N-gram Language Models

Statistical Language Processing

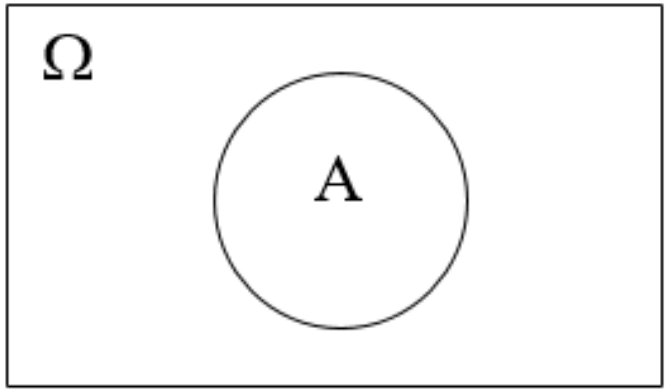
- In the solution of many problems in the natural language processing, **statistical language processing** techniques can be also used.
  - optical character recognition
  - spelling correction
  - speech recognition
  - machine translation
  - part of speech tagging
  - parsing
- Statistical techniques can be used to disambiguate the input.
- They can be used to select the most probable solution.
- Statistical techniques depend on the probability theory.
- To able to use statistical techniques, we will need corpora to collect statistics.
- Corpora should be big enough to capture the required knowledge.

Basic Probability

- **Probability Theory**: predicting how likely it is that something will happen.
- **Probabilities**: numbers between 0 and 1.
- **Probability Function**:
  - P(A) means that how likely the event A happens.
  - P(A) is a number between 0 and 1
  - P(A)=1 => a certain event
  - P(A)=0 => an impossible event
- **Example**: a coin is tossed three times. What is the probability of 3 heads?
  - 1/8
  - uniform distribution

Probability Spaces

- There is a sample space and the subsets of this sample space describe the events.
- Ω is a sample space.
  - Ω is the certain event
  - the empty set is the impossible event.



P(A) is between 0 and 1

P(Ω) = 1

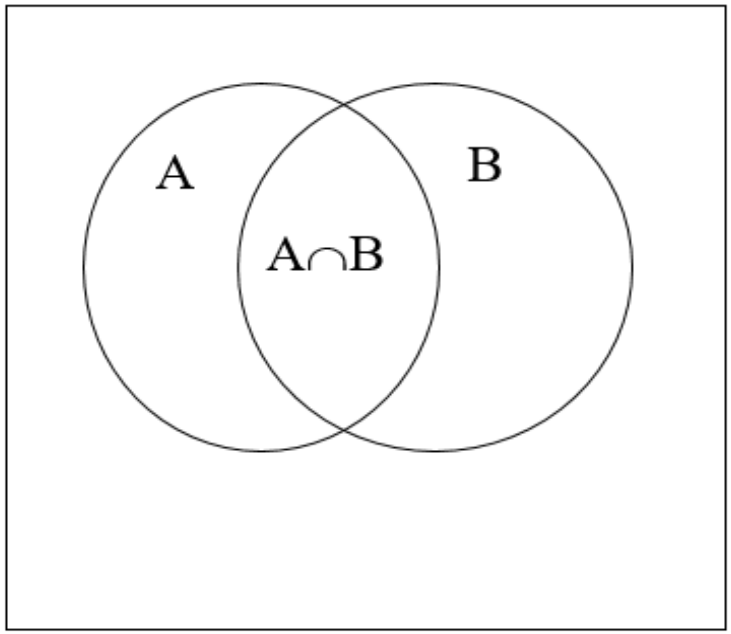
Unconditional and Conditional Probability

- **Unconditional Probability** or **Prior Probability**
  - P(A)
  - the probability of the event A does not depend on other events.

- **Conditional Probability -- Posterior Probability -- Likelihood**
  - P(A|B)
  - this is read as the probability of A given that we know B.

