Sequence Modeling

Part 1: N-grams

Aaron Mueller
CAS CS 505: Introduction to Natural Language Processing
Spring 2026
Boston University

[Many slides inspired by a lecture by Mohit Iyyer.]

Admin

- **HW0** is due this coming Tuesday, **Feb. 3**!
  - There are *two* Gradescope submissions: one for your code, and one for your written answers.
  - You should upload your homeworks individually.

- Optional last-minute homework help session! This will happen at your Monday lab section on **Feb. 2.**
  - Come with questions! This will basically be office hours++.

Overview of Concepts

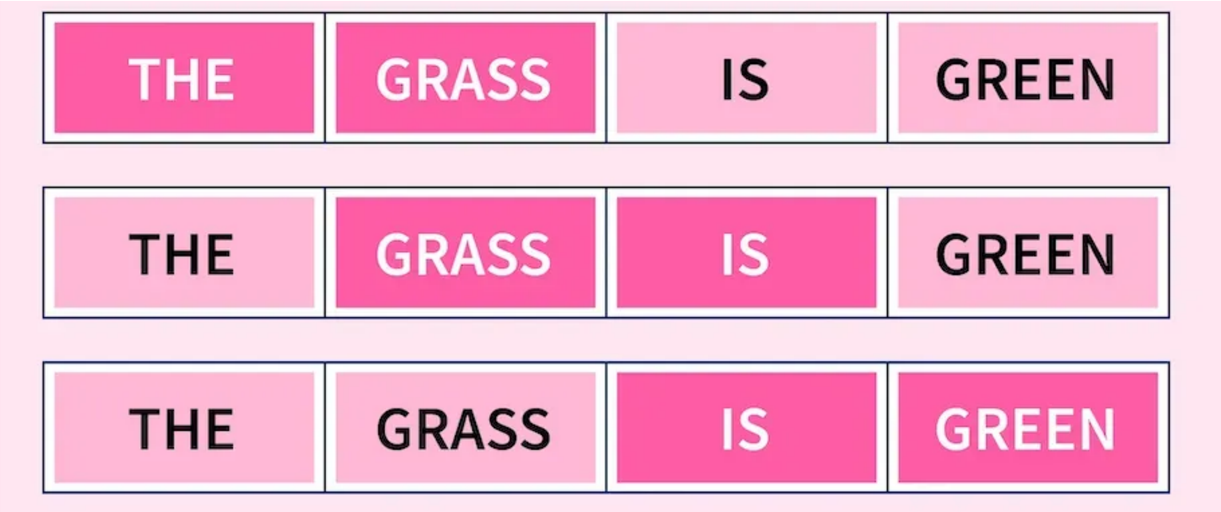
An **n-gram** is an ordered list of tokens.

A **language model** is a system that produces probabilities over *sequences of* tokens.

We use the **chain rule** of probabilities to break sequence probabilities into a series of conditional continuation probabilities.

A **Markov assumption** allows us to ignore distant context.

Smoothing, interpolation, and **backoff** allow LMs to gracefully handle unseen token sequences.



Why would you want to assign probability to text?

- Translation:

  $p(\text{he drove to the restaurant}) \gg p(\text{he flew to the restaurant})$

- Speech recognition:

  $p(\text{NLP systems recognize speech}) \gg p(\text{In elpy systems, wreck a nice beach})$

- Autocomplete:

  $p(\text{drawing is } \textbf{fun}) \gg p(\text{drawing is } \textbf{JsonObject})$

A Brief History of Language Modeling

- 1910s: N-grams
- 1940s: (Cross-)entropy
- 1950s: Chomsky, context-free grammars
- 1980s-1990s: Recurrent neural networks, LSTMs
- 2000s: Neural language models
- 2010s - present: Transformers, large language models

Today, we'll start with the simplest language model: n-grams.

