CS 505: Introduction to Natural Language Processing

Wayne Snyder
Boston University

Lecture 5: Language Models Continued: Smoothing, Perplexity

Generative Language Models: A Review

How to build an N-gram language model for some N (usually 2 … 4):

1. Preprocess your text/corpus into sentences, with boundary markers

   <s> this is the sentence </s>

2. Calculate the probability distribution of all K-Grams for $2 \leq K \leq N$

3. Sample from the distribution of bigrams with first token **<s>** to get first word w_1;

4. Sample from the distribution of trigrams with first tokens **<s>** w_1 to get second word w_2; etc. ….. until have

   <s> w_1 w_2 … w_{N-1} ;

5. Continue to sample from distribution of N-grams which match last N-1 words generated until </s> is generated.

A Bigram Example of an N-Gram Model

Text:

```
John likes to watch movies
Mary likes to play cards
John likes to play cards too but Mary likes to play cards more than John
```

1. Preprocess your text/corpus into sentences, with boundary markers

```
['<s>', 'john', 'likes', 'to', 'watch', 'movies', '</s>']
['<s>', 'mary', 'likes', 'to', 'play', 'cards', '</s>']
['<s>', 'john', 'likes', 'to', 'play', 'cards', 'too', 'but', 'mary', 'likes', 'to', 'play', 'cards', 'more', 'than',
'john', '</s>']
```

2. Calculate the probability distribution of all bigrams:

```
1 N_grams[2]
```

```
defaultdict(<function __main__.get_N_grams.<locals>.<lambda>()>,
            {('<s>', 'john'): 0.07142857142857142,
             ('john', 'likes'): 0.07142857142857142,
             ('likes', 'to'): 0.14285714285714285,
             ('to', 'watch'): 0.03571428571428571,
             ('watch', 'movies'): 0.03571428571428571,
             ('movies', '</s>'): 0.03571428571428571,
             ('<s>', 'mary'): 0.03571428571428571,
             ('mary', 'likes'): 0.07142857142857142,
             ('to', 'play'): 0.10714285714285714,
             ('play', 'cards'): 0.10714285714285714,
             ('cards', '</s>'): 0.03571428571428571,
             ('cards', 'too'): 0.03571428571428571,
             ('too', 'but'): 0.03571428571428571,
             ('but', 'mary'): 0.03571428571428571,
             ('cards', 'more'): 0.03571428571428571,
             ('more', 'than'): 0.03571428571428571,
             ('than', 'john'): 0.03571428571428571,
             ('john', '</s>'): 0.03571428571428571})
```

<s> john	2/28
john likes	2/28
likes to	4/28
to watch	1/28
watch movies	1/28
movies </s>	1/28
<s> mary	1/28
mary likes	2/28
to play	3/28
play cards	3/28
cards </s>	1/28
cards too	1/28
too but	1/28
but mary	1/28
cards more	1/28
more than	1/28
than john	1/28
john </s>	1/28

A Bigram Example of an N-Gram Model

```
John likes to watch movies
Mary likes to play cards
John likes to play cards too but Mary likes to play cards more than John
```

3,4. Sample from the distribution of bigrams with first token **<s>** to get first word w_1

$$P(\text{<s> john} \mid \text{<s>}) = \frac{P(\text{<s> john})}{P(\text{<s> } \cdots)} = \frac{2/28}{3/28} = 2/3$$

$$P(\text{<s> mary} \mid \text{<s>}) = \frac{P(\text{<s> mary})}{P(\text{<s> } \cdots)} = \frac{1/28}{3/28} = 1/3$$

```
<s> john          2/28
john likes        2/28
likes to          4/28
to watch          1/28
watch movies      1/28
movies </s>       1/28
<s> mary          1/28
mary likes        2/28
to play           3/28
play cards        3/28
cards </s>        1/28
cards too         1/28
too but           1/28
but mary          1/28
cards more        1/28
more than         1/28
than john         1/28
john </s>         1/28
```

```
from numpy.random import choice

choice( ['john','mary'], p=[2/3,1/3] )
```

Suppose the random sample gives us w_1 = 'john'

A Bigram Example of an N-Gram Model

```
John likes to watch movies
Mary likes to play cards
John likes to play cards too but Mary likes to play cards more than John
```

5. Continue to sample bigrams whose first word is the last word generated:

Sentence so far	Choices	Prob	Sample
<s> john	john </s>	2/3	
	john likes	1/3	likes
<s> john likes	likes to	1	to
<s> john likes to	to play	3/4	
	to watch	1/4	play
<s> john likes to play	play cards	1	cards
<s> john likes to play cards	cards </s>	1/3	
	cards too	1/3	
	cards more	1/3	</s>

