Natural Language Processing

Language modeling

Sofia Serrano
sofias6@cs.washington.edu

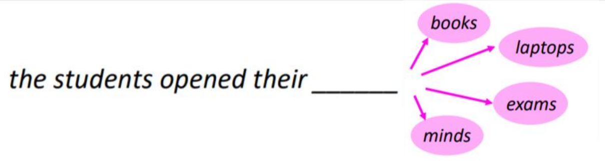
Credit to Yulia Tsvetkov and Noah Smith for slides

Announcements

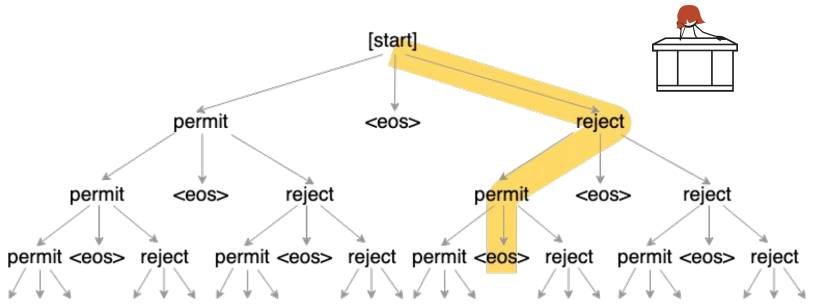
- A1 is due at 11:59pm today
    - For details on how to tag and submit your assignment, see Leo's tutorial video linked in Sunday's Ed announcement
    - Late days
        - Remember: you can use up to **three** late days on A1 out of your pool of five total for the quarter with no penalty (absolute latest to tag your A1 submission using all three possible late days would be 11:59pm on Monday 1/30)
- We're holding extra office hours this week
    - See the office hour schedule google doc from the Sunday Ed announcement, also linked on the course website under "Announcements"
- Daksh's regular Monday office hour time and place has shifted to 4:30-5:30pm in-person in Gates 152 (website is updated)

Last class we covered...

- What language modeling is



- How language models produce a probability distribution over language
    - And how to calculate the probability a particular language model allocates to any utterance



- How n-gram language models work, and how to estimate them

$$P(w_i \mid w_1, w_2, \ldots, w_{i-1}) = \frac{C(w_1, w_2, \ldots, w_{i-1}, w_i)}{C(w_1, w_2, \ldots, w_{i-1})}$$