- Example:
  - P(put) is the probability of to see the word *put* in a text
  - P(on|put) is the probability of to see the word *on* after seeing the word *put*.

Unconditional and Conditional Probability



$$P(A|B) = P(A \cap B) / P(B)$$

$$P(B|A) = P(A \cap B) / P(A)$$

Bayes' Theorem

- Bayes' theorem is used to calculate P(A|B) from given P(B|A).
- We know that:

  $P(A \cap B) = P(A|B)\,P(B)$

  $P(A \cap B) = P(B|A)\,P(A)$

- So, we will have:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \qquad P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}$$

Language Model

- Models that assign probabilities to sequences of words are called **language models (LMs).**
- The simplest language model that assigns probabilities to sentences and sequences of words is the **n-gram**.
- An **n-gram** is a sequence of N words:
  - A 1-gram (unigram) is a single word sequence of words like “please” or “ turn”.
  - A 2-gram (bigram) is a two-word sequence of words like “please turn”, “turn your”, or ”your homework”.
  - A 3-gram (trigram) is a three-word sequence of words like “please turn your”, or “turn your homework”.
- We can use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire word sequences.

Probabilistic Language Models

- Probabilistic language models can be used to assign a probability to a sentence in many NLP tasks.
- Machine Translation:
  - P(high winds tonight) > P(large winds tonight)
- Spell Correction:
  - Thek office is about ten minutes from here
  - P(The Office is) > P(Then office is)
- Speech Recognition:
  - P(I saw a van) >> P(eyes awe of an)
- Summarization, question-answering, …

Probabilistic Language Models

- Our goal is to compute the probability of a sentence or sequence of words W (=w_1,w_2,…w_n):
  - P(W) = P(w_1,w_2,w_3,w_4,w_5…w_n)
- What is the probability of an upcoming word?:
  - P(w_5|w_1,w_2,w_3,w_4)
- A model that computes either of these:
  - P(W) or P(w_n|w_1,w_2…w_{n-1}) is called a **language model**.

Chain Rule of Probability

- How can we compute probabilities of entire word sequences like $w_1,w_2,\ldots w_n$?
  - The probability of the word sequence $w_1,w_2,\ldots w_n$ is $P(w_1,w_2,\ldots w_n)$.
- We can use the **chain rule of the probability** to decompose this probability:

$$\mathbf{P(w_1^n) = P(w_1)\,P(w_2|w_1)\,P(w_3|w_1^2)\ldots P(w_n|w_1^{n-1})}$$

$$= \prod_{k=1}^{n} P(w_k \mid w_1^{k-1})$$

Example:

P(the man from jupiter) =

P(the) P(man|the) P(from|the man) P(jupiter|the man from)

Chain Rule of Probability and Conditional Probabilities

- The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words.

- Definition of Conditional Probabilities:

$$\mathbf{P(B|A) = P(A,B) / P(A) \quad \rightarrow \quad P(A,B) = P(A)\, P(B|A)}$$

- Conditional Probabilities with More Variables:

$$P(A,B,C,D) = P(A)\, P(B|A)\, P(C|A,B)\, P(D|A,B,C)$$

- **Chain Rule:**

$$\mathbf{P(w_1 \ldots w_n) = P(w_1)\, P(w_2|w_1)\, P(w_3|w_1 w_2) \ldots P(w_n|w_1 \ldots w_{n-1})}$$

Computing Conditional Probabilities

- To compute the exact probability of a word given a long sequence of preceding words is difficult (sometimes impossible).
- We are trying to compute $\mathbf{P(w_n|w_1 \ldots w_{n-1})}$ which is the **probability of seeing $\mathbf{w_n}$ after seeing $\mathbf{w_1^{n-1}}$**.
- We may try to compute $\mathbf{P(w_n|w_1 \ldots w_{n-1})}$ exactly as follows:

$$\mathbf{P(w_n|w_1 \ldots w_{n-1}) = count(w_1 \ldots w_{n-1} w_n) / count(w_1 \ldots w_{n-1})}$$

- Too many possible sentences and we may never see enough data for estimating these probability values.
- So, we need to compute $\mathbf{P(w_n|w_1 \ldots w_{n-1})}$ approximately.