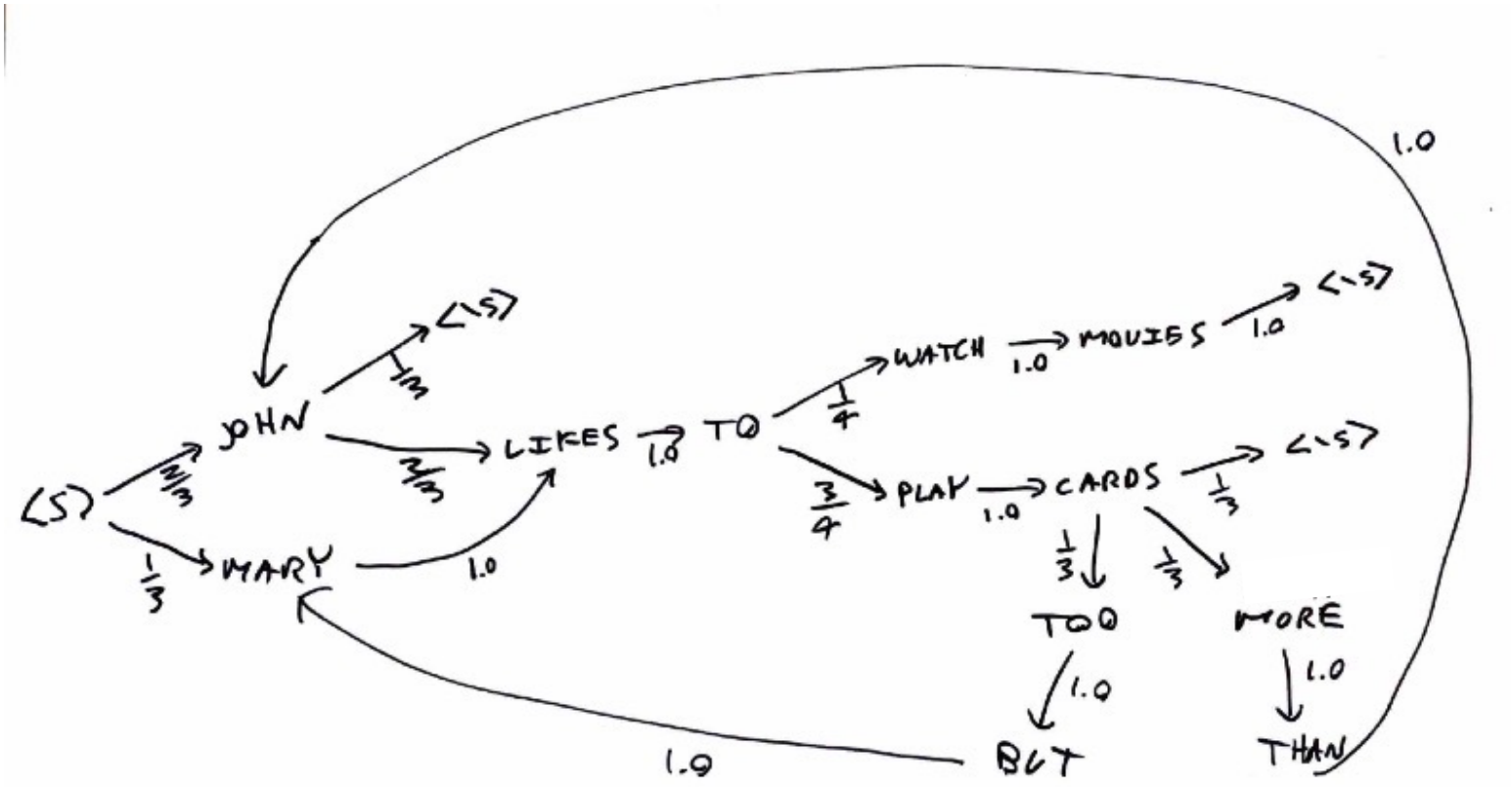
<s> john	2/28
john likes	2/28
likes to	4/28
to watch	1/28
watch movies	1/28
movies </s>	1/28
<s> mary	1/28
mary likes	2/28
to play	3/28
play cards	3/28
cards </s>	1/28
cards too	1/28
too but	1/28
but mary	1/28
cards more	1/28
more than	1/28
than john	1/28
john </s>	1/28

<s> john likes to play cards </s>

A Bigram Example of an N-Gram Model

```
John likes to watch movies
Mary likes to play cards
John likes to play cards too but Mary likes to play cards more than John
```

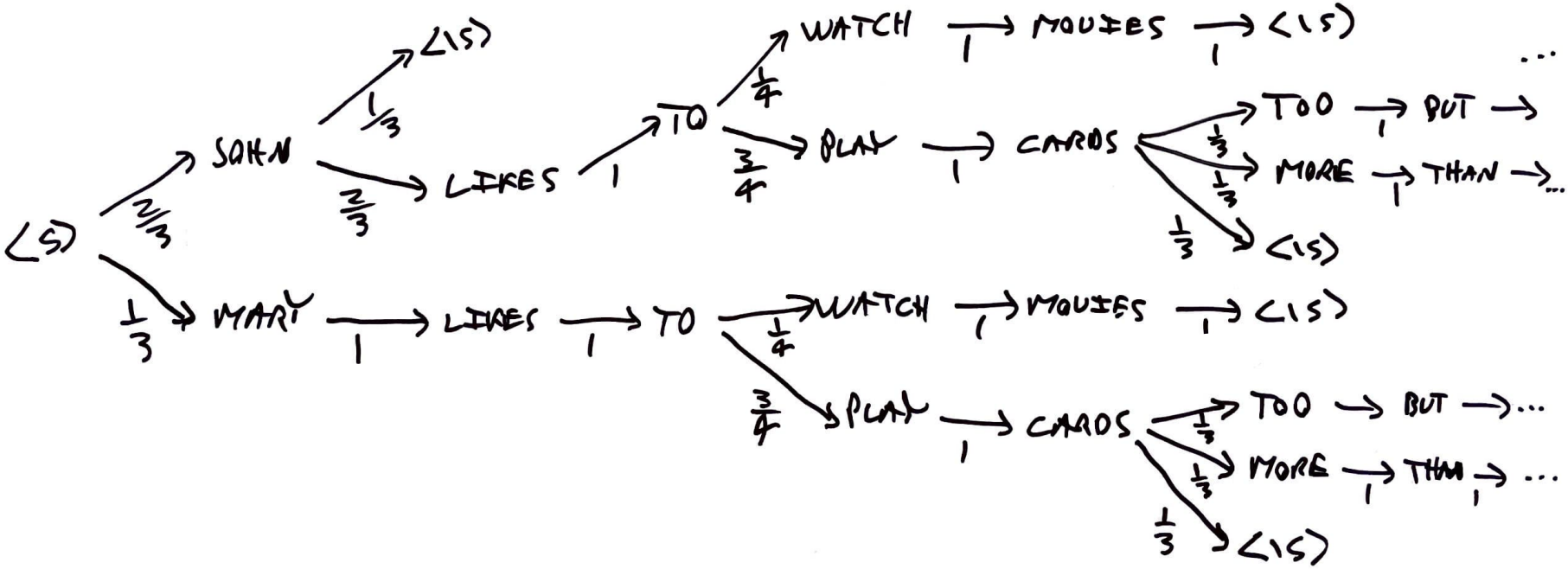
Not surprisingly, you can represent this as a Markov Chain:



A Bigram Example of an N-Gram Model

```
John likes to watch movies
Mary likes to play cards
John likes to play cards too but Mary likes to play cards more than John
```

Or as an infinite tree:



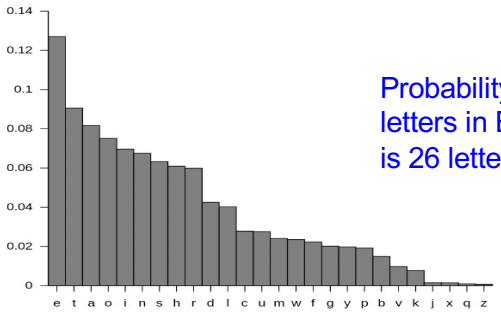
We'll come back to this soon….

