Queen Mary
University of London

ECS763 Natural Language Processing

Unit 3: Language Models

Lecturer: Julian Hough
School of Electronic Engineering and Computer Science

OUTLINE

1) Language Models: motivation
2) Language Models: ngram models
3) Language Models: evaluation
4) Smoothing

OUTLINE

1) Language Models: motivation
2) Language Models: ngram models
3) Language Models: evaluation
4) Smoothing

Sequence Modelling Tasks

- We considered classification tasks last week (e.g. sentiment analysis) $d \rightarrow c$
- Many problems are about modelling (labelling, characterising, evaluating) **sequences**:
  - Part-of-speech tagging
  - Dialogue act tagging
  - Named entity recognition
  - Speech recognition
  - Spelling correction
  - Machine translation
  - ...

Sequence Likelihood Tasks

- Speech recognition

  `I saw a van`
  `eyes awe of an`
- Spelling correction

  `It's about fifteen minuets from my house`
  `It's about fifteen minutes from my house`
- Machine translation

  `vjetar će biti noćas jak:`
  `the wind tonight will be strong`
  `the wind tonight will be powerful`
  `the wind tonight will be a yak`

Language Models

- To do effective sequence prediction we want to know the likelihood of different sequences (of words).
- Language models are designed to do this and are machines which play the Shannon Game (1951), reframing the challenge as:
  - How well can we **predict the next word given the history of previous words**?

I always order pizza with cheese and ____

The 33^{rd} President of the US was ____

I saw a ____

mushrooms 0.1

pepperoni 0.1

anchovies 0.01

….

fried rice 0.0001

….

and 1e-100

OUTLINE

1) Language Models: motivation
2) Language Models: ngram models
3) Language Models: evaluation
4) Smoothing

What is a Language Model?

- Answering the following questions would be useful for assigning probabilities to sequences:
  - **What is the probability of observed sequence O?**
    $p(O) = p(o_1,o_2,o_3,\ldots o_n)$
  - **Given observed sequence O = $o_1 \ldots o_{n-1}$, what is the probability of observing symbol o_n next?**
    $p(o_n|o_1,o_2,o_3,\ldots o_{n-1})$

- i.e. What is p("john likes mary") or p("john likes") or p("mary"| "john likes")?
- A model which computes these is a **language model.**

What is a Language Model?

- A language model estimates the probability function *p*:

$$p(w_i \mid w_1, w_2, w_3 \ldots w_{i-1})$$

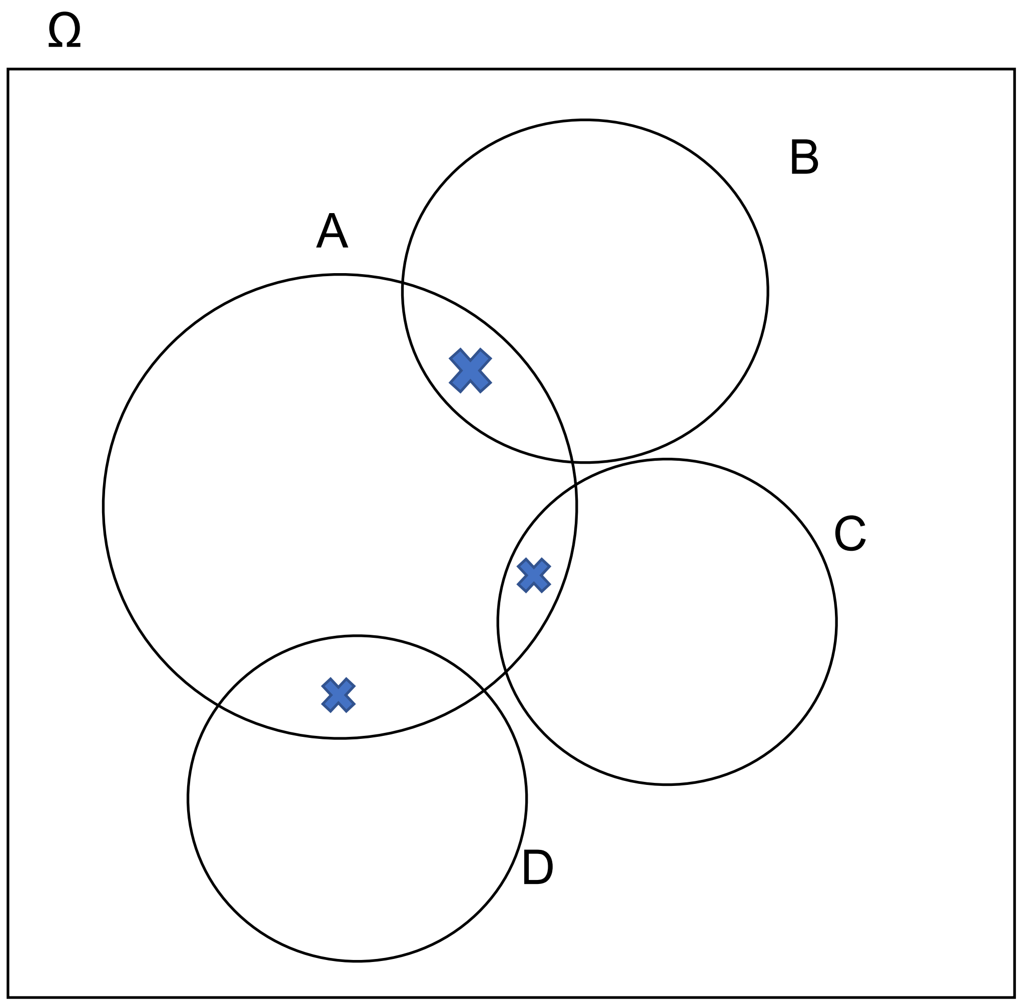
History/context

Current word

- For each context it gives a discrete probability distribution over all words in the vocabulary for w_i.

- It assigns a probability value for a given word observed at position w_i given the context observed at $w_1 \ldots w_{i-1}$

Remember: Discrete Probability Distributions

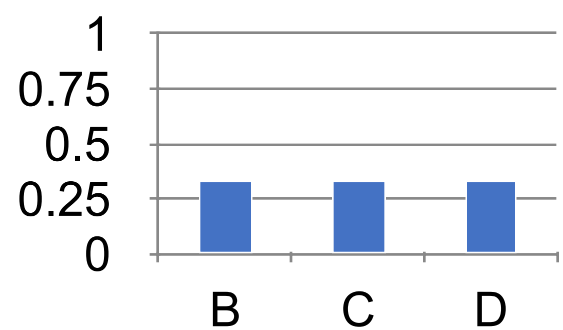


$$p\left(X = ?\mid A\right)$$

$$p\left(B \mid A\right) = \frac{\left|A \cap B\right|}{\left|A\right|} = \frac{1}{3}$$

$$p\left(C \mid A\right) = \frac{\left|A \cap C\right|}{\left|A\right|} = \frac{1}{3}$$

$$p\left(D \mid A\right) = \frac{\left|A \cap D\right|}{\left|A\right|} = \frac{1}{3}$$



What is a Language Model?

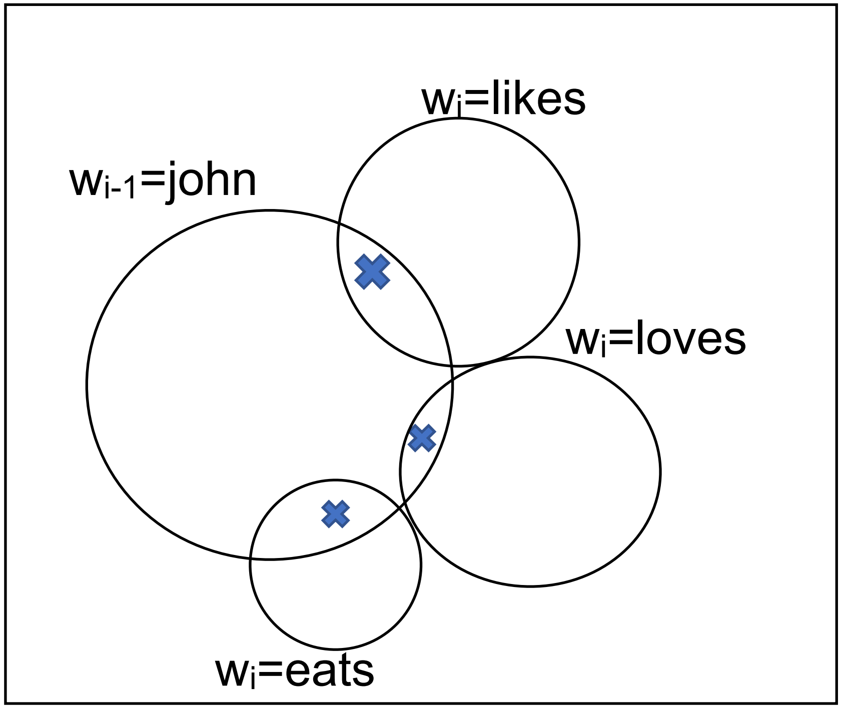
- The probability of the next word being a given value, (e.g. 'loves') independent of the previous words is the ***unigram*** probability. In event terms:

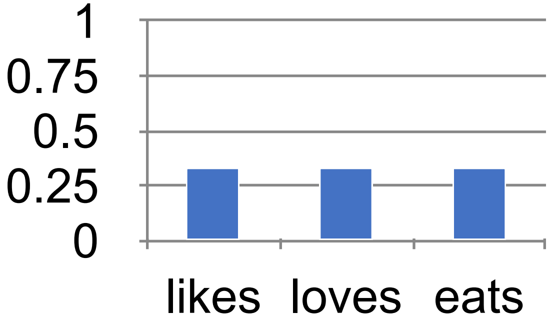
$$p\left(w_i = loves\right) = \frac{\left|w_i = loves\right|}{\sum_{x \in vocab}\left|w_i = x\right|}$$

- This is a bag-of-words model and doesn't consider the word order.
- Using the probability of the next word given the previous one i.e. the conditional probability $p(w_i|w_{i-1})$ (e.g. for 'john loves') is the ***bigram*** probability. In event terms:

$$p(w_i = loves \,|\, w_{i-1} = john) = \frac{|\, w_{i-1} = john \cap w_i = loves \,|}{|\, w_{i-1} = john \,|}$$

What is a Language Model?





$$p(w_i = likes \,|\, w_{i-1} = john) = \frac{|\, w_{i-1} = john \cap w_i = likes \,|}{w_{i-1} = john \,|} = \frac{1}{3}$$

$$p(w_i = loves \,|\, w_{i-1} = john) = \frac{|\, w_{i-1} = john \cap w_i = loves \,|}{w_{i-1} = john \,|} = \frac{1}{3}$$

$$p(w_i = eats \,|\, w_{i-1} = john) = \frac{|\, w_{i-1} = john \cap w_i = eats \,|}{w_{i-1} = john \,|} = \frac{1}{3}$$

The Chain Rule

- In ngram models, how do we assign probabilities to an entire sequence of words, or the probability of a word given the words so far?
- We can address both via the **Chain Rule (for probability)**
- Recall the definition of conditional probabilities (through the **product rule**)

  Rewriting: $\mathbf{P(A,B) = P(A)P(B|A)}$

- More than two variables, apply the product rule over and over again, i.e. the Chain Rule in general:

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

- The Chain Rule in a sequence model for a sequence of length n:

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

Using the chain rule

- How do we estimate probabilities? E.g. for the sentence 'Its water is so transparent'
- Count and divide:

$$p(its\ water\ is\ so\ transparent) = p(transparent \mid its\ water\ is\ so) = \frac{C(its\ water\ is\ so\ transparent)}{C(its\ water\ is\ so)}$$

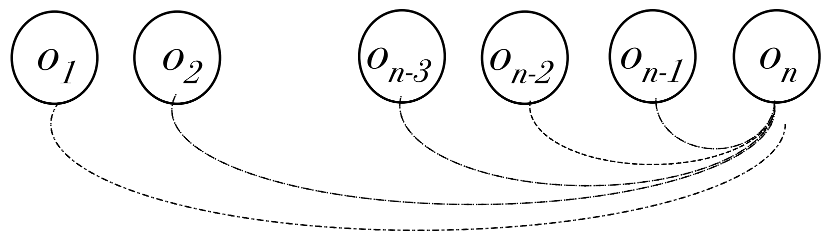
- According to the chain rule:

$$
\begin{aligned}
p(\text{“its water is so transparent”}) = \\
& p(its) \times \\
& p(water|its) \times \\
& p(is|its\ water) \times \\
& p(so|its\ water\ is) \times \\
& p(transparent|its\ water\ is\ so)
\end{aligned}
$$

- However, we'll never see enough data, so use the **Markov Assumption-** probability of next word only **depends on a fixed number of words back**
- E.g. a bigram model only depends on previous word, a trigram model depends on the previous two words only.

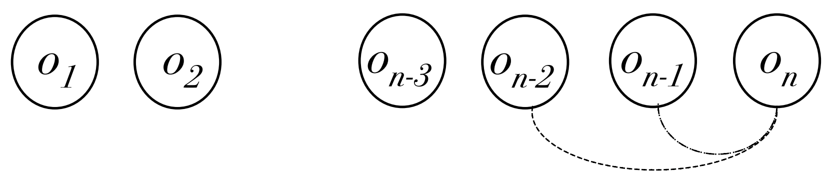
Markov Assumption

- Instead of:



- We approximate by:
  - “n-gram model of length k” (where k = n-1)

trigram model
(k=2):



- In general not sufficient – but often good approximation for high *k*.
  - Ignores long-distance dependencies:
    - “the computer I just put into the machine room on the fifth floor crashed”

Language Models

- This can go up to any arbitrary length (or '**order**'), e.g. unigram, bigram, trigram, 4-gram….7-gram… etc.
- In general **n-gram models** (Shannon ,1948).

- Unigram *its water is so transparent*

- Bigram *its water is so transparent*

- Trigram *its water is so transparent*

- 4-gram etc.