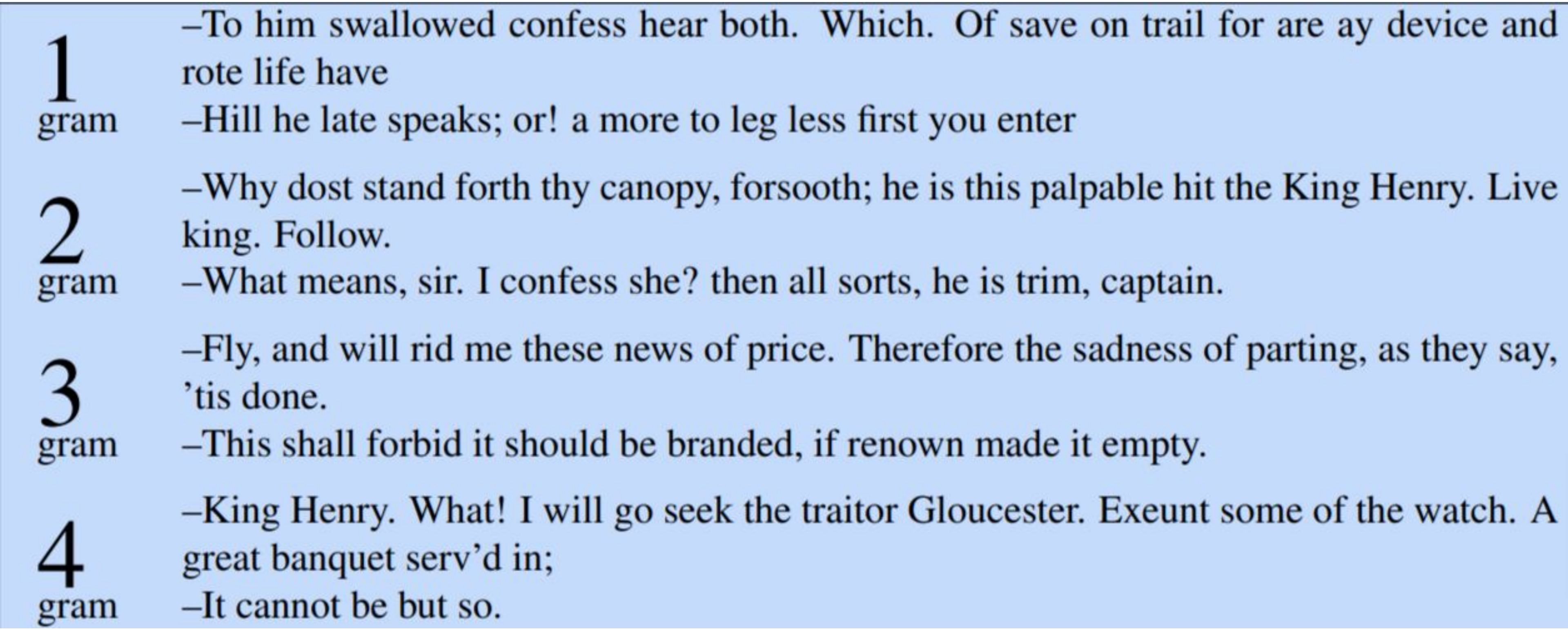
Our agenda for today

- Wrapping up n-gram language models
  - What’s not great about them?
  - What are some techniques we can use to address those weaknesses?

- How do we evaluate language models?

- A more careful consideration of language model vocabulary

Sampling from a language model



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.

Addressing n-gram models' weaknesses

Sparsity

- Maximum likelihood for estimating q
    - Let c(w$_1$, …, w$_n$) be the number of times that *n*-gram appears in a corpus

$$q(w_i \mid w_{i-2}, w_{i-1}) = \frac{c(w_{i-2}, w_{i-1}, w_i)}{c(w_{i-2}, w_{i-1})}$$

    - If vocabulary has 20,000 words ⇒ Number of parameters is 8 x 10^{12}!

Dealing with sparsity

- For most N-grams, we have few observations
- General approach: modify observed counts to improve estimates
    - Back-off:
        - use trigram if you have good evidence;
        - otherwise bigram, otherwise unigram
    - Interpolation: approximate counts of N-gram using combination of estimates from related denser histories
    - Discounting: allocate probability mass for unobserved events by discounting counts for observed events

Linear interpolation

- Given a corpus of length M

Trigram model:

$$q(w_i \mid w_{i-2}, w_{i-1}) = \frac{c(w_{i-2}, w_{i-1}, w_i)}{c(w_{i-1}, w_i)}$$

Bigram model:

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

Unigram model:

$$q(w_i) = \frac{c(w_i)}{M}$$

Linear interpolation

- Combine the three models to get all benefits

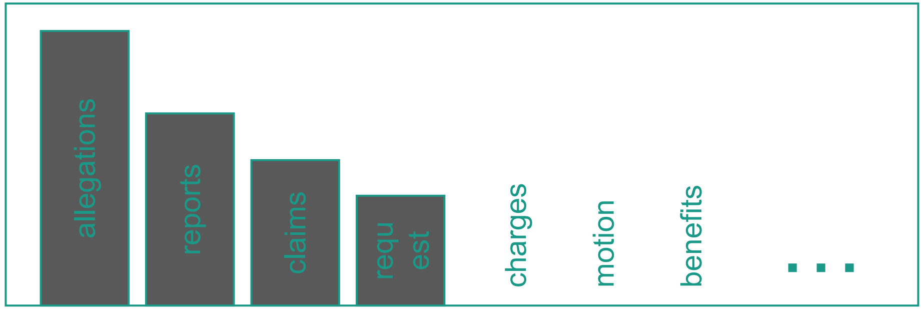
$$\begin{aligned} q_{LI}(w_i \mid w_{i-2}, w_{i-1}) &= \lambda_1 \times q(w_i \mid w_{i-2}, w_{i-1}) \\ &+ \lambda_2 \times q(w_i \mid w_{i-1}) \\ &+ \lambda_3 \times q(w_i) \end{aligned}$$