Language Models

A language model is a system that assigns a probability to a sequence of tokens:

$$p(x_1, x_2, \ldots, x_n)$$

Using the **chain rule**, we can break this down into a product of conditional next-token probabilities:

$$p(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_t \mid x_{<t})$$

Thus, a language model is a system that produces probability distributions over next tokens given prior context.

Reminder: The Chain Rule

- Recall the definition of conditional probabilities:

$$p(B \mid A) = \frac{p(A, B)}{p(A)} \qquad \text{Equivalently: } p(A, B) = p(A)p(B \mid A)$$

- We can add arbitrarily many variables to this equation:

$$p(A, B, C, D) = p(A)p(B \mid A)p(C \mid A, B)p(D \mid A, B, C)$$

- In general:

$$p(x_1, \ldots, x_n) = p(x_1)p(x_2 \mid x_1) \ldots p(x_n \mid x_1, \ldots, x_{n-1})$$

Applying the Chain Rule

$$p(w_1 w_2 \ldots w_n) = \prod_i p(w_i \mid w_1 w_2 \ldots w_{i-1})$$

Prefix/Context

$$p(\text{The water was remarkably clear}) =$$

$$p(\text{The}) \times p(\text{water} \mid \text{The}) \times p(\text{was} \mid \text{The water})$$

$$\times\, p(\text{remarkably} \mid \text{The water was})$$

$$\times\, p(\text{clear} \mid \text{The water was remarkably})$$

Estimating Sequence Probabilities

- Let's try to count and divide:

$$p(\text{Maine}|\text{I like to vacation in}) = \frac{C(\text{I like to vacation in Maine})}{C(\text{I like to vacation in})}$$

- How likely are you to have seen this entire sentence in a corpus?
- Many sentences you hear/read have never been uttered before!
  - But we *really* want to avoid $p = 0$.

- To handle this, we will make a **Markov assumption.**

Markov Assumption

- Simplifying assumption:

$$p(\text{Maine} \mid \text{I like to vacation in}) \approx p(\text{Maine} \mid \text{in})$$

- Or maybe:

$$p(\text{Maine} \mid \text{I like to vacation in}) \approx p(\text{Maine} \mid \text{vacation in})$$

- So basically, we assume we can ignore distant context.
- A **Markov assumption** holds that the future state of a system depends solely on its current state, regardless of its history.

Andrei Markov

Markov Assumption

- More generally:

$$p(w_1 w_2 \ldots w_n) \approx \prod_i p(w_i \mid w_{i-k} \ldots w_{i-1})$$

- So each component of the product is approximated as:

$$p(w_i \mid w_1 w_2 \ldots w_{i-1}) \approx p(w_i \mid w_{i-k} \ldots w_{i-1})$$

n-grams

Cats like to eat salmon.

Unigrams (1-grams):	*Bigrams (2-grams):*	*Trigrams (3-grams):*
Cats	(Cats, like)	(Cats, like, to)
like	(like, to)	(like, to, eat)
to	(to, eat)	(like, to, eat)
eat	(eat, salmon)	(eat, salmon, .)
salmon	(salmon, .)	
.		

A Unigram Language Model

- A **unigram** is an n-gram where $n = 1$.
  - (It's just a token probability.)

$$p(w_1 w_2 \ldots w_n) \approx \prod_i p(w_i)$$

- These probabilities come from training on a dataset D:

$$p(w_i) = \frac{C(w_i)}{|D|}$$

$C(w_i)$ — token frequency

$|D|$ — number of tokens in dataset

How can we generate text from this model?