Language Models

- This can go up to any arbitrary length (or '**order**'), e.g. unigram, bigram, trigram, 4-gram….7-gram… etc.
- In general **n-gram models** (Shannon ,1948).

- Unigram *its water is so transparent*

- Bigram *its water is so transparent*

- Trigram *its water is so transparent*

- 4-gram etc.

Language Models

- General method when processing sequences is to extract the relevant n-grams (word sequences) according to the value of *n.*
- In training count the frequency of the ngrams occurring in the training data and store the counts.
- In testing use those counts to get probabilities of sequences of unseen data.
- Deriving the probabilities can be done with a variety of methods, called **n-gram language models.**
- Watch out for terminology: ’n-gram’ is used to mean either the word sequence itself or the predictive model that assigns it a probability.

Language Models

- **Unigram model:** after training a Maximum Likelihood Estimation (MLE) model from counting function *C* from a corpus:

Frequency of
the word (unigram)

$$p(w_i) = \frac{C(w_i)}{\sum_{w \in Vocab} C(w)}$$

The summed frequencies
of all the words in the vocab-
i.e. the length
of the training data

Language Models

- **Bigram model:** After training a Maximum Likelihood Estimation (MLE) bigram model from counting function *C* from a corpus:

Frequency of
n-gram (bigram)

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

Frequency of context
(previous word
from position i)

Language Models

- **General n-gram model:** After training a Maximum Likelihood Estimation (MLE) n-gram model from counting function *C* from a corpus:

Frequency of n-gram

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

Frequency of context (previous n-1 words from word in position i)

Language Models

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

- Example corpus

(note beginning (<s>) and end-of-sentence (</s>) markers):

<s> I am Sam </s>

<s> Sam I am </s>

<s> I do not like green eggs and ham </s>

- **<u>Exercise: what is the MLE bigram estimate for:</u>**

p(I|<s>) = 　　p(Sam|<s>) = 　　p(am|I) =

p(</s>|Sam) = 　　p(Sam|am) = 　　p(do|I) =

Language Models

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

- Example corpus

(note beginning (<s>) and end-of-sentence (</s>) markers):

<s> I am Sam </s>

<s> Sam I am </s>

<s> I do not like green eggs and ham </s>

- **Exercise: what is the MLE bigram estimate for:**

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

p(</s>|Sam) = 1/2 p(Sam|am) = 1/2 p(do|I) = 1/3

Language Models

- (Real corpus) Berkeley Restaurant Project sentences:

    - can you tell me about any good cantonese restaurants close by
    - mid priced thai food is what i’m looking for
    - tell me about chez panisse
    - can you give me a listing of the kinds of food that are available
    - i’m looking for a good place to eat breakfast
    - when is caffe venezia open during the day

Language Models

- Bigram counts from 9222 sentences

Word i (the bigram is Word i-1,Word i)

Word i-1
(context)

	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

Language Models

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

- Bigram MLE estimates:

- Normalize by unigram counts (which are the w_{i-1} counts):

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

- Result:

	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

Language Models

- Bigram MLE estimates (example knowledge of the model after counts):

    - p(english|want) = .0011
    - p(chinese|want) = .0065
    - p(to|want) = .66
    - p(eat|to) = .28
    - p(food|to) = 0
    - p(want|spend) = 0
    - p(i|<s>) = .25

Language Models

- Bigram MLE probability estimates of **sentences/ sequences**: use multiplication of probabilities of continuous n-grams assuming **independence** of bigram probabilities:

p(<s> I want english food </s>) =

p(I|<s>)

× p(want|I)

× p(english|want)

× p(food|english)

× p(</s>|food)

= .000031

Language Models

- Practical reality: we do everything in **log** space
  - Avoids underflow on a digital computer.
  - Adding is faster than multiplying.

- Unigram:

$$log(p(w_1) \times p(w_2) \times p(w_3)) = log(p(w_1)) + log(p(w_2)) + log(p(w_3))$$

- Bigram:

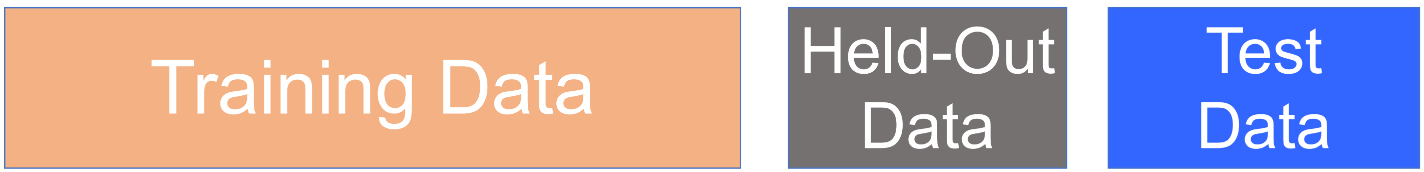
$$log(p(w_1 | w_0) \times p(w_2 | w_1) \times p(w_3 | w_2)) = log(p(w_1 | w_0)) + log(p(w_2 | w_1)) + log(p(w_3 | w_2))$$

OUTLINE

1) Language Models: motivation
2) Language Models: ngram models
3) Language Models: evaluation
4) Smoothing

How do we evaluate a LM?

