CAS CS 505
Lec 4

Language Models Continued: Smoothing, Perplexity

1. Generative Language Models: A Review
    a. Preprocess your text/corpus into sentences, with boundary markers
        i. <s> this is the sentence </s>
    b. Calculate the probability distribution of all K-Grams for $2 \leq K \leq N$
    c. Sample from the distribution of bigrams with first token <s> to get the first word w_1
    d. Sample from the distribution of trigrams with first tokens <s> w_1 to get second word w2: etc. … until have
        i. <s> w_1 w_2 … w_{n-1}
    e. Continue to sample from distribution of N-grams which match last N-1 words generated until </s> is generated.
2. A Bigram Example of an N-gram Model
    a. 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
```

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

```
['<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>']
```

        ii. Might include punctuation at the end to check when the sentences are done
    c. Calculate the probability distribution of all bigrams:
        i.

```
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

d. Sample from the distribution of bigrams with first token <s> to get the first word w1

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

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

```
from numpy.random import choice

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

i. Suppose the random sample gives us **w_1 = 'john'**

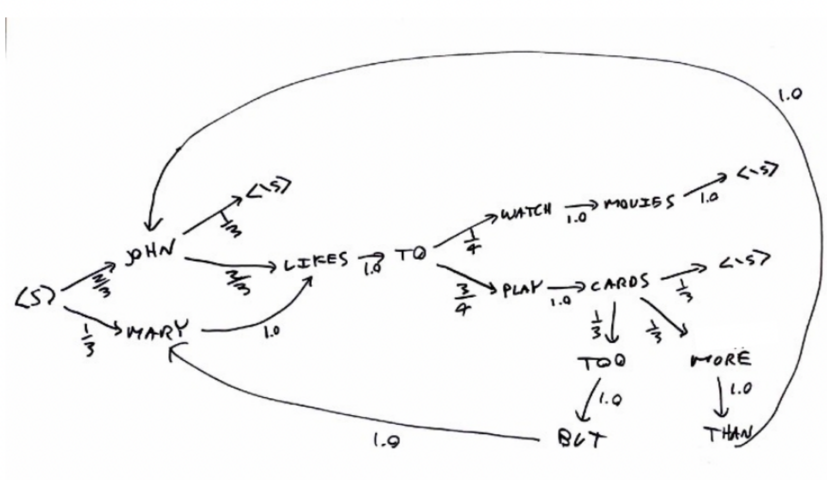
ii. Don't choose the most frequent one, but do it randomly according to the probability so that we do not get the same sentence every time

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

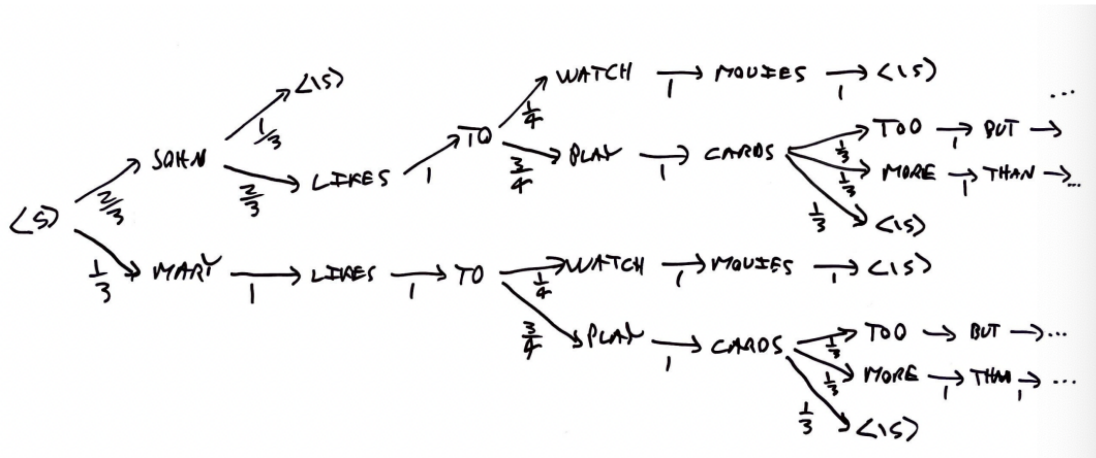
i.