N-Grams

- The intuition of the n-gram model (simplifying assumption):
  - instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words.

$\mathbf{P(w_n \mid w_1 \ldots w_{n-1}) \approx P(w_n)}$	**unigram**
$\mathbf{P(w_n \mid w_1 \ldots w_{n-1}) \approx P(w_n \mid w_{n-1})}$	**bigram**
$\mathbf{P(w_n \mid w_1 \ldots w_{n-1}) \approx P(w_n \mid w_{n-1} w_{n-2})}$	**trigram**
$\mathbf{P(w_n \mid w_1 \ldots w_{n-1}) \approx P(w_n \mid w_{n-1} w_{n-2} w_{n-3})}$	**4-gram**
$\mathbf{P(w_n \mid w_1 \ldots w_{n-1}) \approx P(w_n \mid w_{n-1} w_{n-2} w_{n-3} w_{n-4})}$	**5-gram**

- In general, **N-Gram** is

$$\mathbf{P(w_n \mid w_1 \ldots w_{n-1}) \approx P\left(w_n \mid w_{n-N+1}^{n-1}\right)}$$

N-Grams

computing probabilities of word sequences

Unigrams -- $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k)$

Bigrams -- $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k \mid w_{k-1})$

Trigrams -- $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k \mid w_{k-1} w_{k-2})$

4-grams -- $P(w_1^n) \approx \prod_{k=1}^{n} P(w_k \mid w_{k-1} w_{k-2} w_{k-3})$

N-Grams

computing probabilities of word sequences (Sentences)

Unigram

P(<s> the man from jupiter came </s>) ≈
P(the) P(man) P(from) P(jupiter) P(came)

Bigram

P(<s> the man from jupiter came </s>) ≈
P(the|<*s*>) P(man|the) P(from|man) P(jupiter|from) P(came|jupiter) P(</s>|came)

Trigram

P(<s> the man from jupiter came </s>) ≈
P(the|<*s*> <*s*>) P(man|<*s*> the) P(from|the man) P(jupiter|man from)
P(came|from jupiter) P(</s>|jupiter came) P(</s>|came </s>)

N-gram models

- In general, a n-gram model is an insufficient model of a language because languages have **long-distance dependencies**.
  - "The **computer(s)** which I had just put into the machine room **is (are)** crashing."
  - But we can still effectively use N-Gram models to represent languages.
- Which N-Gram should be used as a language model?
  - Bigger N, the model will be more accurate.
    - But we may not get good estimates for N-Gram probabilities.
    - The N-Gram tables will be more sparse.
  - Smaller N, the model will be less accurate.
    - But we may get better estimates for N-Gram probabilities.
    - The N-Gram table will be less sparse.
  - In reality, we do not use higher than Trigram (not more than Bigram).
  - How big are N-Gram tables with 10,000 words?
    - Unigram -- 10,000
    - Bigram – 10,000*10,000 = 100,000,000
    - Trigram – 10,000*10,000*10,000 = 1,000,000,000,000

N-Grams and Markov Models

- The assumption that the probability of a word depends only on the previous word(s) is called **Markov assumption**.
- **Markov models** are the class of probabilistic models that assume that we can predict the probability of some future unit without looking too far into the past.
- A **bigram** is called a **first-order** Markov model (because it looks one token into the past);
- A **trigram** is called a **second-order** Markov model;
- In general a **N-Gram** is called a **N-1 order** Markov model.

Estimating N-Gram Probabilities

- Estimating n-gram probabilities is called **maximum likelihood estimation (**or **MLE).**
- We get the MLE estimate for the parameters of an n-gram model by ***getting counts from a corpus***, and **normalizing** the counts so that they lie between 0 and 1.