Probabilistic LMs as Probability Distribution

Language models assign a probability to a sequence of tokens (letters, words, etc.)

Thus, a language model is a probability distribution!

Some LMs have a finite range:



Probability distribution of letters in English: Range is 26 letters.

But those we consider in this course have an infinite range, namely:

Sequences of tokens/words in the (infinite) language.

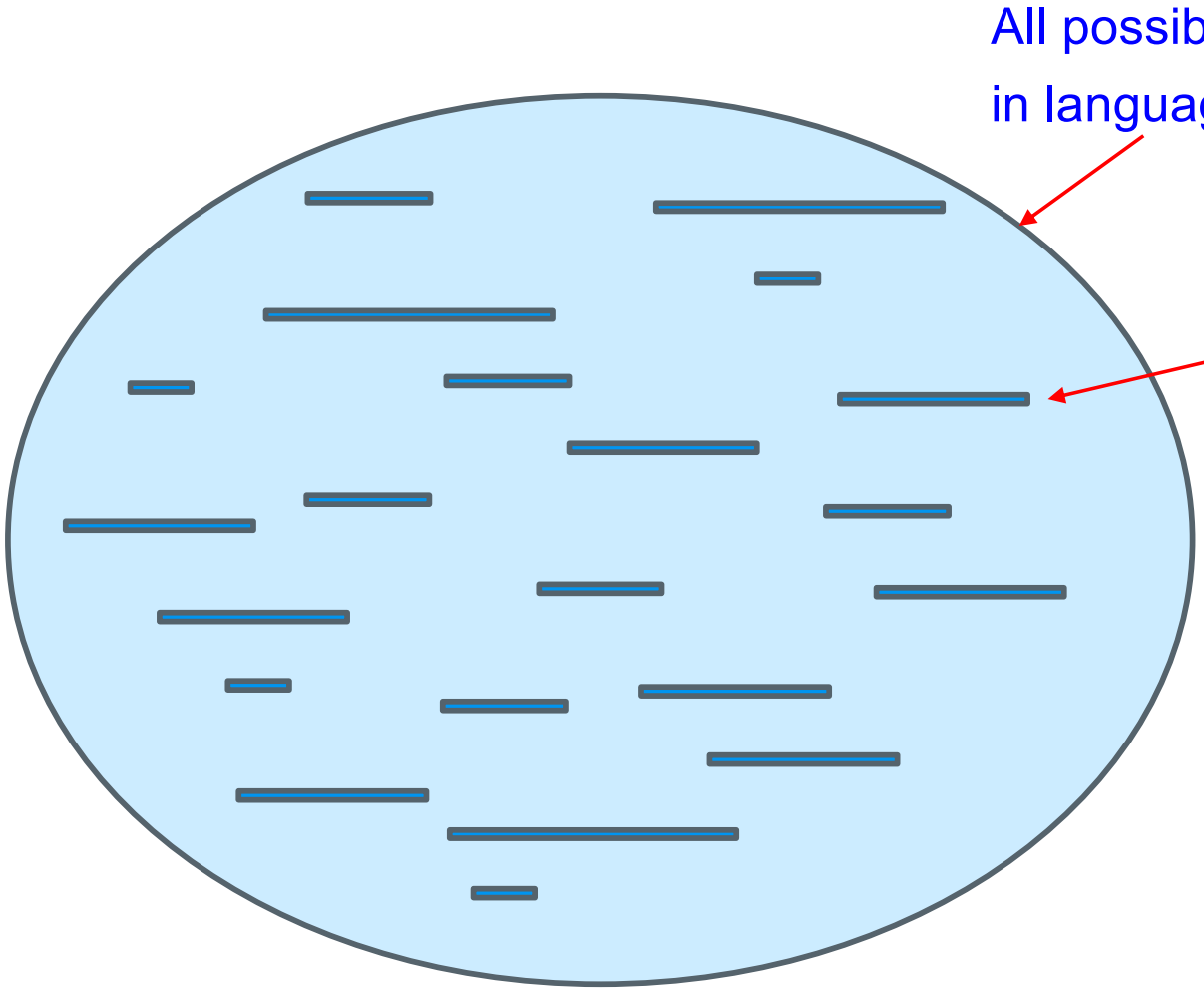
A data set is a finite sample of this infinite domain; put another way,

it is a discrete probability distribution with infinite domain, but have only a

finite number of sample points where the probability is non-zero.

Probabilistic LMs as Probability Distribution

So we have a finite approximation of an infinite discrete distribution, say of all sentences:



Classical Sampling Problem:

How well does this finite sample characterize the infinite language?

Probabilistic LMs as Sampled Distributions

Quality of the sample depends on:

- How large? (Bigger is better!)
- How representative of the language are the sample sentences?
  - Samples from news reports will not be representative of novels.
  - If you want to build a chat bot, sample from conversations!
  - General language models need diverse sources.

Two important issues about building good language models:

- How do we evaluate the quality of our language model?
- What do we do about missing information?

Evaluation of Language Models

How good is our LM?

Extrinsic evaluation of N-gram models uses information exterior to model:

Extrinsic evaluation for comparing models A and B:

Put each model in a task in real life:

Spelling corrector, speech recognizer, translation system

Run the task, get an accuracy for A and for B

How many misspelled words corrected properly?