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>

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

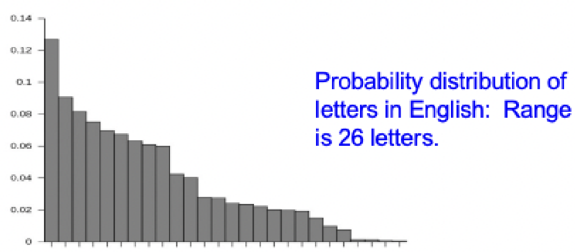


f.

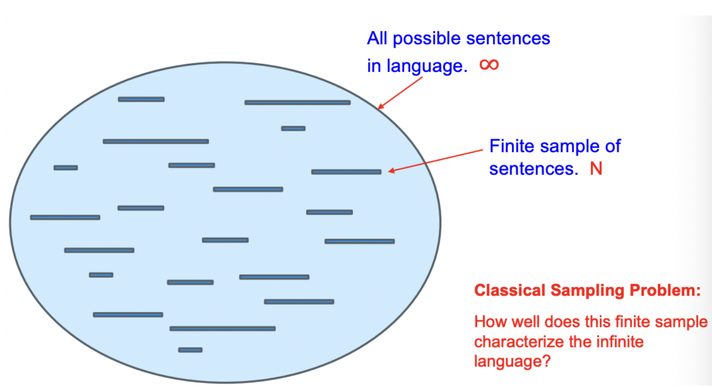


g.
h. How to come up with good sentences? → Come back later
i. We are going from left to right in terms of writing sentences

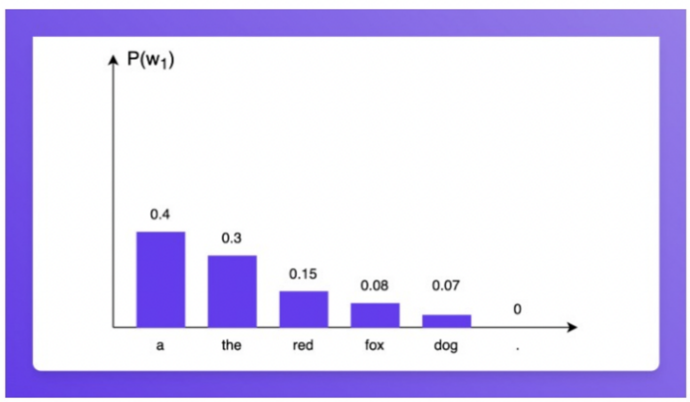
3. Probabilistic LMs as Probability Distribution
   a. Language models assign a probability to a sequence of tokens (letters, words, etc.)
   b. Thus, a language model is a probability distribution!
   c. Some LMs have a finite range:

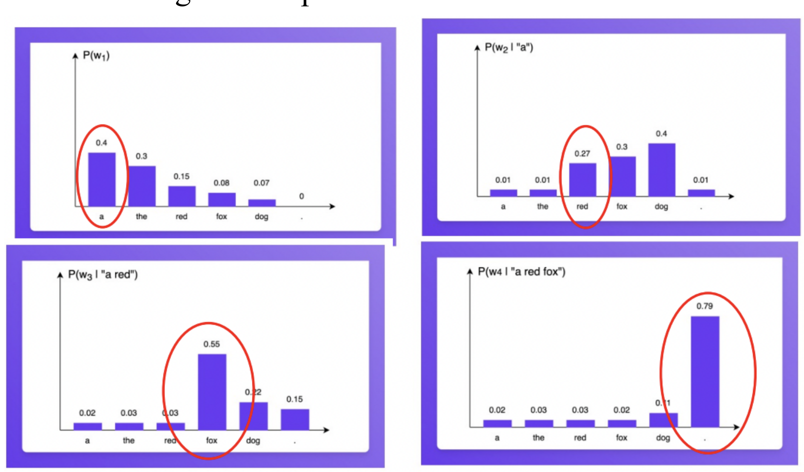


   d. But those we consider in this course have an infinite range, namely:
      i. Sequences of tokens/words in the (infinite) language.