$$\lambda_i \geq 0,\ \ \lambda_1 + \lambda_2 + \lambda_3 = 1$$

Discounting/smoothing methods

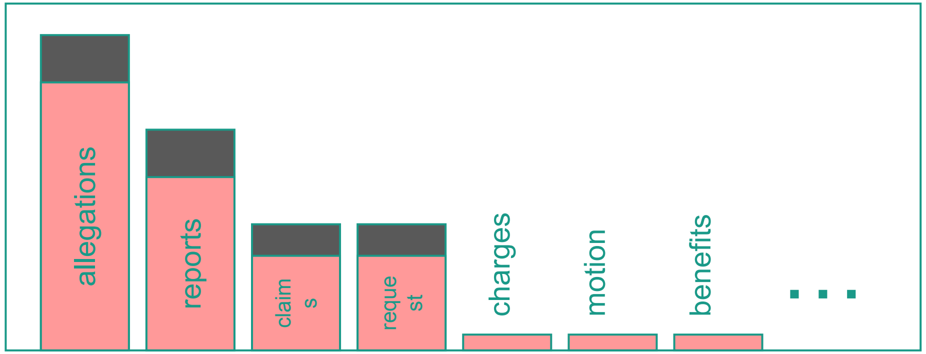
- We often want to make estimates from sparse statistics:

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



- Smoothing flattens spiky distributions so they generalize better:

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



▪ Very important all over NLP, but easy to do badly

Evaluating language models

Intuition

We want a model that assigns probability to words in a way that mirrors actual language.

We want a model that assigns high probability to actual language (that it wasn’t trained on) that it’s meant to model.

We want a model that isn’t “surprised” by the actual test data it sees.

→ We want a model with low “perplexity”

Building up a definition of perplexity

What do we have to work with?

- Test data:

  {"w_1^1 w_2^1 w_3^1 w_4^1 <eos>",

  "w_1^2 w_2^2 <eos>",

  "w_1^3 w_2^3 w_3^3 <eos>"}

- The model's probability of that data

Aside: how do we calculate probability over the full data?

Up until now we've only seen probabilities over single instances like

$$\begin{aligned} p(\text{the dog barks STOP}) = & q(the \mid *, *)\times \\ & q(dog \mid *, the)\times \\ & q(barks \mid the, dog)\times \\ & q(STOP \mid dog, barks) \end{aligned}$$

We'll call "the dog barks <eos>" instance 1, with a length of 4 (counting the <eos> token). The probability of the *full test data* will then be

$$\prod_{\text{instance } i \in \text{test instances}} p(\text{instance } i)$$

Building up a definition of perplexity

- Test data: S = {s_1, s_2, …, s_{sent}}

$$p(\mathcal{S}) = \prod_{i=1}^{sent} p(s_i)$$

  - *sent* is the number of sentences (instances) in the test data

Practical issue

- Multiplying very small numbers results in numerical underflow
  - Solution: we do every operation in log space
  - (also adding is faster than multiplying)

Building up a definition of perplexity

- Test data: S = $\{s_1, s_2, \ldots, s_{sent}\}$

$$p(\mathcal{S}) = \prod_{i=1}^{sent} p(s_i)$$

$$\log_2 p(\mathcal{S}) = \sum_{i=1}^{sent} \log_2 p(s_i)$$

  - *sent* is the number of sentences in the test data

Problem: this is sensitive to amount of test data.

Let’s say our model produced the following probabilities for these two single-instance test sets:

p(hello how are you) = p(hello) * p(how | hello) * p(are | hello how) * p(you | hello how are)

2^-5 2^-4 2^-3 2^-3

* p(<eos> | hello how are you)

2^-4

p(waltz pudding) = p(waltz) * p(pudding | waltz) * p(<eos> | waltz pudding)

2^-7 2^-7 2^-5

Problem: this is sensitive to amount of test data.

Let's say our model produced the following probabilities for these two single-instance test sets:

p(hello how are you) = p(hello) * p(how | hello) * p(are | hello how) * p(you | hello how are)

* p(<eos> | hello how are you)

p(waltz pudding) = p(waltz) * p(pudding | waltz) * p(<eos> | waltz pudding)

Solution: normalize the log probabilities by the total length (in tokens) of the test data!

Building up a definition of perplexity

- Test data: $S = \{s_1, s_2, \ldots, s_{sent}\}$

$$p(\mathcal{S}) = \prod_{i=1}^{sent} p(s_i)$$

$$\log_2 p(\mathcal{S}) = \sum_{i=1}^{sent} \log_2 p(s_i)$$

$$l = \frac{1}{M} \sum_{i=1}^{sent} \log_2 p(s_i)$$

Our new quantity still goes from -∞ to 0...