A Bigram Language Model

$$p(w_1 w_2 \ldots w_n) \approx \prod_i p(w_i \mid w_{i-1})$$

- We can estimate bigram probabilities with **maximum likelihood estimation (MLE)**

$$p(w_i \mid w_{i-1}) = \frac{C(w_{i-1}, w_i)}{\sum_{w'} C(w_{i-1}, w')} \equiv \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}$$

Relative frequency

Example

$$p(w_i \mid w_{i-1}) \overset{\text{MLE}}{=} \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}$$

<s> I am Sam </s>

<s> Sam I am </s>

<s> Do you like green eggs and ham? </s>

$p(\text{I} \mid \text{<s>}) = \frac{1}{3} = 0.33$

$p(\text{Sam} \mid \text{<s>}) = ?$

$p(\text{eggs} \mid \text{like}) = ?$

$p(\text{</s>} \mid \text{Sam}) = \frac{1}{2} = 0.5$

$p(\text{Sam} \mid \text{am}) = ?$

$p(\text{Sam} \mid \text{I}) = ?$

Example

$$p(w_i \mid w_{i-1}) \overset{\text{MLE}}{=} \frac{C(w_{i-1}, w_i)}{C(w_{i-1})}$$

<s> I am Sam </s>

<s> Sam I am </s>

<s> Do you like green eggs and ham? </s>

$$p(\text{I} \mid \text{<s>}) = \frac{1}{3} = 0.33 \qquad p(\text{Sam} \mid \text{<s>}) = \frac{1}{3} = 0.33 \qquad p(\text{eggs} \mid \text{like}) = \frac{0}{1} = 0.0$$

$$p(\text{</s>} \mid \text{Sam}) = \frac{1}{2} = 0.5 \qquad p(\text{Sam} \mid \text{am}) = \frac{1}{2} = 0.5 \qquad p(\text{Sam} \mid \text{I}) = \frac{0}{2} = 0.0$$

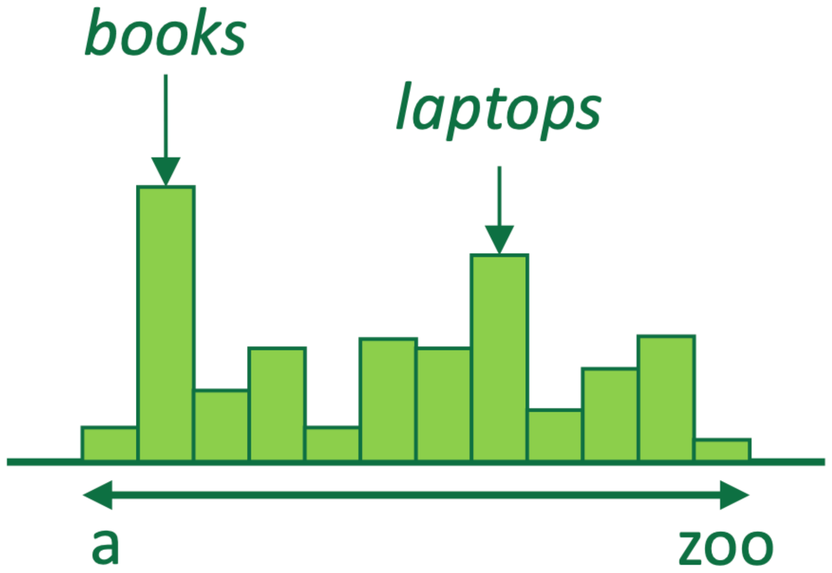
Generating from a Language Model

- Prefix: “students opened their”

- In **greedy decoding**, we always pick the token with the highest probability:

$$w_t = \arg\max_w p(w_t \mid w_{<t})$$

- In **sampling**, we randomly pick a token according to its probability



Generating Shakespeare

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.

These look better the higher our *n* is, so let's train higher-order *n*-gram models!

A Bigger Example

The Berkeley Restaurant Project corpus:

can you tell me about any good cantonese restaurants close by

tell me about chez panisse

when is caffe venezia open during the day

i'm looking for a good place to eat breakfast

Notice how *sparse* this distribution is.

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

We normalize row-wise to get bigram probabilities:

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

The Probability of a *Sequence of* n-grams

- We can estimate the probability of a sequence via a product of n-gram probabilities:

$$p(\text{<s> i want english food </s>}) =$$

$$p(\text{i} \mid \text{<s>}) \times p(\text{want} \mid \text{i}) \times p(\text{english} \mid \text{want})$$

$$\times p(\text{food} \mid \text{english}) \times p(\text{</s>} \mid \text{food})$$

$$= 0.25 \times 0.33 \times 0.0011 \times 0.5 \times 0.68 = 0.000031$$

- This probability is *very* close to 0—as are most sentence probabilities.
- These probabilities get tiny when we have longer inputs or rarer words.

