff Trigram LM

\1-grams:
-1.2041 <UNK> 0.0000
-1.2041 </s> 0.0000
-1.2041 <s> -0.2730
-0.4260 one -0.5283
-1.2041 three -0.2730
-0.4260 two -0.5283
\2-grams:
-0.1761 <s> one 0.0000
-0.4771 one three 0.1761
-0.3010 one two 0.3010
-0.1761 three two 0.0000
-0.3010 two one 0.3010
-0.4771 two three 0.1761
\3-grams:
-0.3010 <s> one two
-0.3010 one three two
-0.4771 one two one
-0.4771 one two three
-0.3010 three two one
-0.4771 two one three
-0.4771 two one two
-0.3010 two three two

Obtaining an N-gram probability from a backoff N-gram LM

- To retrieve a probability P(word | wa wb wc …)
  - How would a function written for returning N-gram probabilities work?
- Look for the probability P(word | wa wb wc …) in the LM
  - If it is explicitly stored, return it
- If P(word | wa wb wc …) is not explicitly stored in the LM retrive it by backoff to lower order probabilities:
  - Retrieve backoff weight β(wa wb wc..) for word sequence wa wb wc …
    - If it is stored in the LM, return it
    - Otherwise return 1
  - Retrieve P(word | wb wc …) from the LM
    - If P(word | wb wc ..) is not explicitly stored in the LM, derive it backing off
    - This will be a recursive procedure
  - Return P(word | wb wc …) * β(wa wb wc..)

Training a language model using CMU-Cambridge LM toolkit

http://mi.eng.cam.ac.uk/~prc14/toolkit.html

http://www.speech.cs.cmu.edu/SLM_info.html

Contents of textfile

<s> the term cepstrum was introduced by Bogert et al and has come to be accepted terminology for the
inverse Fourier transform of the logarithm of the power spectrum
of a signal in nineteen sixty three Bogert Healy and Tukey published a paper with the unusual title
The Quefrency Analysis of Time Series for Echoes Cepstrum Pseudoautocovariance Cross Cepstrum and Saphe Cracking
they observed that the logarithm of the power spectrum of a signal containing an echo has an additive
periodic component due to the echo and thus the Fourier transform of the logarithm of the power
spectrum should exhibit a peak at the echo delay
they called this function the cepstrum
interchanging letters in the word spectrum because
in general, we find ourselves operating on the frequency side in ways customary on the time side and vice versa
Bogert et al went on to define an extensive vocabulary to describe this new signal processing technique however only the term cepstrum has been widely used
The transformation of a signal into its cepstrum is a homomorphic transformation and the concept of the cepstrum is a fundamental part of the theory of homomorphic systems for processing signals that have been combined by convolution
</s>

Contents of contextfile

<s>

vocabulary

<s>
</s>
the
term
cepstrum
was
introduced
by
Bogert
et
al
and
has
come
to
be
accepted
terminology
for
inverse
Fourier
transform
of
logarithm
Power
. . .

Training a language model using CMU-Cambridge LM toolkit

To train a bigram LM (n=2):

$bin/text2idngram -vocab vocabulary -n 2 -write_ascii < textfile > idngm.tempfile

$bin/idngram2lm -idngram idngm.tempfile -vocab vocabulary -arpa MYarpaLM -context contextfile -absolute -ascii_input -n 2 (optional: -cutoffs 0 0 or –cutoffs 1 1 ….)