   e. A data set is a finite sample of this infinite domain; put another way, it is a discrete probability distribution with infinite domain, but have only a finite number of sample points where the probability is non-zero.
   f. So we have a finite approximation of an infinite discrete distribution, say of all sentences:
      i.



    g. Quality of the sample depends on:
        i. How large? (Bigger is better!)
        ii. How representative of the language are the sample sentences?
            1. Samples from news reports will not be representative of novels.
            2. If you want to build a chat bot, sample from conversations!
            3. General language models need diverse sources.
    h. Two important issues about building good language models
        i. How do we evaluate the quality of our language model?
        ii. What do we do about missing information?
4. Evaluation of Language Models
    a. How good is our LM?
    b. Extrinsic evaluation of N-gram models uses information exterior to model:
    c. Extrinsic evaluation for comparing models A and B:
        i. Put each model in a task in real life
            1. Spelling corrector, speech recognizer, translation system
        ii. Run the task, get an accuracy for A and for B
            1. How many misspelled words corrected properly?
            2. How many words recognized/translated correctly?
        iii. Compare accuracy for A and B
    d. Difficulty of extrinsic (IRL) evaluation of N-gram models
        i. Extrinsic evaluation: deploy your model IRL and measure it
            1. Time-consuming: accuracy is proportional to length of time, so can take days/weeks/months
            2. May be difficult, subject to proper design of experiment, statistical analysis, etc.
            3. May be impossible: How would you test an NLP system used on first manned mission to Mars?
    e. So, at least in the development phase, we need an intrinsic model…
5. Intrinsic Testing of LM using Train/Test Split
    a. Randomly permute the set of sentences, then separate into
        i. Training Set (e.g., 80%)
        ii. Testing Set (e.g., 20%)
    b. Create your model from the Training Set (create N-Gram distributions, train a network, etc.)
    c. Evaluate how likely your Testing Set is using the model: the sentences in the Testing Set should be probable!

6. Perplexity
    a. A LM (a probability distribution over sequences of tokens) can
        i. Evaluate the “goodness” of sequences (e.g., N-grams, sentences), and
        ii. Generate plausible sequences (as if a human wrote them).
    b. A LM should give a higher probability to a well-written text, and be “perplexed” by a badly-written text.
    c. The perplexity of badly-written text is large, and of a well-written text is small.
    d. “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.”
    e. Example 1
        i. Suppose our language has vocabulary
            1. V = { “ “a”, “the”, “red”, “fox”, “dog”, ”and”}
        ii. we want our LM to predict the probability of
            1. W = “a red fox .”
        iii. Thus:
            1. P(W) = P(“a”) ∗ P(“red” | “a”) ∗ P(“fox” | “a red”) ∗ P(“.” | “a red fox”)
        iv. Suppose our LM assigns these probabilities to the first word in a sentence:
            1. 

        v. Suppose our LM assigns these probabilities:
            1. 

vi. Thus
   1. P(W) = P("a") ∗ P("red" | "a") ∗ P("fox" | "a red") ∗ P("." | "a red fox") = 0.4 * 0.27 * 0.55 * 0.79 = 0.0469

vii. BUT, notice that the product of probabilities gets smaller and smaller as the sentences gets longer! So:
   1. P("a red fox .") > P("a red fox and a dog .") ?

viii. That's not what happens in natural language

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

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

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

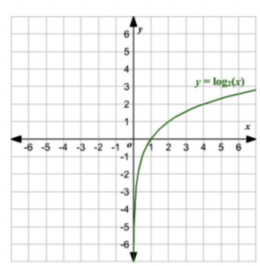
or, equivalently, as the arithmetic mean in logscale:

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

   i.