Avoiding Numeric Underflow

- Most n-grams are *ridiculously* sparse, so probabilities will be extremely small. Underflow is a real risk.
- For numeric stability, we often work with log-probabilities instead.

$$\log \prod_{i=1}^{n} p(w_i \mid w_{i-1}) = \sum_{i=1}^{n} \log p(w_i \mid w_{i-1})$$

$p($<s> i want english food </s>$) =$

$$= 0.25 \times 0.33 \times 0.0011 \times 0.5 \times 0.68 = 0.000031$$

$$\log(0.25) + \log(0.33) + \log(0.0011) + \log(0.5) + \log(0.68) \approx -4.51$$

Perplexity

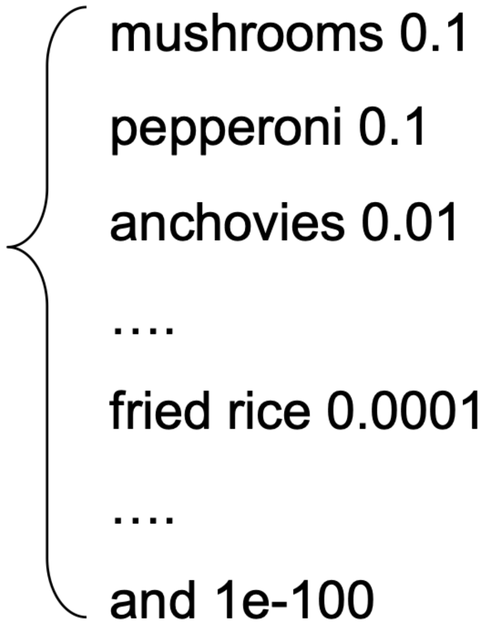
Intuition

- The Shannon Game: how well can we predict the next word?

I always order pizza with cheese and _____

The 5th president of the US was _____

I saw a _____



  - Unigrams are terrible at this.

- A better model of text is one that assigns a higher probability to the word that actually appeared.

Evaluating a Language Model

- *Intuition:* Does our language model prefer good sentences over bad sentences?
  - Good sentences are more likely to appear in a corpus
  - Ideally, an LM should assign higher probabilities to well-formed or frequently observed sentences

- We will train our language model on a **training set**.
- We will evaluate the model on a **test set**—data it hasn't seen during training.

Evaluating a Language Model

- A good language model should assign a high probability to a well-formed dataset that it hasn't seen before.
- Raw probabilities aren't great for this: they depend on sequence length.
- Thus, we should use a metric computed *as a function of* probability, but normalized by count.

- We will use a metric called **perplexity**.

Perplexity

- The perplexity of an LM on dataset D containing N tokens is:

$$\text{ppl}(D) = p(x_1, x_2, \ldots, x_N)^{-\frac{1}{N}}$$

$$\equiv \sqrt[N]{\frac{1}{p(x_1, x_2, \ldots, x_N)}}$$

Chain rule

$$\text{ppl}(D) = \sqrt[N]{\prod_i^N \frac{1}{p(w_i \mid w_1, \ldots, w_{i-1})}}$$

Markov assumption

$$\text{ppl}(D) = \sqrt[N]{\prod_i^N \frac{1}{p(w_i \mid w_{i-1})}}$$

- Probability is inversely related to perplexity. Thus, *lower perplexity is better.*

Perplexity as Weighted Average Branching Factor

- Branching factor: number of possible next words that can follow a given word
- Assume this is our vocabulary:

$$L = \{\text{red, green, blue}\}$$

- Let's define a probabilistic language where all tokens have $p = \frac{1}{3}$
- The perplexity of this dataset: "red red red red blue" is:

$$\left(\left(\frac{1}{3}\right)^5\right)^{-\frac{1}{5}} = \left(\frac{1}{3}\right)^{-1} = 3$$

Higher n → better models

- After training on 38M words from the Wall Street Journal (test set 1.5M words):

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

Higher n → better models?