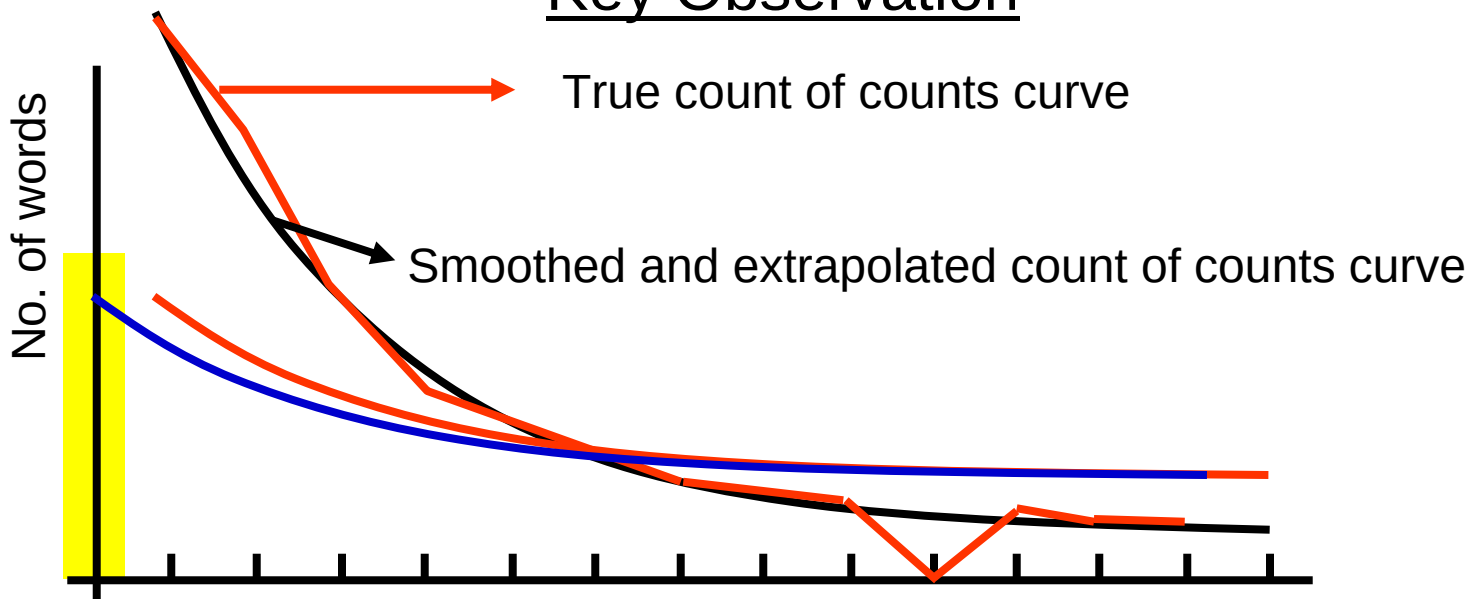
OR

$bin/idngram2lm -idngram idngm.tempfile -vocab vocabulary -arpa MYarpaLM -context contextfile -good_turing -ascii_input -n 2

….

SRILM uses a single command called “ngram” (I believe)

Key Observation



- The vocabulary of the LM is specified at training time
  - Either as an external list of words or as the set of all words in the training data
- The number of words in this vocabulary is used to compute the probability of zero-count terms
  - Divide the total probability mass in the yellow region by the total number of words that were not seen in the training data
- Words that are not explicitly listed in the vocabulary will not be assigned any probability
  - Effectively have zero probability

Changing the Format

- The SRILM format must be changed to match the sphinx format.

The UNK word

- The vocabulary to be recognized must be specified to the language modelling toolkit
- The training data may contain many words that are not part of this vocabulary
- These words are “unknown” as far as the recognizer is concerned
- To indicate this, they are usually just mapped onto “UNK” by the toolkit
- Leads to the introduction of probabilities such as P(WORD | UNK) and P(UNK | WORD) in the language model
  - These are never used for recognition

<s> and </s>

- The probability that a word can begin a sentence also varies with the word
  - Few sentences begin with “ELEPHANT”, but many begin with “THE”
  - It is important to capture this distinction
- The <s> symbol is a “start of sentence” symbol.
- It is appended to the start of every sentence in the training data
  - E.g. “It was a sunny day” → “<s> It was a sunny day”
- This enables computation of probabilities such as P(it | <s>)
  - The probability that a sentence begins with “it”.
  - Higher order N-gram probabilities can be computed: P(was | <s> it)
    - Probability that the second word in a sentence will be “was” given that the first word was “it”