How many words recognized/translated correctly?

Compare accuracy for A and B

Evaluation of Language Models

Difficulty of extrinsic (IRL) evaluation of N-gram models

Extrinsic evaluation: deploy your model IRL and measure it

- Time-consuming: accuracy is proportional to length of time, so can take days/weeks/months
- May be difficult, subject to proper design of experiment, statistical analysis, etc.
- May be impossible: How would you test an NLP system used on first manned mission to Mars?

So, at least in the development phase, we need an intrinsic model…

Intrinsic testing of LM using Train/Test Split

Randomly permute the set of sentences, then separate into

- Training Set (e.g., 80%)
- Testing Set (e.g., 20%)

- Create your model from the Training Set (create N-Gram distributions, train a network, etc.),
- Evaluate how likely your Testing Set is using the model: the sentences in the Testing Set should be probable!

A very common metric is perplexity….

Perplexity

This explaination is due to Fabio Chiusano in Two Minutes NLP (posted on the web site).

A LM (a probability distribution over sequences of tokens) can

- Evaluate the "goodness" of sequences (e.g., N-grams, sentences), and
- Generate plausible sequences (as if a human wrote them).

A LM should give a higher probability to a well-written text, and be "perplexed" by a badly-written text.

The perplexity of badly-written text is large, and of a well-written text is small.

"Thus, the perplexity metric in NLP is a way to capture the degree of 'uncertainty' a model has in predicting (i.e. assigning probabilities to) text."

Perplexity

Example 1

Suppose our language has vocabulary

V = { " "a", "the", "red", "fox", "dog", "and"}

and we want our LM to predict the probability of

W = "a red fox ."

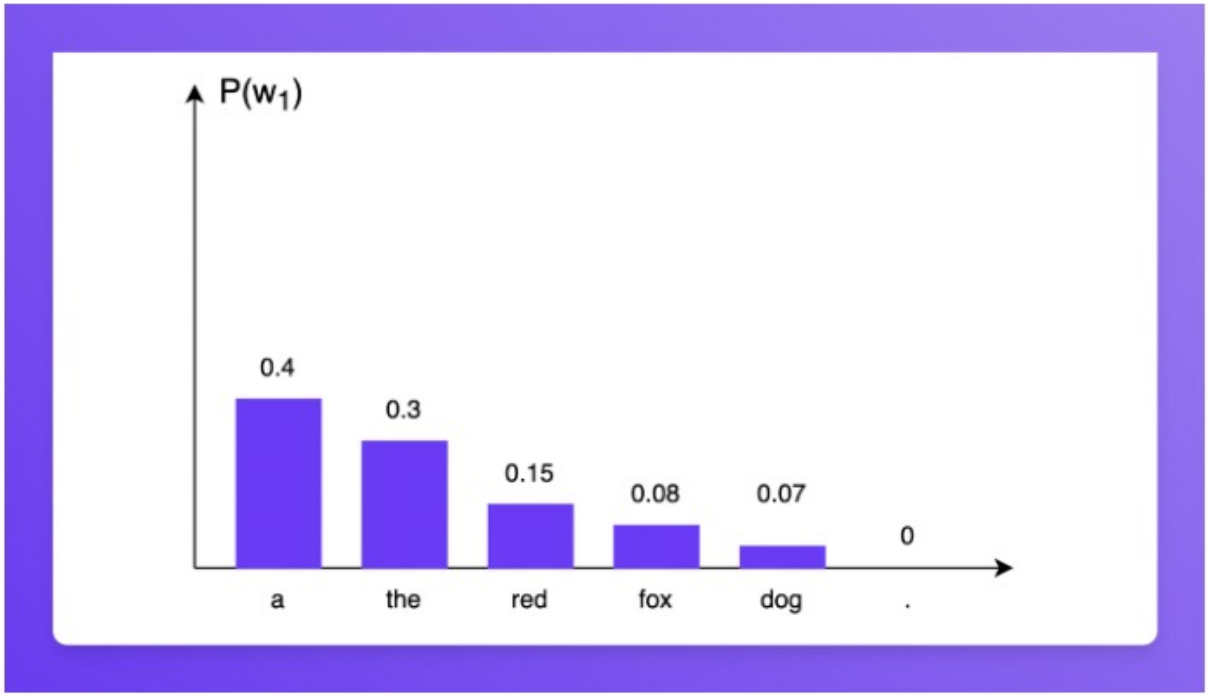
Thus:

P(W) = P("a") * P("red" | "a") * P("fox" | "a red") * P("." | "a red fox")

Perplexity

W = “a red fox”

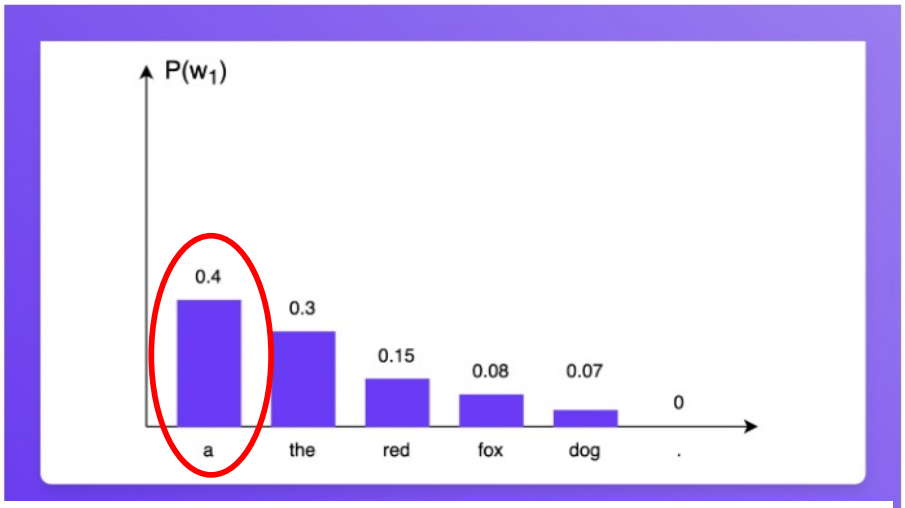
Suppose our LM assigns these probabilities to the first word in a sentence:

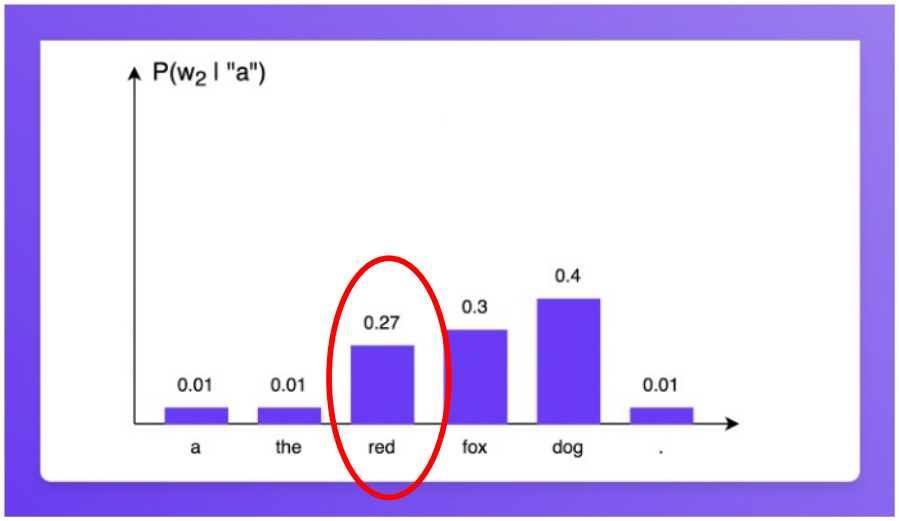


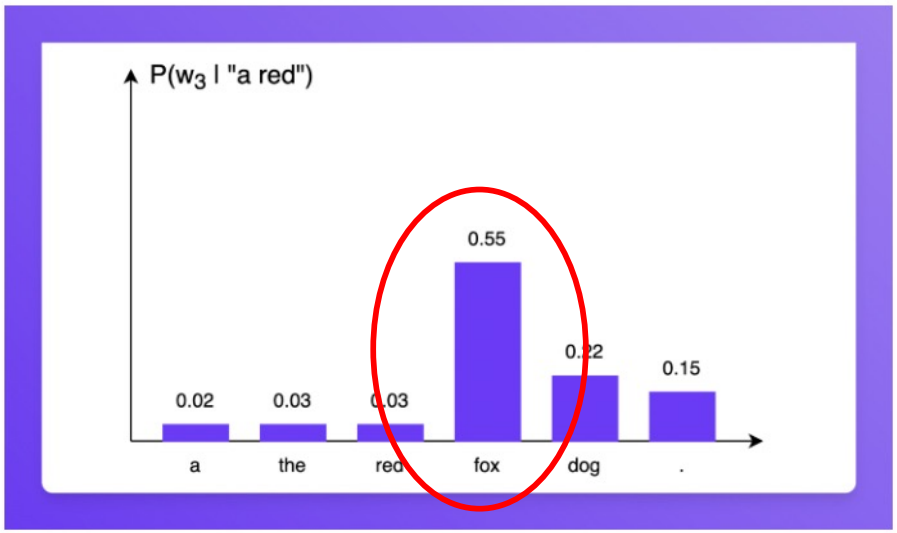
Perplexity

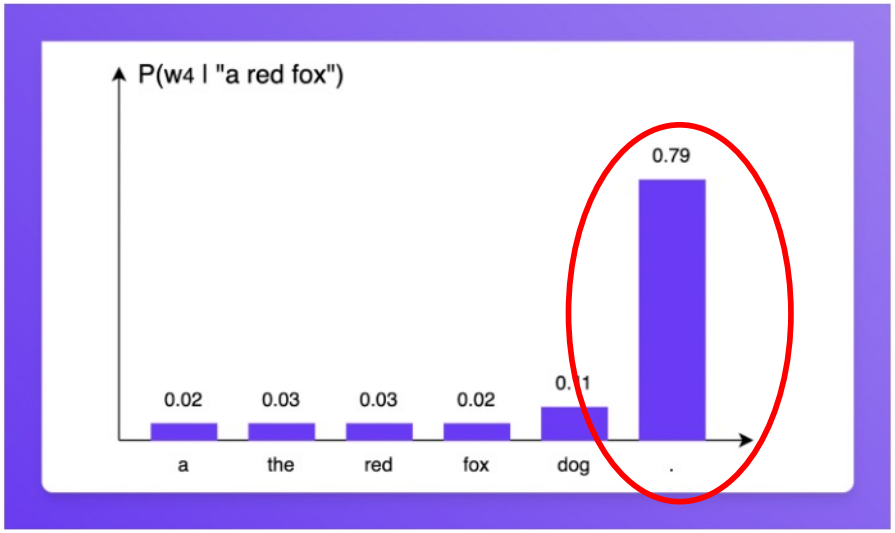
W = “a red fox .”

Suppose our LM assigns these probabilities:









Perplexity

Thus:

$$P(W) = P(\text{“a”}) * P(\text{“red” | “a”}) * P(\text{“fox” | “a red”}) * P(\text{“.” | “a red fox”})$$

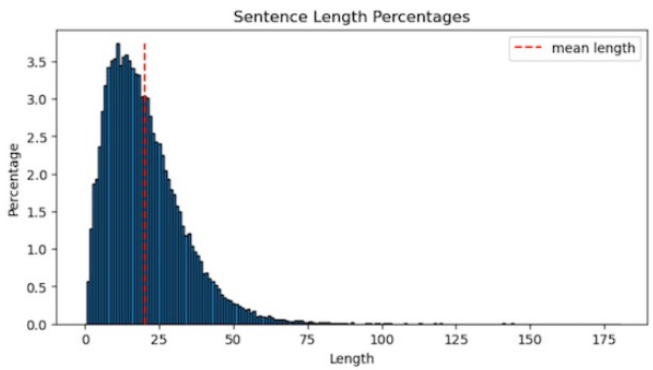
$$= 0.4 * 0.27 * 0.55 * 0.79$$

$$= 0.0469$$

BUT, notice that the product of probabilities gets smaller and smaller as the sentences gets longer! So:

P(“a red fox .”) > P(“a red fox and a dog .”) ?