7. Digression: How do we calculate probabilities in machine learning?
   a. We do everything in log space! o Avoid loss of precision from underflow (prob p might be tiny)
      i. Adding is much faster than multiplying o log is monotonic, so it preserves order for probs (p ≥ 0):
         1. $p < q \leftrightarrow \log p < \log(q)$
      ii. Can easily recover probs using exp(...)

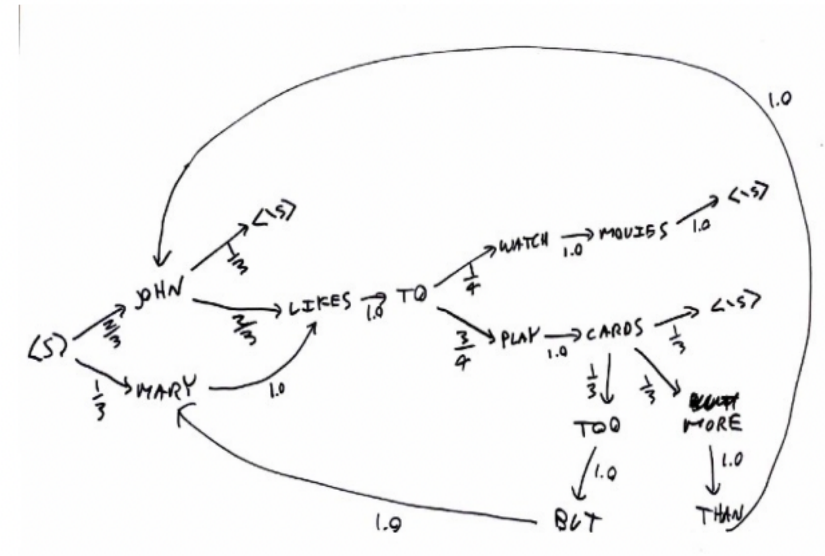
$$\begin{array}{c} p_1 * p_2 * \cdots * p_n \\ \text{log} \downarrow \quad\quad \uparrow \text{exp(...)} \\ \log(p_1 * p_2 * \cdots * p_n) = \log(p_1) + \log(p_2) + \cdots + \log(p_n) \end{array}$$



   b.
   c. Why use log?
      i. You can store more information in the log
      ii. Addition is faster than multiplication

8. Perplexity (cont.)
    a. For W = w1w2...wn let us define the normalized version of P(W) using the Geometric Mean:
        i. $$\text{Pnorm}(W) = P(W)^{1/n} = \sqrt[n]{P(W)}$$
    b. and so
        i. $$\text{Pnorm}(\text{"a red fox."}) = \sqrt[4]{P(\text{"a red fox."})} = \sqrt[4]{0.0469} = 0.465$$
    c. 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:
        i. $$PP(W) = \frac{1}{Pnorm(W)} = \frac{1}{P(W)^{1/n}} = \sqrt[n]{\frac{1}{P(W)}} = P(w_1 w_2 ... w_N)^{-\frac{1}{N}}$$
    d. Remember: Low perplexity is good, high perplexity is bad!
9. Perplexity Examples
    a. Let's suppose a sentence of length N consists of random bits, e.g.,
        i. W = 101111
    b. What is the perplexity of this sentence according to a model that gives a uniform probability to each bit, i.e., exactly 0.5?
    c. 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:
        i. $$PP(W) = P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} = (0.5^N)^{-\frac{1}{N}} = 0.5^{-1} = 2$$
    d. 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.
    e. W = 10111
    f. What is the perplexity of this sentence according to this model?
    g. Intuitively, it should be less surprising than in the previous model, because you would expect there to be more 1's than 0's:
        i. $$PP(W) = P(10111)^{-\frac{1}{5}} = (0.75 * 0.25 * 0.75 * 0.75 * 0.75)^{-\frac{1}{5}} = 1.66$$
    h. The perplexity of a string of all 1's is always $\frac{1}{0.75} = 1.3333$
    i. The perplexity of a string of all 0's is always $\frac{1}{0.25} = 4.0$

j. Given the figure here, what is the perplexity of the following?