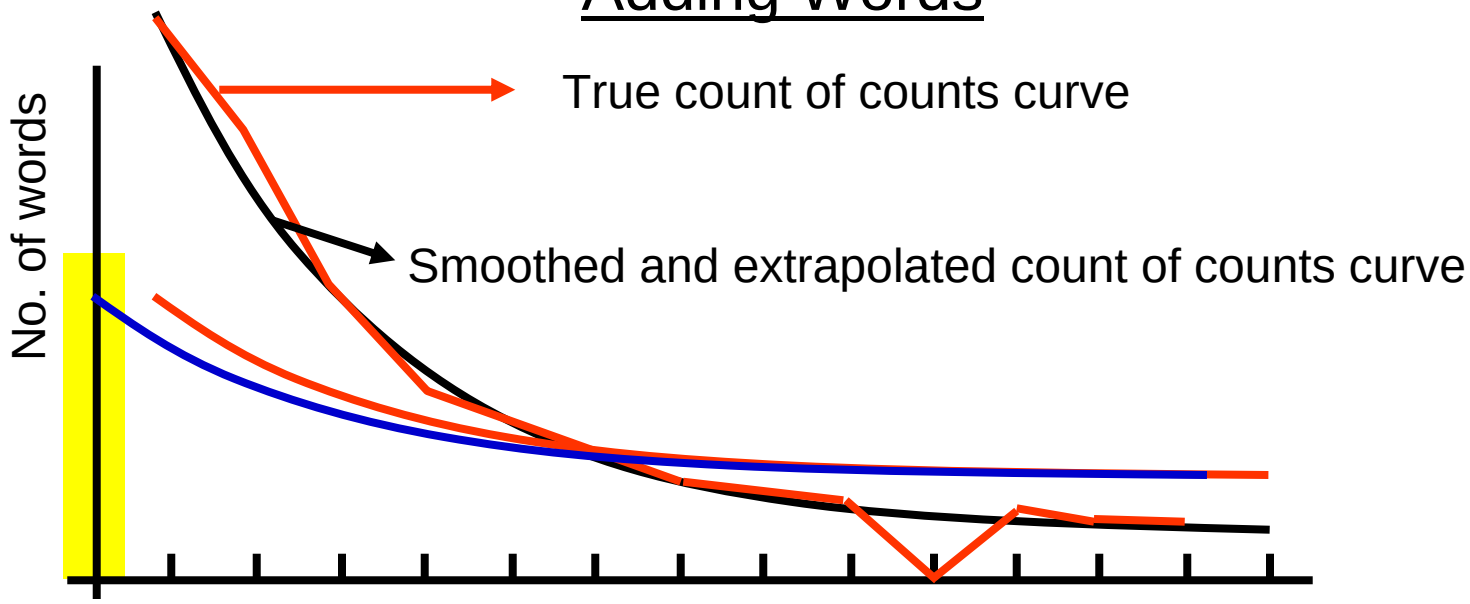
<s> and </s>

- Ends of sentences are similarly distinctive
  - Many sentences end with “It”. Few end with “An”.
- The </s> symbol is an “end of sentence” symbol.
  - It is appended to the *end* of every sentence in the training data
  - E.g. “It was a sunny day” → “<s> It was a sunny day </s>”
- This enables computation of probabilities such as P(| it)
  - The probability that a sentence ends with “it”.
  - Higher order N-gram probabilities can be computed: P(</s> | good day)
    - Probability that the sentence ended with the word pair “good day”.