- Gather a corpus.
- Divide it into 3 standard sections:



- Gather all the counts/estimations from the training data
- Iteratively develop by assigning probability to the heldout (not the test!) data.
- Experiment with value of *n* and other parameters like *discounts* (more later).
- Get the **Perplexity** score on the Test data (measure of how confused the model is by the unseen corpus).

How do we evaluate a LM?

- Though, don’t forget the preprocessing first!
    - Tokenizing raw text.
    - Spelling normalization (including capitalization).
    - Removal of punctuation (though possibly not all!)
- What about words not in the training data but which appear at testing time (remember language is Zipfian!), i.e. are not in the vocabulary?
    - These could give a zero and mess up the model/perplexity measures!
- How do we estimate how many **unknown or ‘out of vocabulary’ (OOV)** words we’re likely to encounter at testing?

How do we evaluate a LM?

- As it's a sequence modelling task, unlike bag-of-words based classification, **we shouldn't just remove OOV/unknown words, leads to ungrammatical sequences.**

- Several approaches to OOV words:

  - 1. Define the vocab by stipulating a **minimum document frequency** for words in the training data (e.g. 2). Any words appearing less than that, replace with an unknown word token **<unk/>** in the training data.

  - 2. Define the vocab by setting some **heldout data** aside- any words appearing in that which are not in the main training data are defined as OOV- replace their occurrences with <unk/> in the training data.

- On **test data,** always replace all OOV words with <unk/>.

- **Warning-** for a fair comparison of different models' perplexities, **the same vocab must be used!**

1. OOV words with minimum doc frequency

Training Data- pass 1

John likes Mary
John adores Mary
John adores Bill

Get counts only for the vocab selection.

Vocab counts: John: 3, likes:1, adores: 2, Mary: 2, Bill: 1

Min. Doc. freq = 2,
Vocab = {John, adores, Mary}

Training Data pass 2

John likes Mary
John adores Mary
John adores Bill

Replace OOV words with <unk/>, then get counts for the language model.

John **<unk/>** Mary
John adores Mary
John adores **<unk/>**

Counts: John: 3, adores: 2, Mary: 2, <unk/>: 2

Test Data

John despises Mary
Billl adores John

John **<unk/>** Mary
<unk/> adores John

2. OOV words from heldout training data

Training Data

John likes Mary
John adores Mary
John adores Bill

Do the counts for all words without replacement and define vocab as all words observed in this data.

Counts: John: 3, likes:1, adores: 2, Mary: 2, Bill: 1

Vocab = {John, likes, Mary, adores, Bill}

Held-Out Training Data

John hates Mary
Billl adores Mary

Replace OOV words with <unk/>, then keep adding to the model counts

John **<unk/>** Mary
Bill adores Mary

Counts: John: 4, likes:1, adores: 3, Mary: 4, Bill: 2, <unk/> : 1

Test Data

John despises Mary
Billl adores John

John **<unk/>** Mary
Bill adores John

Perplexity

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

    I always order pizza with cheese and ____

    The 33rd President of the US was ____

    I saw a ____

  - Unigrams are terrible at this game. (Why?)

mushrooms 0.1

pepperoni 0.1

anchovies 0.01

….

fried rice 0.0001

….

and 1e-100

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

Perplexity

- The best language model is one that best predicts an unseen test data *W*, i.e. the one that gives the highest probability for those sentences.
- Perplexity is the inverse probability of the test set, normalised by the number of words:

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

Chain rule:

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

For bigrams:

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

Minimising perplexity is the same as maximising probability

Perplexity from Cross-Entropy

- **Cross-entropy** is another metric used for evaluating the confusion of the language model on a test corpus.
- Practically, it is easy to calculate as just the negative sum of the log probabilities divided by the length of the corpus:

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

- **Perplexity** can be simply **calculated from cross-entropy** as it's 2 (or whatever log base you're using) to the power of the cross-entropy:

$$\text{Perplexity}(W) \; = \; 2^{H(W)}$$

So, minimising cross-entropy is also the same as maximising probability

Lower perplexity, better model

- Training 38 million words, test 1.5 million words, Wall St. Journal

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

OUTLINE

1) Language Models: motivation
2) Language Models: ngram models
3) Language Models: evaluation
4) Smoothing

The danger of overfitting!