That’s not what happens in natural language!



Perplexity

The quality of sentences should NOT be inversely proportional to their length, so we will normalize by their length...

The usual way we take the mean of numbers being multiplied is using the Geometric Mean instead of the Arithmetic mean:

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = \sqrt[n]{x_1 x_2 \cdots x_n}$$

or, equivalently, as the arithmetic mean in logscale:

$$\exp\left(\frac{1}{n}\sum_{i=1}^{n} \ln a_i\right)$$

Question: what happens if one of the numbers is 0?

This is something we'll have to deal with!

Digression: How do we calculate probabilities in machine learning?

We do everything in log space!

- Avoid loss of precision from underflow (prob p might be tiny)
- Adding is *much* faster than multiplying
- log is monotonic, so it preserves order for probs ($p \geq 0$):

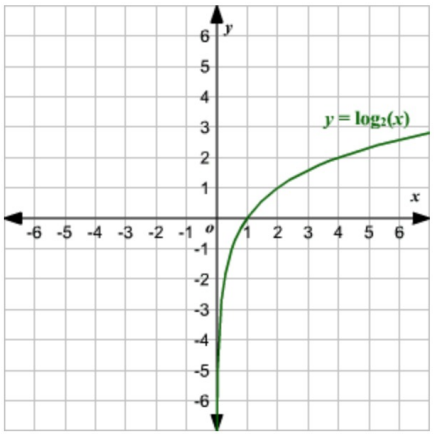
$$p < q \;\leftrightarrow\; \log(p) < \log(q)$$

- Can easily recover probs using exp(...)

$p_1 * p_2 * \cdots * p_n$

log ↓ ↑ exp(…)

$$\log(\, p_1 * p_2 * \cdots * p_n \,) = \log(p_1) + \log(p_2) + \cdots + \log(p_n)$$



Perplexity

For $W = w_1 w_2 \ldots w_n$ let us define the normalized version of P(W) using the Geometric Mean::

$$\text{Pnorm}(W) = P(W)^{1/n} = \sqrt[n]{P(W)}$$

and so

$$\text{Pnorm}(\text{"a red fox."}) = \sqrt[4]{P(\text{"a red fox."})} = \sqrt[4]{0.0469} = 0.465$$

Thus: a well-written sentence will have a large Pnorm, and a poorly-written sentence will have a small Pnorm. But remember, we want the opposite, so perplexity is just the reciprocal of the Pnorm:

$$PP(W) = \frac{1}{Pnorm(W)} = \frac{1}{P(W)^{1/n}} = \sqrt[n]{\frac{1}{P(W)}} = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

Remember: Low perplexity is good, high perplexity is bad!

Perplexity Examples

Let's suppose a sentence of length N consists of random bits, e.g.,

W = 101111

What is the perplexity of this sentence according to a model that gives a uniform probability to each bit, i.e., exactly 0.5?

No matter how long the sentence is, the perplexity is 2, meaning, you always are "perplexed" as to which of the 2 bits will be next:

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} = (0.5^N)^{-\frac{1}{N}} = 0.5^{-1} = 2$$

Perplexity Examples

Now suppose that the probability of a 1 is 3 times the probability of a 0, i.e., P(1) = 0.75 and P(0) = 0.25.

W = 10111

What is the perplexity of this sentence according to this model?

Intuitively, it should be less surprising than in the previous model, because you would expect there to be more 1's than 0's:

$$PP(W) = P(10111)^{-\frac{1}{5}} = (0.75 * 0.25 * 0.75 * 0.75 * 0.75)^{-\frac{1}{5}} = 1.66$$

The perplexity of a string of all 1's is always $\frac{1}{0.75} = 1.3333$

The perplexity of a string of all 0's is always $\frac{1}{0.25} = 4.0$

Perplexity Examples

What is the perplexity of

"John"