<s> and </s>

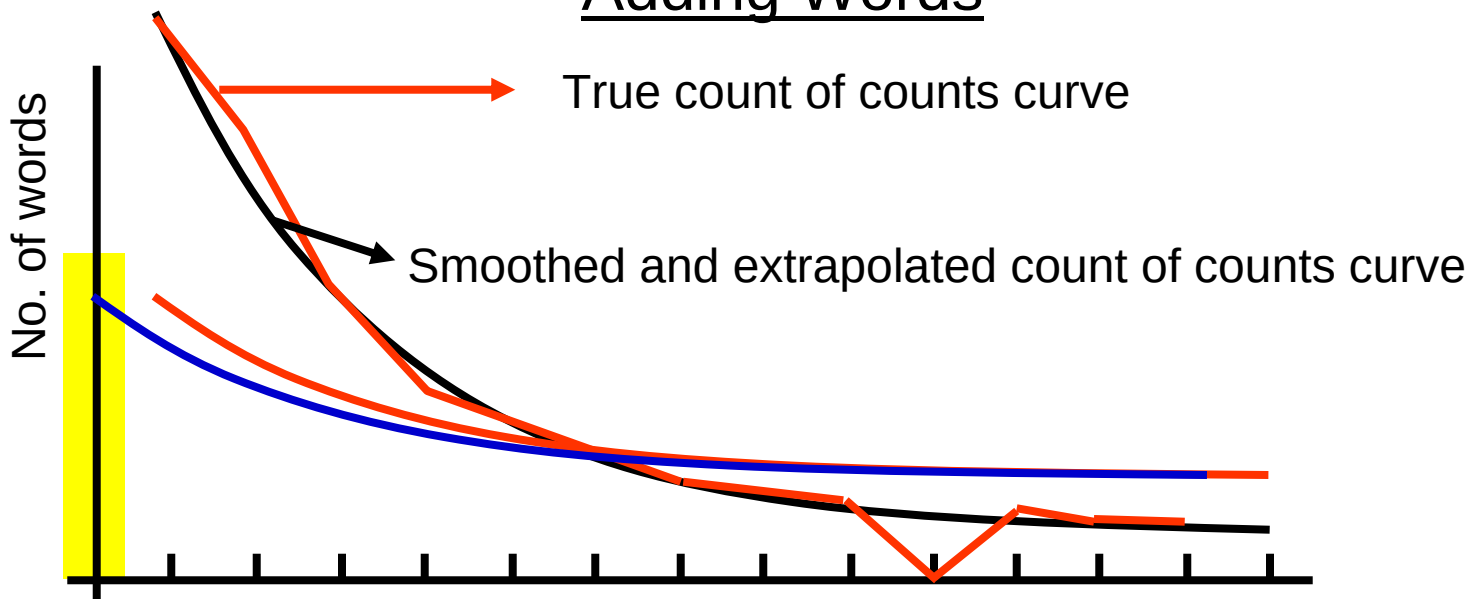
- Training probabilities for <s> and </s> may give rise to spurious probability entries
  - Adding <s> and </s> to “It was a dark knight. It was a stormy night” makes it “<s> It was a dark knight </s> <s> It was a stormy night </s>”
  - Training probabilities from this results in the computation of probability terms such as P(<s> | </s>)
    - The probabilitiy that a sentence will begin after a sentence ended
  - And other terms such as P(<s> | knight </s>)
    - The probability that a sentence will begin when the previous sentence ended with night
- It is often advisable to avoid computing such terms
  - Which may be meaningless
- Hard to enforce however
  - The SRI LM toolkit deals with it corrrectly if every sentence is put on a separate line
  - <s> It was a dark knight </s>
  - <s> It was a story knight </s>
  - There are no words before <s> and after </s> in this format

Adding Words



- Adding words to an existing LM can be difficult
  - The vocabulary of the LM is already specified when it is trained
  - Words that are not in this list will have zero probability
    - Simply extending the size of a dictionary won't automatically introduce the word into the LM

Adding Words



- New words that are being added have not been seen in training data
  - Or will be treated as such anyway
  - They are zerotons!
- In order to properly add a word to the LM and assign it a probability, the probability of *other* zeroton words in the LM must reduced
  - So that all probabilities sum to 1.0
- Reassign the probability mass in the yellow region of the plot to actually account for the new word too

Adding Words

- Procedure to adjust Unigram probabilities
  - First identify all words in the LM that represent zeroton words in the training data
    - This information is not explicitly stored
    - Let there be N such words
    - Let P be the backed-off UG probability
  - Modify the unigram probabilities of all zeroton words to P*N/(N+1)
    - Basically reduce their probabilities so that after they are all summed up, a little is left out
  - Assign the probability P*N/(N+1) to the new word being introduced
- Mercifully, Bigram and Trigram probabilities do not have to be adjusted
  - The new word was never seen in any context

Identifying Zeroton Words

- The first step is to identify the current zeroton words
- Characteristics
  - Zeroton words have *only* unigram probabilities
  - Any word that occurs in training data also produces bigrams
    - E.g. if “HELLO” is seen in the training data, it *must* have been followed either by a word or by an </s>
    - We will at least have a bigram P(</s> | word)
- For every word
  - Look for the existence of at least *one* bigram with the word as context
  - If such a bigram does not exist, treat it as a zeroton

Adding Ngrams

- Ngrams cannot be arbitrarily introduced into the language model
  - A zeroton Ngram is also a zeroton unigram
  - So its probabilities will be obtained by backing off to unigrams