- Estimating bigram probabilities:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

where C is the count of that pattern in the corpus

- Estimating N-Gram probabilities

$$P(w_n \mid w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})}$$

Estimating N-Gram Probabilities

A Bigram Example

- ***A mini-corpus:*** We augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word, and special end-symbol </s>.

  <s> I am Sam </s>
  <s> Sam I am </s>
  <s> I fly </s>

- ***Unique words:*** I, am, Sam, fly

- ***Bigrams:*** <s> and </s> are also tokens. There are 6(4+2) tokens and 6*6=36 bigrams

P(I|<s>)=2/3	P(Sam|<s>)=1/3	P(am|<s>)=0	P(fly|<s>)=0	P(<s>|<s>)=0	P(</s>|<s>)=0
P(I|I)=0	P(Sam|I)=0	P(am|I)=2/3	P(fly|I)=1/3	P(<s>|I)=0	P(</s>|I)=0
P(I|am)=0	P(Sam|am)=1/2	P(am|am)=0	P(fly|am)=0	P(<s>|am)=0	P(</s>|am)=1/2
P(I|Sam)=1/2	P(Sam|Sam)=0	P(am|Sam)=0	P(fly|Sam)=0	P(<s>|Sam)=0	P(</s>|Sam)=1/2
P(I|fly)=0	P(Sam|fly)=0	P(am|fly)=0	P(fly|fly)=0	P(<s>|fly)=0	P(</s>|fly)=1
P(I|</s>)=0	P(Sam|</s>)=1/3	P(am|</s>)=1/3	P(fly|</s>)=1/3	P(<s>|</s>)=0	P(</s>|</s>)=0

Estimating N-Gram Probabilities

Example

<s> I am Sam </s>
<s> Sam I am </s>
<s> I fly </s>

- **Unigrams:** I, am, Sam, fly

P(I)=3/8 P(am)=2/8 P(Sam)=2/8 P(fly)=1/8

- **Trigrams:** There are 6*6*6=216 trigrams.
  - Assume there are two tokens <s> <s> at the begining, and two tokens </s> </s> at the end.

P(I|<s> <s>)=2/3 P(Sam|<s> <s>)=1/3
P(am|<s> I)=1/2 P(fly|<s> I)=1/2
P(I|<s> Sam)=1
P(Sam|I am)=1/2 P(</s>|I am)=1/2
P(</s>|am Sam)=1
P(</s>|Sam </s>)=1

Estimating N-Gram Probabilities

Corpus: Berkeley Restaurant Project Sentences

- There are 9222 sentences in the corpus.
- Raw biagram counts of 8 words (out of 1446 words)

	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

Estimating N-Gram Probabilities

Corpus: Berkeley Restaurant Project Sentences

	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

- Unigram counts:

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

- Normalize bigrams by unigram counts:

	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

Bigram Estimates of Sentence Probabilities

- Some other bigrams:

  P(i|<s>)=0.25 P(english|want)=0.0011

  P(food|english)=0.5 P(</s>|food)=0.68

	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

- Compute the probability of sentence **I want English food**

  P(<s> i want english food </s>)

  = P(i|<s>) P(want|i) P(english|want) P(food|english) P(</s>|food)

  = 0.25*0.33*0.0011*0.5*0.68

  = 0.000031

Log Probabilities

- Since probabilities are less than or equal to 1, the more probabilities we multiply together, the ***smaller the product becomes***.
  - Multiplying enough n-grams together would result in **numerical underflow**.
- By using **log probabilities** instead of ***raw probabilities***, we get numbers that are not as small.
  - Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them.
    - adding is faster than multiplying
  - The result of doing all computation and storage in log space is that we only need to convert back into probabilities if we need to report them at the end

$$\log(p_1 \cdot p_2 \cdot p_3 \cdot p_4) = \log p_1 + \log p_2 + \log p_3 + \log p_4$$

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4)$$

Evaluating Language Models

- Does our language model prefer good sentences to bad ones?
  - Assign higher probability to “real” or “frequently observed” sentences than “ungrammatical” or “rarely observed” sentences?
- We train parameters of our model on a **training set**.
- We test the model’s performance on data we haven’t seen.
  - A **test set** is an unseen dataset that is different from our training set, totally unused.
- An **evaluation metric** tells us how well our model does on the *test set*.