- Note: better perplexity does *not* always mean we generate better sentences!
- Higher-order n-grams will generate more **fluent** sentences.
- Using maximum likelihood estimators, higher-order n-grams will always give you better likelihood on the **training set**, but not necessarily the **test set**.
    - E.g., you could overfit to the training set more easily with higher-order n-grams.

Practical Details

- We actually use *log-probabilities* to compute perplexity.

$$\text{ppl}(W) = \exp\left(-\frac{1}{N} \sum_{i}^{N} \log p(w_i \mid w_{<i}) \right)$$

- I.e., perplexity is the *exponentiated token-level negative log-likelihood*.

The Shakespeare Corpus, revisited

N=884,647 tokens **V**=29,066 types

Shakespeare produced 300,000 bigram types out of V^2=844 million possible bigrams.

99.96% of the possible bigrams never appeared.

It gets worse if we scale up to 4-grams: a model starts to regurgitate text from the Shakespeare corpus, with very little novel content!

Overfitting

- N-grams only work well if the test corpus looks a lot like the training corpus
  - In practice, it often doesn't

- How do we train robust models that generalize well to new data?
- One kind of generalization: novel n-grams!
  - N-grams that never occur in the training corpus, but occur in the test corpus

Train set	**Test set**
...denied the allegations	...denied the offer
...denied the requests	...denied the loan
...denied the claims	

What about novel n-grams?

- During evaluation, what is the probability of an n-gram that wasn't in the training corpus?

$$p(w_i \mid w_{i-1}) = \frac{C(w_{i-1}w_i)}{C(w_{i-1})}$$

What about novel n-grams?

- During evaluation, what is the probability of an n-gram that wasn't in the training corpus?

$$p(w_i \mid w_{i-1}) = \frac{C(w_{i-1}w_i)}{C(w_{i-1})} = \frac{0}{C(w_{i-1})}$$

- This isn't great: multiplying anything by this yields $p = 0$
  - ...which yields a perplexity of ∞!

- Novel n-grams are *very, very common*: language is infinitely creative!

Smoothing

- One solution: add a constant s.t. nothing truly has a probability of 0

$$p_{\text{Laplace}}(w_i \mid w_{i-1}) = \frac{C(w_{i-1}w_i) + 1}{C(w_{i-1}) + V}$$

- Adding a constant to the numerator and denominator is called **Laplace smoothing**
- If there's a big vocabulary, this can *massively* reduce the probability of seen sequences

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

$$p_{\text{Laplace}}(w_i \mid w_{i-1}) = \frac{C(w_{i-1}w_i) + 1}{C(w_{i-1}) + V}$$

	i	**want**	**to**	**eat**	**chinese**	**food**	**lunch**	**spend**
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

Interpolation

- Sometimes, it helps to use *less* context.

- A second solution: **interpolate** between LMs trained on lower-order n-grams

$$p(x_{k-n+1} : x_n) = \lambda_1 p(x_n) + \lambda_2 p(x_{n-1} x_n) + \lambda_3 p(x_{n-2} : x_n) + \ldots$$

Unigram LM Bigram LM Trigram LM

- Another way to do this is to just use the highest-order n-gram probability we can. If the count for that n-gram is 0, we recursively **backoff** to the (n-1)-gram.
- (Interpolation usually works better.)