Domain specificity in LMs

- An N-gram LM will represent linguistic patterns for the specific domain from which the training text is derived
  - E.g. training on "broadcast news" corpus of text will train an LM that represents news broadcasts
- For good recognition it is important to have an LM that represents the data
- Often we find ourselves in a situation where we do not have an LM for the exact domain, but one or more LMs from close domains
  - E.g. We have a large LM trained from lots of newspaper text that represents typical news data, but its very grammatical
  - We have a smaller LM trained from a small amount of *broadcast* news text
    - But the training data are small and the LM is not well estimated
- We would like to *combine* them somehow to get a good LM for our domain

Interpolating LMs

- The probability that word2 will follow word1, as specified by LM1 is P(word2 | word1, LM1)
  - LM1 is well estimated but too generic
- The probability that word2 will follow word1, as specified by LM2 is P(word2 | word1, LM2)
  - LM2 is related to our domain but poorly estimated
- Compute P(word2 | word1) for the domain by *interpolating* the values from LM1 and LM2

P(word2 | word1) = α P(word2 | word1, LM1) + (1- α)P(word2 | word1,LM2)

- The value α must be tuned to represent the domain of our test data
  - Can be done using automated methods
  - More commonly, just hand tuned (or set to 0.5)

Interpolating LMs

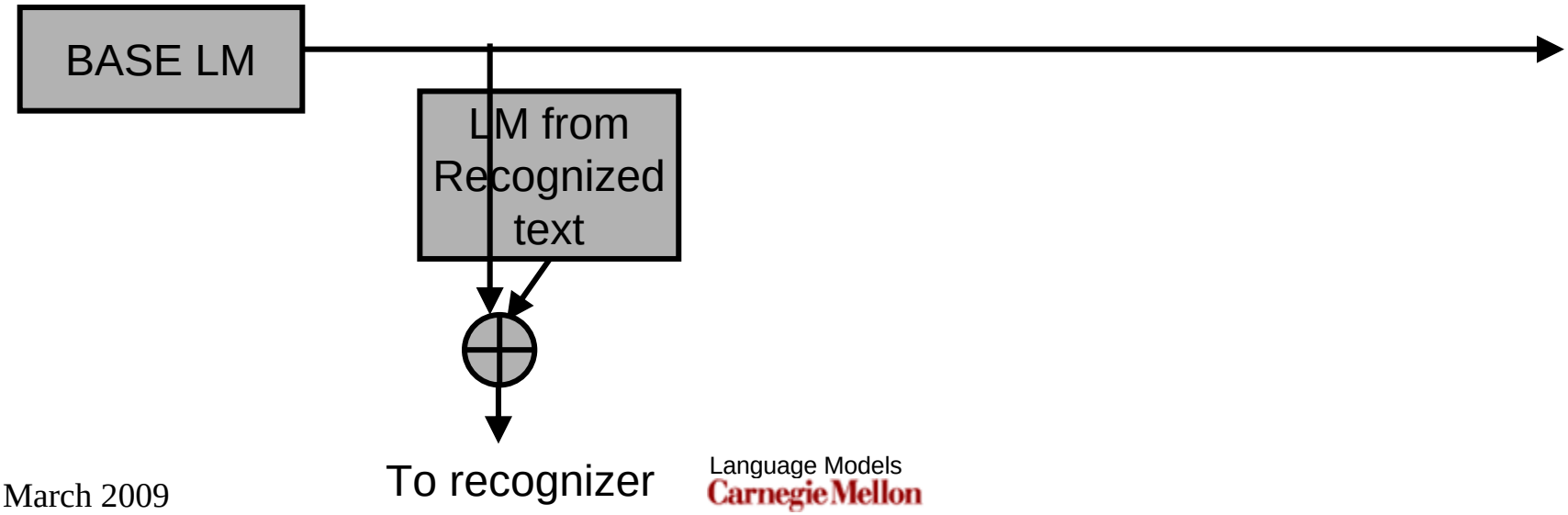
- However, the LM probabilities cannot just be interpolated directly
  - The vocabularies for the two LMs may be different
  - So the probabilities for some words found in one LM may not be computable from the other
- *Normalize* the vocabularies of the two LMs
  - Add all words in LM1 that are not in LM2 to LM2
  - Add all words in LM2 that are not in LM1 to LM1
- Interpolation of probabilities is performed with the normalized-vocabulary LMs
- This means that the recognizer actually loads up two LMs during recognition
  - Alternately, all interpolated bigram and trigram probabilities may be computed offline and written out