Evaluating Language Models

Extrinsic Evaluation

- **Extrinsic Evaluation** of a N-gram *language model* is to use it in an application and measure how much the application improves.
- To compare two language models A and B:
  - Use each of language model in a task such as spelling corrector, MT system.
  - Get an accuracy for A and for B
    - How many misspelled words corrected properly
    - How many words translated correctly
  - Compare accuracy for A and B
    - The model produces the better accuracy is the better model.
- Extrinsic evaluation can be time-consuming.

Evaluating Language Models
Intrinsic Evaluation

- An **intrinsic evaluation** metric is one that measures the quality of a model independent of any application.

- When a corpus of text is given and to compare two different n-gram models,
  - Divide the data into ***training*** and ***test sets***,
  - Train the parameters of both models on the *training set*, and
  - Compare how well the two trained models fit the test set.
    - Whichever model assigns a higher probability to the test set

- In practice, *probability-based metric* called **perplexity** is used instead of raw probability as our metric for evaluating language models.

Evaluating Language Models
Perplexity

- The ***best language model*** is one that best predicts an unseen test set
  - Gives the highest P(testset)
- The **perplexity** of a language model on a test set is the *inverse probability of the test set*, normalized by the number of words.
- **Minimizing perplexity is the same as maximizing probability**
- The **perpelexity PP** for a test set $W=w_1w_2\ldots w_n$ is

$$PP(W) = \sqrt[N]{\frac{1}{P(w_1w_2\ldots w_N)}} = \sqrt[N]{\prod_{i=1}^{N}\frac{1}{P(w_i|w_1\ldots w_{i-1})}}$$ by chain rule

- The **perpelexity PP** for bigrams:

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

Evaluating Language Models

Perplexity as branching factor

- Perplexity can be seen as the weighted average *branching factor of a language*.
  - The branching factor of a language is the number of possible next words that can follow any word.
- Let's suppose a sentence consisting of random digits
- What is the perplexity of this sentence according to a model that assign P=1/10 to each digit?

$$
\begin{aligned}
\mathrm{PP}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\
&= \left(\frac{1}{10}^{N}\right)^{-\frac{1}{N}} \\
&= \frac{1}{10}^{-1} \\
&= 10
\end{aligned}
$$

Evaluating Language Models

Perplexity

- Lower perplexity = better model
- Training 38 million words, test 1.5 million words, WSJ

	Unigram	Bigram	Trigram
Perplexity	962	170	109

- An intrinsic improvement in perplexity does not guarantee an (extrinsic) improvement in the performance of a language processing task like speech recognition or machine translation.
  - Nonetheless, because perplexity often correlates with such improvements, it is commonly used as a quick check on an algorithm.
  - But a model's improvement in perplexity should always be confirmed by an end-to-end evaluation of a real task before concluding the evaluation of the model.

Generalization and Zeros

- The **n-gram model**, like many statistical models, is ***dependent on the training corpus***.
  - the probabilities often encode specific facts about a given training corpus.
- 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: Getting rid of **Zeros**!
    - Things that don't ever occur in the training set, but occur in the test set
- **Zeros**: things that don't ever occur in the training set but do occur in the test set causes problem for two reasons.
  - First, underestimating the probability of all sorts of words that might occur,
  - Second, if probability of any word in test set is 0, entire probability of test set is 0.

Unknown Words

- We have to deal with words we haven't seen before, which we call **unknown words**.
- We can model these potential unknown words in the test set by adding a ***pseudo-word called <UNK>*** into our training set too.