Choosing Lambdas

$$p(x_{k-n+1} : x_n) = \lambda_1 p(x_n) + \lambda_2 p(x_{n-1} x_n) + \lambda_3 p(x_{n-2} : x_n) + \ldots$$

- Use a **held-out** development corpus to evaluate different lambda settings
  - Fix n-gram probabilities using the training data
  - Search for lambdas that give largest probability to held-out development set

Absolute Discounting

- The most performant n-gram language models are based on **Kneser-Ney smoothing.**
- This is a *recursive* backoff algorithm based on **absolute discounting**.
  - Subtract a fixed (absolute) discount d from each n-gram count

Backoff to (n-1)-gram

$$p_{\text{AD}}(w_i \mid w_{i-1}) = \frac{C(w_{i-1}w_i) - d}{\sum_{w'} C(w_{i-1}w')} + \lambda(w_{i-1})p(w_i)$$

Similar to smoothing

Interpolation weight

Bigram count in training set	Bigram count in heldout set
0	0.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

Kneser-Ney Discounting

Intuition

- My vision is bad, so I often need to use reading _____.
  - "San Francisco" is more frequent than "glasses", but "Francisco" *only* occurs after "San".
  - **Idea**: the unigram probability should not depend on the *frequency of* w, but on the *number of contexts in which* w *appears*

$$C_{\text{KN}}(\cdot w): \quad \text{Number of word types } w' \text{ which precede } w$$
$$C_{\text{KN}}(\cdot\cdot): \quad \sum_{w'} C_{\text{KN}}(\cdot w)$$

- **Kneser-Ney Smoothing:** use absolute discounting, but also use C_{KN}

Kneser-Ney Discounting

- $p_{\text{continuation}}(w)$: "How likely is w to appear as a novel continuation?"
  - For each word, count the number of bigrams it completes
  - Every bigram type was novel the first time it was seen

$$p_{\text{continuation}}(w) \propto |\{w_{i-1} : c(w_{i-1}w) > 0\}|$$

Kneser-Ney Discounting

- How many times does w appear as a novel continuation:

$$p_{\text{continuation}}(w) \propto |\{w_{i-1} : c(w_{i-1}w) > 0\}|$$

- Normalize by the number of word bigram types:

$$|\{(w_{j-1}, w_j) : c(w_{j-1}, w_j) > 0\}|$$

$$p_{\text{continuation}}(w) = \frac{|\{w_{i-1} : c(w_{i-1}w) > 0\}|}{|\{(w_{j-1}w_j) : c(w_{j-1}, w_j) > 0\}|}$$

- Example: “Francisco” occurring only after “San” will have low continuation probability

Interpolated Kneser-Ney Smoothing

KN bigram probability

unigram coefficient

$$p_{\text{KN}}(w_i \mid w_{i-1}) = \frac{\max(C(w_{i-1}w_i) - d, 0)}{C(w_{i-1})} + \lambda(w_{i-1}) p_{\text{continuation}}(w_i)$$

- λ is a normalization constant; it weights the probability we've discounted

$$\lambda(w_{i-1}) = \frac{d}{C(w_{i-1})} \mid \{w : C(w_{i-1}w) > 0\} \mid$$

normalized discount

number of types that follow w_{i-1}

Recursive Kneser-Ney Smoothing

$$p_{\text{KN}}(w_i \mid w_{i-n+1}^{i-1}) = \frac{\max(C_{\text{KN}}(w_{i-n+1}^{i}) - d, 0)}{C_{\text{KN}}(w_{i-n+1}^{i-1})} + \lambda(w_{i-n+1}^{i-1}) p_{\text{KN}}(w_i \mid w_{i-n+2}^{i-1})$$

$$C_{\text{KN}}(\cdot) = \begin{cases} \text{count}(\cdot) & \text{for highest order} \\ \text{continuation count}(\cdot) & \text{for lower order} \end{cases}$$

One last modification (**modified Kneser-Ney smoothing**): instead of a single discount d, we could use different discounts d_1, d_2, d_{3+} for n-grams with counts of 1, 2, or 3 or more, respectively. This is the best-performing version of Kneser-Ney smoothing.

A state-of-the-art n-gram language model!

- *Infini-gram*: trained on 5 trillion tokens!
- ∞−gram LM with backoff:

https://arxiv.org/pdf/2401.17377

https://huggingface.co/spaces/liujch1998/infini-gram