- N-gram models 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 generalise.

- One kind of generalisation: re-estimating ngrams with 0 counts:
  - To pre-empt sequences 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

- Test set
  - … denied the offer
  - … denied the loan

p(offer | denied the) = 0

- ngrams with zero probability!
  - We will assign 0 probability to the test set and hence we cannot compute perplexity (can't divide by 0)!
  - Also, given the model has seen this context and this word before (though not together), a more intelligent model should be able to assign a probability based on its knowledge of similar sequences- deal with known unknowns…

What can we do about this?

- Three main approaches:
  - **Smoothing**
    - Hold back some probability mass for unseen events
  - **Backoff & Interpolation**
    - Estimate n-gram probability from (n-1)-gram probability
  - **Class-based models**
    - Group words together, estimate class n-gram probability

Smoothing

- When we have sparse statistics from the counts:

C(denied the, w)
3 allegations
2 reports
1 claims
1 request

7 total

- ‘Steal’/spread around probability mass to generalize better. I.e. **Discount** some of the seen counts and add that discount to unseen counts:

C(denied the, w)
2.5 allegations
1.5 reports
0.5 claims
0.5 request
2 other

7 total

Add-one smoothing

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

- MLE estimate:

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

- Add-1 estimate:

$$p^{add-one}(w_i \mid w_{i-1}) = \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + V}$$

Vocab size

Add-one smoothing

- Add one to all counts (can be done during testing too). New counts will look like this:

Add 1 to all counts.
Now no 0
counts.

	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

Add-one smoothing

- Results in a **discount** (reduction) of the seen counts, but adding to the unseen ones to give the smoothed probabilities.

	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

Now no 0 probs.
Some higher non-0 probs slightly lower (discounted); some previously low probs now higher

Add-one smoothing

- Add-1 estimation is a blunt instrument- too much mass given to unseen events, some seen events reduced too much.

- So add-1 isn't used very much for language modelling:
  - We'll have a look at a couple of better methods!

- But add-1 is used to smooth other NLP models
  - For text classification (e.g. often in Naive Bayes, as per Unit 2).
  - In domains where the number of zeros isn't huge.

Add-k smoothing (generalized additive smoothing)

- Also additive Laplace smoothing, though sometimes **'Lidstone'/ add-k smoothing**
- Pretend we saw each word a value *k,* (0,1], more than we did.
- MLE estimate:

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

- Add-k estimate:

$$p^{add-k}(w_i \mid w_{i-1}) = \frac{C(w_{i-1}, w_i) + k}{C(w_{i-1}) + kV}$$

- Add-1 smoothing a special case where *k*=1.
- You can search for *k* that gives largest probability to the held-out data, using an optimisation/gradient descent method to find these efficiently, e.g. the Nelder-Mead algorithm.

Backoff and Interpolation

- Sometimes it helps to use **less** context
  - Condition on shorter context/history for contexts you haven’t learned much about
- **Backoff:**
  - e.g. in a trigram model, use the trigram prob if you have good evidence, otherwise bigram, otherwise unigram
- **Interpolation (with lower orders):**
  - mix unigram, bigram, trigram
- Interpolation tends to work better in general.

Backoff and Interpolation

- Simple interpolation

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

where:

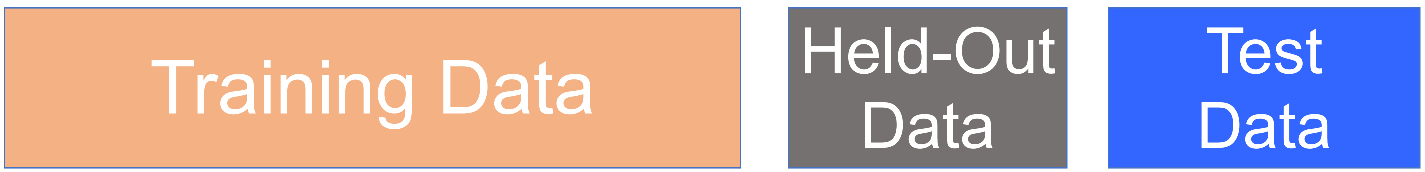
$$\sum_i \lambda_i = 1$$

- Lambdas conditional on context:

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

Backoff and Interpolation

- Use a **held-out** corpus to get the right λs



- 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 largest probability to held-out set
  - Again, you could use an optimisation/gradient descent method to find these efficiently, e.g. the Nelder-Mead algorithm.
- Advanced interpolation + backoff technique- **Kneser-Ney smoothing.** Uses **absolute discounting** and the lower-order models. See Goodman (2001).