- One way to handle unknown words is:
  - Replace words in the training data by <UNK> based on their frequency.
    - For example we can replace by <UNK> all words that occur fewer than n times in the training set, where n is some small number, or
    - Equivalently select a vocabulary size V in advance (say 50,000) and choose the top V words by frequency and replace the rest by <UNK>.
  - Proceed to train the language model as before, treating <UNK> like a regular word.

Smoothing

- To keep a language model from assigning zero probability to these unseen events, we'll have to ***shave off a bit of probability mass from some more frequent events and give it to the events we've never seen***.
- This modification is called **smoothing** (or **discounting**).
- There are many ways to do **smoothing**, and some of them are:
  - **Add-1 smoothing (Laplace Smoothing)**
  - **Add-k smoothing,**
  - **Backoff**
  - **Kneser-Ney smoothing.**

Laplace Smoothing

- The simplest way to do smoothing is to **add one to all the counts**, before we normalize them into probabilities.
  - All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on.
- This algorithm is called **Laplace smoothing (Add-1 Smoothing)**.
- We pretend that we saw each word one more time than we did, and we just add one to all the counts!,

Laplace Smoothing
(Add-1 Smoothing)

Laplace Smoothing for Unigrams:

- The unsmoothed maximum likelihood estimate of the unigram probability of the word w_i is its count c_i normalized by the total number of word tokens N:

$$P(w_i) = c_i / N$$

- **Laplace smoothing** adds one to each count. Since there are V words in the vocabulary and each one was incremented, we also need to adjust the denominator to take into account the extra V observations.

$$P_{Laplace}(w_i) = (c_i + 1) / (N + V)$$

Discounting

- It is convenient to describe how a smoothing algorithm affects the numerator, by defining an **adjusted count c***.
  - This adjusted count is easier to compare directly with the MLE counts and can be turned into a probability like an MLE count by normalizing by N.
- The **adjusted count** c_i^* of word w_i is:

$$c_i^*(w_i) = (c_i + 1)\frac{N}{N+V}$$

$$P_i^*(w_i) = c_i^*(w_i) / N = (c_i + 1) / (N + V) = P_{Laplace}(w_i)$$

- A related way to view smoothing is as **discounting** (lowering) some non-zero counts in order to get the probability mass that will be assigned to the zero counts.
  - Thus, instead of referring to the discounted counts c*, we might describe smoothing in terms of a **relative discount d^c**, the ratio of the discounted counts to the original counts:

$$\mathbf{d^c = c^* / c}$$

Laplace Smoothing for Bigrams

- The **normal bigram probabilities** are computed by normalizing each bigram counts by the unigram count:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

- **Add-one smoothed bigram probabilities:**

$$P^*_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{\sum_w (C(w_{n-1}w)+1)} = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V}$$

- **Adjusted counts** can be computed:

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V}$$

Laplace-smoothed Bigrams

Corpus: Berkeley Restaurant Project Sentences

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

$$P^*(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V}$$

	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

	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

Laplace-smoothed Bigrams

Corpus: Berkeley Restaurant Project Sentences: Adjusted counts

$$C(w_{n-1}w_n)$$

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V}$$

	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

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Add-k Smoothing

- Add-one smoothing has made a very big change to the counts.
  - **C(want to)** changed from 608 to 238!
  - **P(to|want)** decreases from .66 in the unsmoothed case to .26 in the smoothed case.
  - Looking at discount d shows us how counts for each prefix word have been reduced;
    - discount for bigram **want to** is .39, while discount for **Chinese food** is .10, a factor of 10
- The sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros.
- One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events.
- Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?).
- This algorithm is called **add-k smoothing**.

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + k}{C(w_{n-1}) + kV}$$

Smoothing: Backoff and Interpolation

- Sometimes it helps to use less context

- In **backoff**, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram.
- In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram.

- In **interpolation**, we always mix the probability estimates from all the n-gram estimators, weighing and combining the trigram, bigram, and unigram counts.