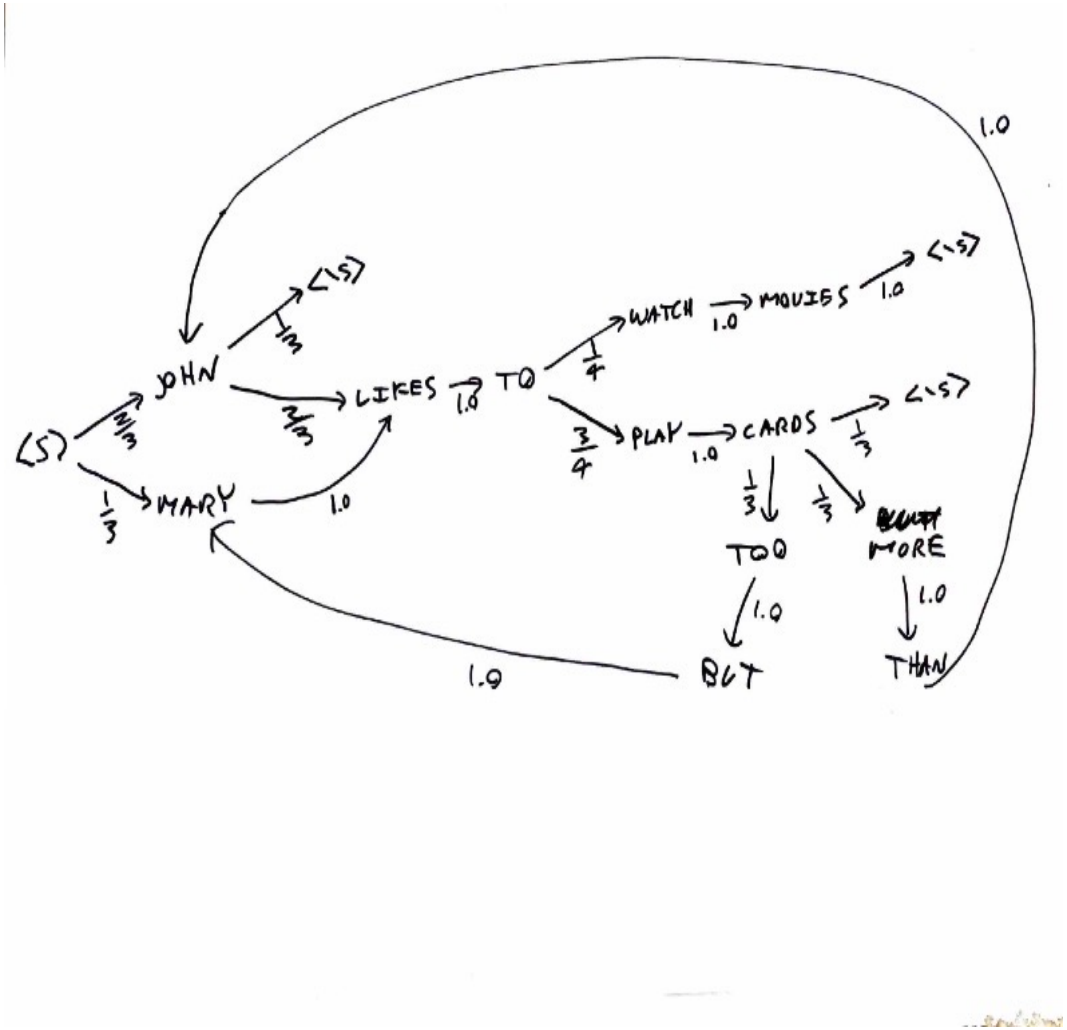
$$(2/3 * 1/3)^{-\frac{1}{2}} = 1.074$$

"Mary likes to watch movies"?

$$(1/3 * 1 * 1 * 1/3 * 1 * 1)^{-\frac{1}{5}} = 1.182$$

"John likes to play cards more than John likes to play cards too but Mary likes to play cards more than John likes to watch movies"

0.759. (?!!!)



Perplexity

The best language model is one that best predicts an unseen test set

Perplexity is the inverse probability of the test set, normalized by the number of words:

$$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Chain rule:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}$$

For bigrams:

$$\mathrm{PP}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_{i-1})}}$$

Lower perplexity = better model

To test a model, training it on training set, test it on testing set:
The quality of the model is the perplexity of the entire test set, considered as one long string!

Example: Training 38 million words, test 1.5 million words, WSJ

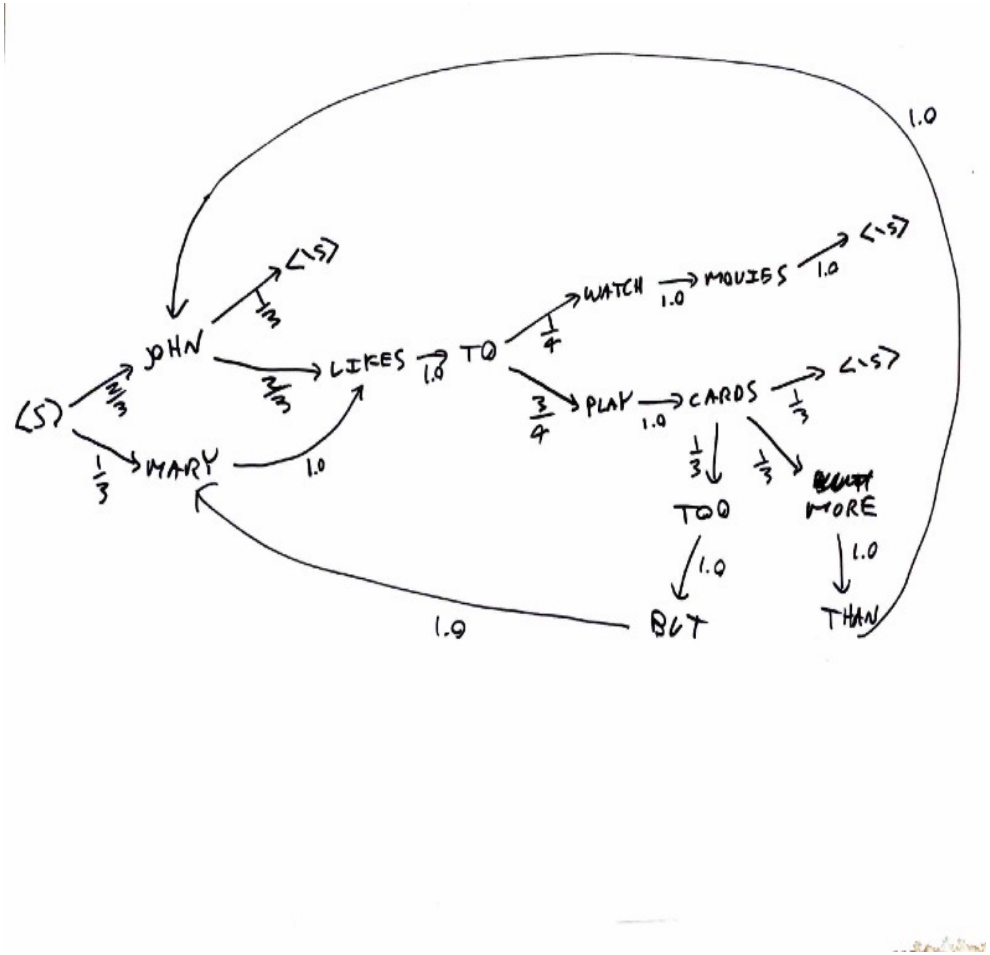
N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

Perplexity Examples

What is the perplexity of

“Mary likes to watch movies with John”

“Mary likes to watch cards”?



$$\frac{1}{0} = \infty$$

Approximating Shakespeare

Generating Shakespeare with N-Grams models:

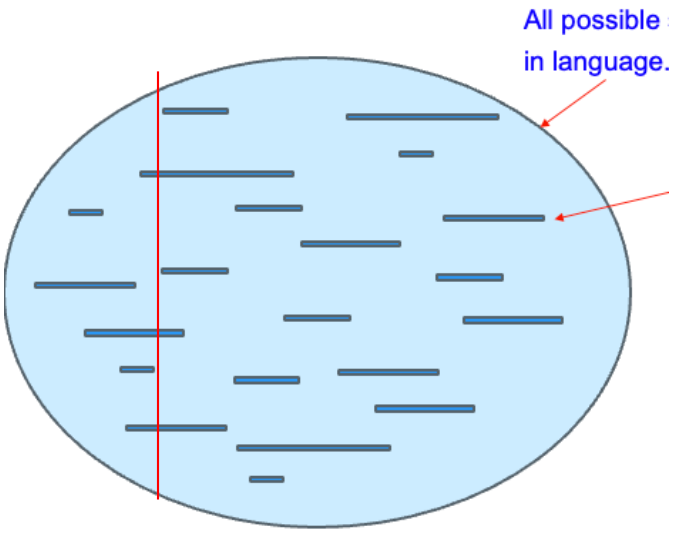
1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Shakespeare as corpus

- N = 884,647 tokens, vocabulary size V =29,066
- Shakespeare produced 300,000 bigram types out of V^2= 844 million possible bigrams.
  - So 99.96% of the possible bigrams were never seen (have zero entries in the table)
- Quadrigrams worse: What's coming out looks like Shakespeare because it *is* Shakespeare

The perils of overfitting

- **N-grams only work well for word prediction if the test corpus looks like the training corpus**
  - In real life, it often doesn't
  - We need to train robust models that generalize!
  - One kind of generalization: Zeros!
    - Things that don't ever occur in the training set
      - But occur in the test set



Zeros

- **Training set:**
  … denied the allegations
  … denied the reports
  … denied the claims
  … denied the request

  P(“offer” | denied the) = 0

- Test set
  … denied the offer
  … denied the loan

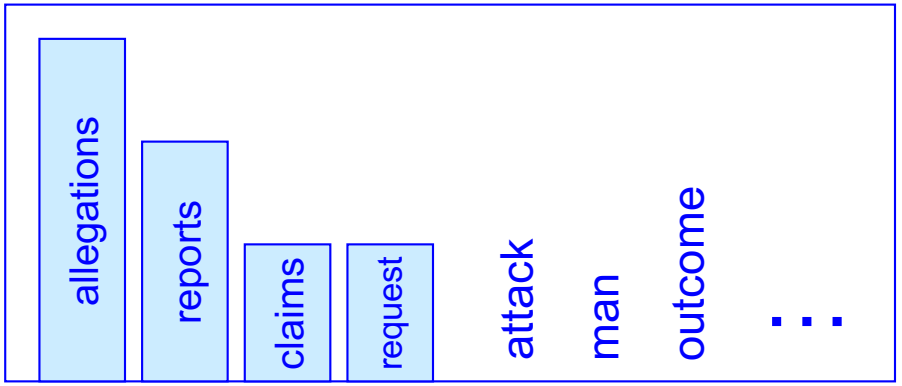
Zero probability bigrams

- **Bigrams with zero probability**
  - mean that we will assign 0 probability to the test set!
- **And hence we cannot compute perplexity (can't divide by 0)!**

The intuition of smoothing (from Dan Klein)

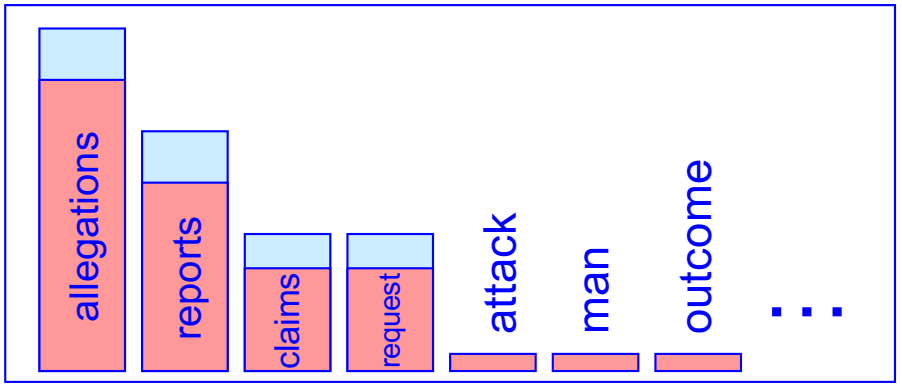
- **When we have sparse statistics:**

P(w | denied the)
3 allegations
2 reports
1 claims
1 request
7 total



- **Steal probability mass to generalize better**

P(w | denied the)
2.5 allegations
1.5 reports
0.5 claims
0.5 request
2 other
7 total



Add-one estimation

- Also called Laplace smoothing
- Pretend we saw each word one more time than we did
- Just add one to all the counts!

- Normal (Most Likely Estimate): $$P_{MLE}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

- Add-1 estimate: $$P_{Add-1}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i) + 1}{c(w_{i-1}) + V}$$

Add-1 estimation is a blunt instrument

- Add-1 isn't optimal for N-grams
  - We'll see better methods in next slides
- But add-1 is used to smooth other NLP models
  - For text classification
  - In domains where the number of zeros isn't so large.

Backoff and Interpolation

- **Sometimes it helps to use less context**
  - Condition on less context for contexts you haven't learned much about
- **Backoff:**
  - use trigram if you have good evidence,
  - otherwise bigram, otherwise unigram
- **Interpolation:**
  - Weighted average of unigram, bigram, trigram, learn weights by training

N-gram Smoothing Summary

Used to deal with missing data

- Add-1 smoothing:
  - OK for text categorization, not for language modeling
- Backoff and Interpolation
  - Learn weights for interpolation
- Combination approaches
  - Extended Interpolated Kneser-Ney (state of the art, covered in the text)