Kneser-Ney Smoothing

- Intuition: a word like “Kong” may appear quite frequently in a certain text, however, it’s likely only to be a continuation for very few words (e.g. “Hong”, “King”). It should not be given high probability in general as it only occurs in those specific contexts.

- For the lower orders in the back-off, instead of probability of occurrence (a function of **token** frequency), use the probability of a word being a continuation in terms of how many **types** the word appears in as a continuation.

Kneser-Ney Smoothing

- e.g. For a tri-gram model, you can use a standard tri-gram model with a fixed absolute discount D for the main model:

$$\frac{C(w_{i-2}, w_{i-1}, w_i) - D}{C(w_{i-2}, w_{i-1})}$$

- But you also have two lower-order counts to back-off to and interpolate with:
  - For the unigram model, we want the number of bigram types the word w appears in as the continuation word: $|\{v : C(v,w) > 0\}|$

    normalised by $\sum_{w'} |\{v : C(vw') > 0\}|$ (number of bigram types)

    Which means 'Kong' will have a low probability as a unigram.

  - For the bigram model, we want the number of trigram types the words w_{i-1}, w_i appear in as the continuation: $|\{v : C(v, w_{i-1}, w_i) > 0\}|$

    normalised by $\sum_{w'} |\{v : C(v, w_{i-1}, w') > 0\}|$ (number of trigram types with this bigram's context word in position 2)

Kneser-Ney Smoothing

- Recursive formulation for lower orders.
- Highest order always uses counts of occurrences (tokens), lower orders use continuation type counts.

$$P_{\texttt{IKN}}(w_i|w_{i-2}w_{i-1}) = \frac{C(w_{i-2}w_{i-1}w_i)-D}{C(w_{i-2}w_{i-1})} + \lambda(w_{i-2}w_{i-1})P_{\texttt{ikn-mod-bigram}}(w_i|w_{i-1})$$

$$P_{\texttt{ikn-mod-bigram}}(w_i|w_{i-1}) = \frac{|\{v|C(vw_{i-1}w_i)>0\}|-D}{\sum_w |\{v|C(vw_{i-1}w)>0\}|} + \lambda(w_{i-1})P_{\texttt{ikn-mod-unigram}}(w_i)$$

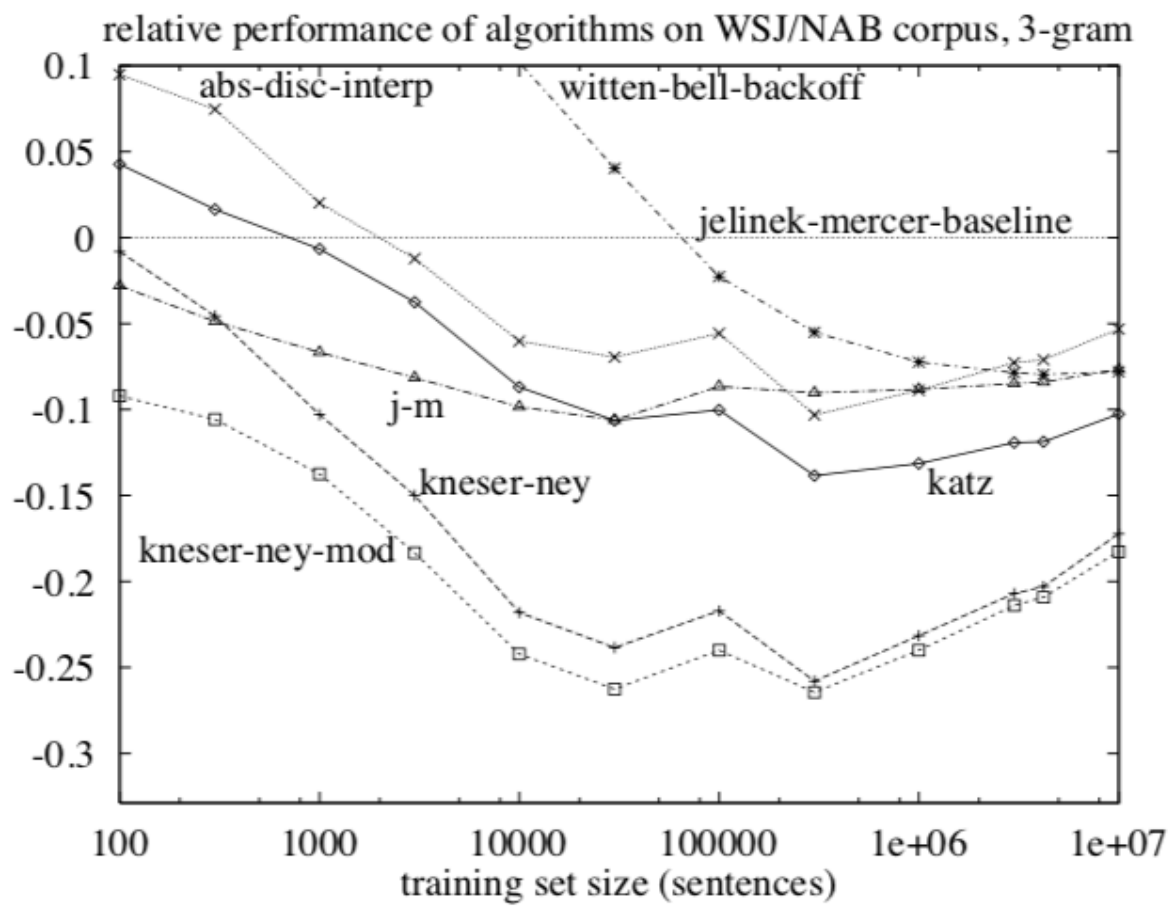
$$P_{\texttt{ikn-mod-unigram}}(w_i|w_{i-1}) = \frac{|\{v|vw_i)>0\}|-D}{\sum_w |\{v|C(vw)>0\}|}$$