Smoothing: Backoff and Interpolation

- In **simple linear interpolation**, we combine different order n-grams by linearly interpolating all the models.

$$\hat{P}(w_n|w_{n-1}w_{n-2}) = \lambda_1 P(w_n|w_{n-1}w_{n-2}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n)$$

$$\sum_i \lambda_i = 1$$

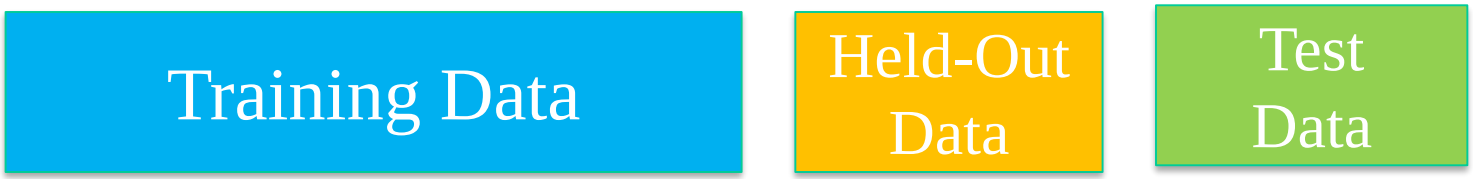
- In a slightly more sophisticated version of linear interpolation, each λ weight is computed by conditioning on the context.
- **Interpolation with context-conditioned weights:**

$$\hat{P}(w_n|w_{n-2}w_{n-1}) = \lambda_1(w_{n-2}^{n-1})P(w_n|w_{n-2}w_{n-1}) + \lambda_2(w_{n-2}^{n-1})P(w_n|w_{n-1}) + \lambda_3(w_{n-2}^{n-1})P(w_n)$$

Smoothing: Backoff and Interpolation

How to set the lambdas?

- Use a **held-out** corpus



- Choose λs to maximize the probability of held-out data:
  - Fix the N-gram probabilities (on the training data)
  - Then search for λs that give highest probability to held-out set:
  - There are various ways to find this optimal set of λs.

Kneser-Ney Smoothing

- One of the most commonly used and best performing n-gram smoothing methods is the interpolated **Kneser-Ney** algorithm.
- Kneser-Ney has its roots in a method called **absolute discounting**.

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

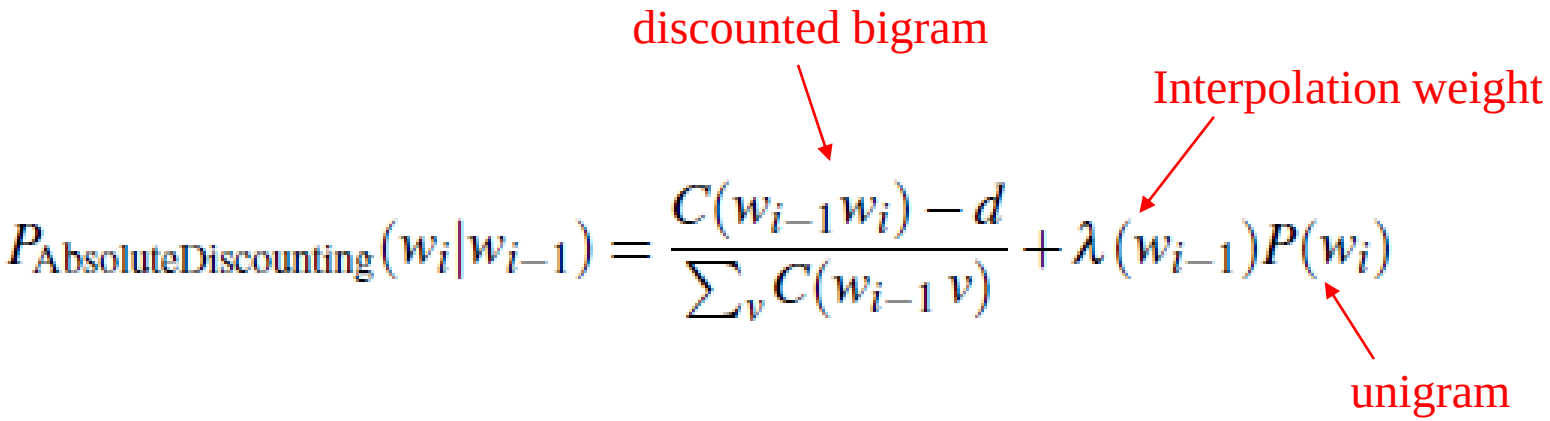
For all bigrams in 22 million words of count 0, 1, 2,...,9, the counts of these bigrams in a held-out corpus also of 22 million words.