Adapting LMs

- The topic that is being spoken about may change continuously as a person speaks

- This can be done by “adapting” the LM
  - After recognition, train an LM from the recognized word sequences from the past few minutes of speech
  - Interpolate this LM with the larger “base” LM

  - This is done continuously

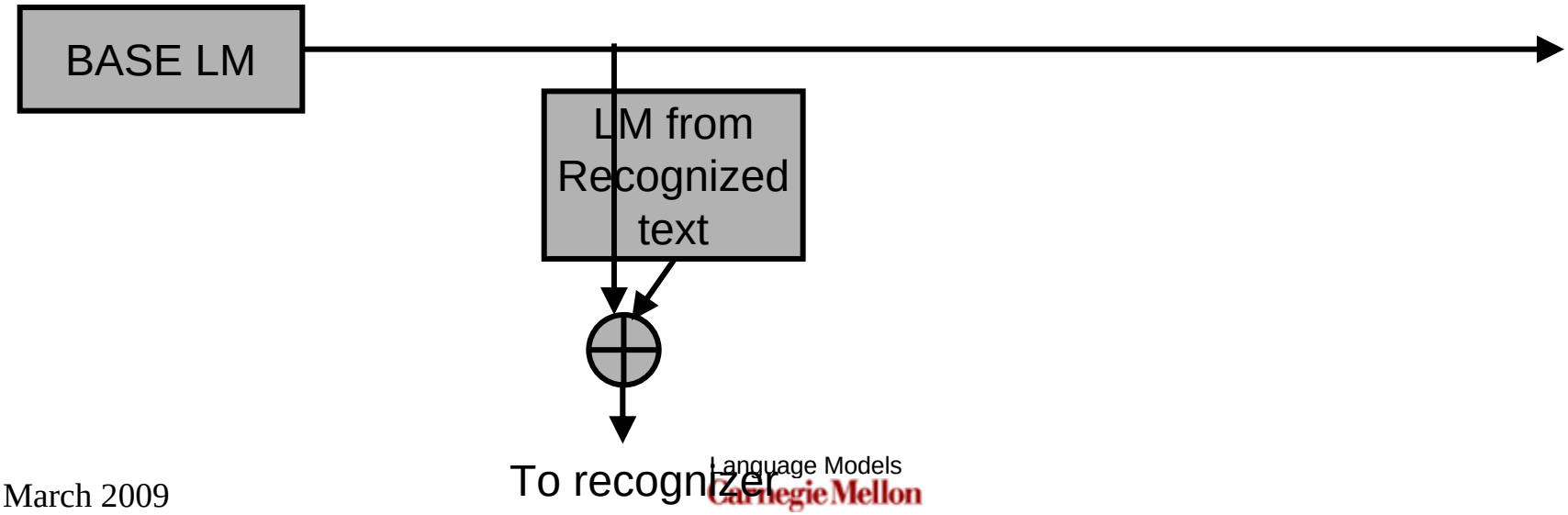


Adapting LMs

- The topic that is being spoken about may change continuously as a person speaks