i. "John"

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

ii. "Mary likes to watch movies"?

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

iii. 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"

1. 0.759

iv. "Mary likes to watch cards"?

1. 1/0 = infinity → worst perplexity since the combination of words don't exist

10. Perplexity

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

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

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

c. Chain rule:

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

d. For bigrams:

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

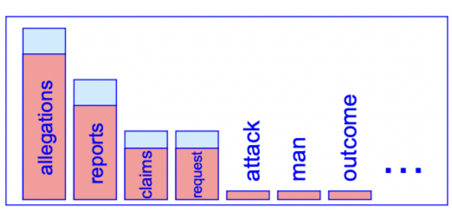
11. Lower perplexity = better model
    a. To test a model, training it on training set, test it on testing set: The quality of the model is the perplexity of the entire test set, considered as one long string!
       i. 

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

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

12. Shakespeare as Corpus
    a. N = 884,647 tokens, vocabulary size V =29,066
    b. Shakespeare produced 300,000 bigram types out of V^2= 844 million possible bigrams.
       i. So 99.96% of the possible bigrams were never seen (have zero entries in the table)
    c. Quadrigrams worse: What's coming out looks like Shakespeare because it is Shakespeare
13. The Perils of overfitting
    a. N-grams only work well for word prediction if the test corpus looks like the training corpus
       i. In real life, it often doesn't
       ii. We need to train robust models that generalize!
       iii. One kind of generalization: Zeros!
            1. Things that don't ever occur in the training set
            2. But occur in the test set

14. Zeros
    a. Training set:
        i. … denied the allegations
        ii. … denied the reports
        iii. … denied the claims
        iv. … denied the request
    b. Test set
        i. … denied the offer
        ii. … denied the loan
    c. P("offer" | denied the) = 0
15. Zero probability bigram
    a. Bigrams with zero probability
        i. mean that we will assign 0 probability to the test set!
    b. And hence we cannot compute perplexity (can't divide by 0)!
16. The intuition of smoothing (from Dan Klein)
    a. When we have sparse statistics:
        i. P(w | denied the)
            1. 3 allegations
            2. 2 reports
            3. 1 claims
            4. 1 request
            5. (7 total)
        ii. 

    b. Steal probability mass to generalize better
        i. P(w | denied the)
            1. 2.5 allegations
            2. 1.5 reports
            3. 0.5 claims
            4. 0.5 request
            5. 2 other
            6. 7 total
        ii. 

17. Add-one estimation
    a. Also called Laplace smoothing
    b. Pretend we saw each word one more time than we did
    c. Just add one to all the counts!
    d. Normal (Most Likely Estimate):

$$P_{MLE}(w_i \mid w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

    e. Add-1 estimate:

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

18. Add-1 estimation is a blunt instrument
    a. Add-1 isn't optimal for N-grams
        i. We'll see better methods in next slides
    b. But add-1 is used to smooth other NLP models
        i. For text classification
        ii. In domains where the number of zeros isn't so large.
19. Backoff and Interpolation
    a. Sometimes it helps to use less context
        i. Condition on less context for contexts you haven't learned much about
    b. Backoff:
        i. use trigram if you have good evidence, • otherwise bigram, otherwise unigram
    c. Interpolation:
        i. Weighted average of unigram, bigram, trigram, learn weights by training
20. N-gram Smoothing Summary
    a. Used to deal with missing data
    b. Add-1 smoothing:
        i. OK for text categorization, not for language modeling o
    c. Backoff and Interpolation
        i. Learn weights for interpolation
    d. Combination approaches
        i. Extended Interpolated Kneser-Ney (state of the art, covered in the text)