- Except for the held-out counts for 0 and 1, all the other bigram counts in the held-out set could be estimated pretty well by just subtracting 0.75 from the count in the training set!
- **Absolute discounting** formalizes this intuition by subtracting a fixed (absolute) discount d from each count.

Kneser-Ney Smoothing

absolute discounting

Interpolated absolute discounting applied to bigrams:



- We could just set all the d values to .75, or we could keep a separate discount value of 0.5 for the bigrams with counts of 1.

Kneser-Ney Smoothing

Kneser-Ney discounting

- **Kneser-Ney discounting** augments absolute discounting with a more sophisticated way to handle the lower-order unigram distribution.
- Instead of P(w) : "How likely is w"
- $P_{CONTINUATION}$(w) : "How likely is w to appear as a novel continuation?
  - For each word, count the number of bigram types it completes
  - Every bigram type was a novel continuation the first time it was seen
  - How many times does w appear as a novel continuation:

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}|$$

  - To turn this count into a probability, we normalize by the ***total number of word bigram types***.

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{|\{(u', w') : C(u'w') > 0\}|}$$

Kneser-Ney Smoothing

Kneser-Ney discounting

- ***Alternative metaphor:*** The number of # of word types seen to precede w

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}|$$

- normalized by the # of words preceding all words:

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{\sum_{w'} |\{v : C(vw') > 0\}|}$$

- A frequent word (**Francisco**) occurring in only one context (**San**) will have a low continuation probability.

Kneser-Ney Smoothing

Kneser-Ney discounting

Interpolated Kneser-Ney smoothing for bigrams is:

$$P_{\text{KN}}(w_i|w_{i-1}) = \frac{\max(C(w_{i-1}w_i) - d, 0)}{C(w_{i-1})} + \lambda(w_{i-1})P_{\text{CONTINUATION}}(w_i)$$

λ is a normalizing constant; the probability mass we've discounted

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

the normalized discount

The number of word types that can follow w_{i-1}
= # of word types we discounted
= # of times we applied normalized discount

Huge Language Models

- By using text from the web, it is possible to build extremely large language models.
  - Google created a very large set of n-gram counts from a corpus containing more than 1 trillion words.
- Efficiency considerations are important when building language models that use such large sets of n-grams.
  - N-grams can also be shrunk by pruning, for example only storing n-grams with counts greater than some threshold (Google used threshold of 40)
- Although we can build web-scale language models using Kneser-Ney smoothing, a simpler smoothing algorithm may be sufficient with very large language models.

N-gram Smoothing Summary

- **Smoothing** algorithms provide a more sophisticated way to estimate the probability of n-grams.
- **Add-1 smoothing** (or Add-k smoothing) is okay for some NLP tasks such as text categorization, it is not so good for language modeling.
- Both **backoff** and **interpolation** are commonly used and they require discounting to create a probability distribution.
- The most commonly used method is the **Interpolated Kneser-Ney**
  - Kneser-Ney smoothing makes use of the probability of a word being a novel continuation.
  - The interpolated Kneser-Ney smoothing algorithm mixes a discounted probability with a lower-order continuation probability.