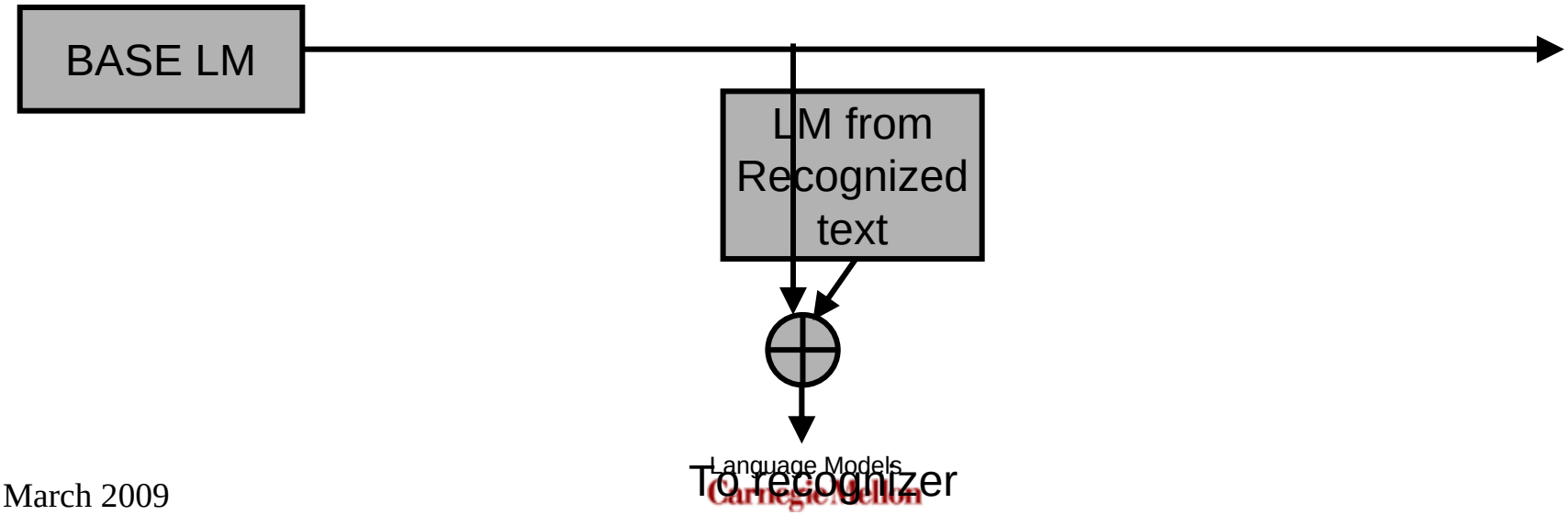
- This can be done by “adapting” the LM
  - After recognition, train an LM from the recognized word sequences from the past few minutes of speech
  - Interpolate this LM with the larger “base” LM

  - This is done continuously



Adapting LMs

- The topic that is being spoken about may change continuously as a person speaks
- This can be done by “adapting” the LM
  - After recognition, train an LM from the recognized word sequences from the past few minutes of speech
  - Interpolate this LM with the larger “base” LM
  - This is done continuously

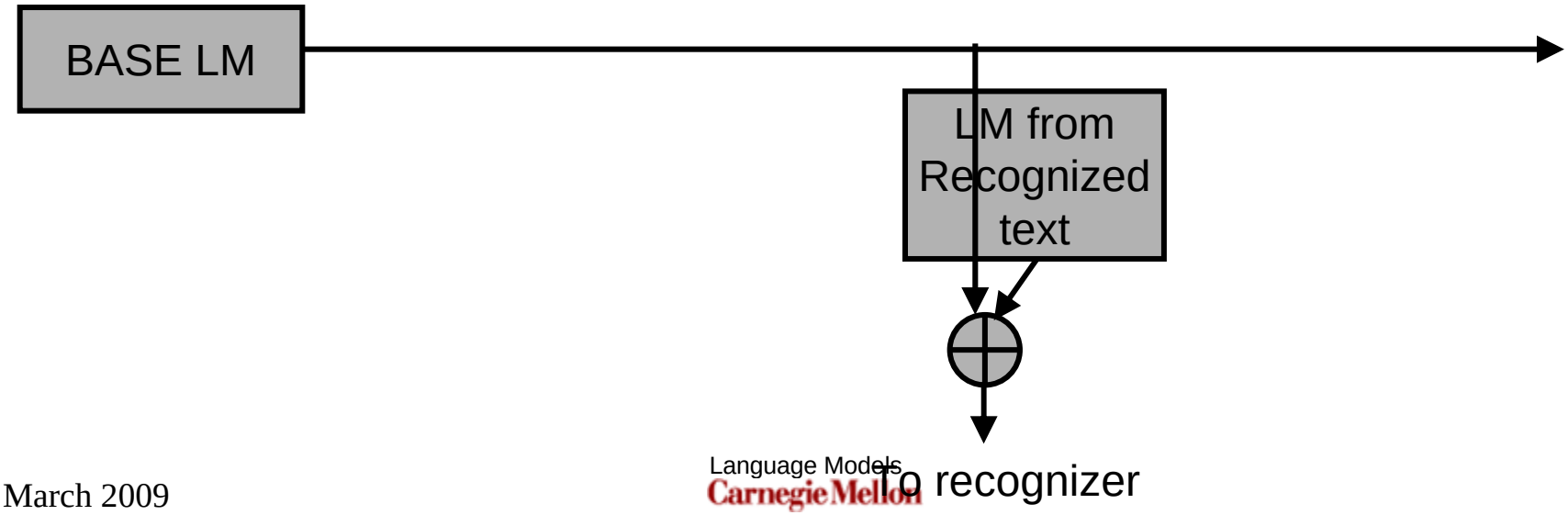


Adapting LMs

- The topic that is being spoken about may change continuously as a person speaks