Kneser-Ney smoothing

```
def kneser_ney_ngram_prob(ngram, discount, order):
    """
    ngram :: list of strings, the ngram
    discount :: float, the discount used
    order :: int, order of the model
    """
    # First, calculate the unigram continuation prob of the last token
    uni_num = |{v : C(v, ngram[-1]) > 0}| # number of bigram types the last word is the second word/continuation for
    uni_denom = ∑w'|{v : C(vw') > 0}| # number of unigram continuation (i.e. bigram) types in total

    probability = previous_prob = uni_num / uni_denom

    # Compute the higher order probs (from 2/bi-gram upwards) and interpolate them
    for d in range(2, order+1):
        context =  ngram[-(d):-1]. # define the context for this ngram of order d

        # Get the context count for the denominator
        if d == order:
            # When d = order (n), this is the counts of tokens of this context counted
            ngram_denom = C(context)
        else:
            # When d < order (n) this is the number of different n-gram types (not tokens) of order d+1 with this context
            ngram_denom = ∑w'|{v : C(v, context, w') > 0}|

        if ngram_denom != 0:
            # Get the ngram count for the numerator
            if d == order:
                ngram_num = C(ngram[-d:]) # number of tokens of ngram counted in corpus
            else:
                ngram_num = |{v : C(v, ngram[-d:]) > 0}| # number of types of ngram of n=d+1 which it is a continuation for
            current_prob = (ngram_num - discount) / ngram_denom # probability with fixed discount

            # Get the lambda for this context λ(context) = discount / ngram_denom * |{w : C(context,w) > 0}|
            # the number of word types that can follow the context (number of time normalised discount has been applied)
            nonzero = |{w : C(context,w) > 0}|
            lambda_context = discount / ngram_denom * nonzero

            # interpolate with previous probability of lower orders calculated so far
            current_prob += lambda_context * previous_prob
            previous_prob = current_prob
            probability = current_prob
        else:
            # if this context (e.g. bigram context for trigrams) has never been seen,
            # then we can only use the last order with a probability (e.g. unigram) and halt
            probability = previous_prob
            break
    return probability
```

Kneser-Ney Smoothing

- Goodman (2001) confirms KN smoothing is the best:



Kneser-Ney Smoothing

- Even in the era of neural language models using Recurrent Neural Nets (RNNs), 5-gram model with KN smoothing has been used in a voting system with RNNs to get the best results
- e.g. Mikolov (2010), training on 4 million words:

Model	PPL
KN 5gram	93.7
feedforward NN	85.1
recurrent NN	80.0
4xRNN + KN5	73.5

Summary

- Language models offer a way to assign a **probability** to a sentence or other sequence of words, and to **predict a word from preceding words.**
- n-gram models are **Markov models** that estimate words from a fixed window of previous words. n-gram probabilities can be estimated by counting in a corpus and normalizing (the maximum likelihood estimate).
- n-gram language models are evaluated extrinsically in some task, or intrinsically using **perplexity**.
- The perplexity of a **test set** according to a language model is the geometric mean of the inverse test set probability computed by the model.

Summary

- **Smoothing** algorithms provide a more sophisticated way to estimate the probability of n-grams. Commonly used smoothing algorithms for n-grams rely on lower-order n-gram counts through backoff or interpolation.
- Both **backoff** and **interpolation** use discounting to create probability distributions over different orders.

- **<u>Lab activity (unassessed): Implement Add-one smoothing, generalised additive smoothing and use Kneser-Ney smoothing.</u>**

Reading

- Manning and Schuetze (1999) Chapter 6
- Jurafsky and Martin (3rd Ed) Chapter 3
- (Optional) Goodman (2001)- "A bit of Progress in Language Modeling"