- This can be done by “adapting” the LM
  - After recognition, train an LM from the recognized word sequences from the past few minutes of speech
  - Interpolate this LM with the larger “base” LM

  - This is done continuously

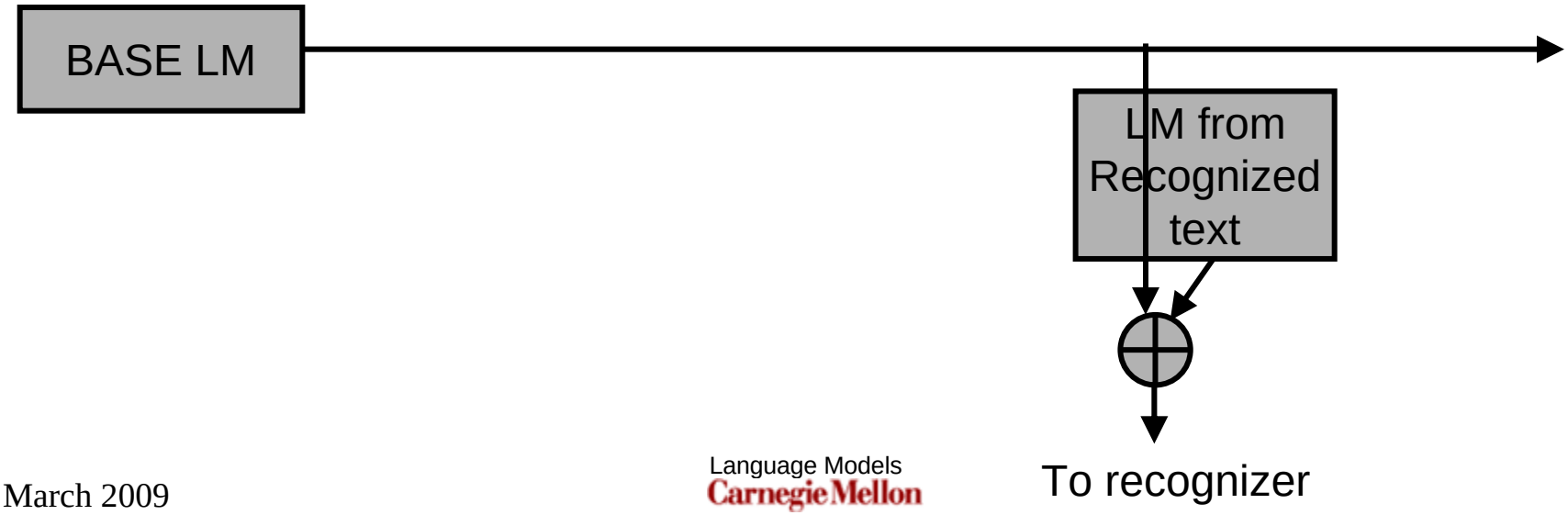


Adapting LMs

- The topic that is being spoken about may change continuously as a person speaks

- This can be done by “adapting” the LM
  - After recognition, train an LM from the recognized word sequences from the past few minutes of speech
  - Interpolate this LM with the larger “base” LM

  - This is done continuously



Identifying a domain

- If we have LMs from different domains, we can use them to recognize the domain of the current speech
- Recognize the data using each of the LMs
- Select the domain whose LM results in the recognition with the highest probability

More Features of the SRI LM toolkit

- Functions

Overall procedure for recognition with an Ngram language model

- Train HMMs for the acoustic model
- Train N-gram LM with backoff from training data
- Construct the Language graph, and from it the language HMM
  - Represent the Ngram language model structure as a compacted N-gram graph, as shown earlier
  - The graph must be dynamically constructed during recognition – it is usually too large to build statically
  - Probabilities on demand: Cannot explicitly store all K^N probabilities in the graph, and must be computed on the fly
    - K is the vocabulary size
  - Other, more compact structures, such as FSAs can also be used to represent the lanauge graph
    - later in the course
- Recognize