  - *sent* is the number of sentences in the test data
  - M is the number of words in the test corpus

Evaluation: perplexity

- Test data: S = {s_1, s_2, …, s_{sent}}

$$p(\mathcal{S}) = \prod_{i=1}^{sent} p(s_i)$$

$$\log_2 p(\mathcal{S}) = \sum_{i=1}^{sent} \log_2 p(s_i)$$

$$\text{perplexity} = 2^{-l},\ l = \frac{1}{M}\sum_{i=1}^{sent} \log_2 p(s_i)$$

  - *sent* is the number of sentences in the test data
  - M is the number of words in the test corpus

Evaluation: perplexity

- Test data: $S = \{s_1, s_2, \ldots, s_{sent}\}$

$$p(\mathcal{S}) = \prod_{i=1}^{sent} p(s_i)$$

$$\log_2 p(\mathcal{S}) = \sum_{i=1}^{sent} \log_2 p(s_i)$$

$$\text{perplexity} = 2^{-l},\ l = \frac{1}{M} \sum_{i=1}^{sent} \log_2 p(s_i)$$

  - *sent* is the number of sentences in the test data
  - M is the number of words in the test corpus
  - A good language model has high p(S) and low perplexity

Gut check for perplexity values we expect

$$\text{perplexity} = 2^{-l},\ l = \frac{1}{M}\sum_{i=1}^{sent} \log_2 p(s_i)$$

- If the model were the “perfect” language model and allocated probability 1 to every word that came next, its perplexity would be 1

- If the model assigned uniform probability over all its vocabulary words each time it estimated the probability distribution over the next output word, its perplexity would be |V|.

(Towards) rare-word-proofing our vocabulary

Is a finite vocabulary realistic?

No

no

n0

-no

notta

№

/no

//no

(no

|no

Why are we talking about this now?

Didn’t we already talk about vocabularies when we talked about text classification?

Yes, BUT in that scenario, if we missed some words from the input text, we were fine.

Language modeling isn’t, because it needs to assign a probability to the entire string of text.

p(The Bundestag passed legislation this week …)

Idea #1: have an out-of-vocabulary token

We call this token <UNK>.

Pros:

- We're guaranteed to allocate probability to every possible text this way

Cons:

- Need to screen for it when using your language model to generate text
- It's… really, really easy to use <UNK> to make it look like your language model's better than it actually is.
    - Map every word to <UNK>, and your perplexity will be 1!
- You can't compare two different language models that have different mappings to <UNK>.

Idea #2: make your language model character-level

Instead of estimating probabilities over *words*, estimate over *characters.*

Pros:

- (Provided you're careful) this should account for just about any text!
- Your LM will only ever try to generate actual, non-<UNK> characters
- Compact vocabulary!

Cons:

- … depending on the language, the compact vocabulary.
  - Model has a much harder time separating out information about how different larger units (words) behave

Idea #3: augment a character vocabulary with some longer character sequences

Maybe we're okay with expanding the size of our vocabulary up to a certain budget!

If we add some common longer sequences to our vocabulary, we could benefit from *both*

- parameters specifically trained for those longer-sequence vocabulary items
- the coverage of possible inputs that character-level LMs provide

How do we choose which longer character sequences to add?

Byte Pair Encoding gives us an answer!

Data compression algorithm from the 1990s

Adapted for NLP vocabulary creation in 2016 by Sennrich et al.

- Originally developed for machine translation models
- But ideas like it have since taken off for language modeling too

Idea: iteratively merge common token pairs to create new tokens, **without removing the smaller component tokens from the vocabulary**

Working through an example

Training corpus consisting of five separate pieces of text (pretend _ is a space):

RATION_	ORATION_	AT_	SAMPLE_	POTION_

Vocabulary:

A, R, T, I, O, N, _, S, M, P, L, E

Working through an example

Training corpus consisting of five separate pieces of text (pretend _ is a space):

RATION_	ORATION_	AT_	SAMPLE_	POTION_

RA, AT, TI, IO, ON, N_	OR, RA, AT, TI, IO, ON, N_	AT, T_	SA, AM, MP, PL, LE, E_	PO, OT, TI, IO, ON, N_

Vocabulary:

A, R, T, I, O, N, _, S, M, P, L, E

AT: 3	ON: 3	SA: 1	...
TI: 3	N_: 3	AM: 1	
IO: 3	RA: 2	MP: 1	

1. Count the number of times each current token pair occurs in the training corpus

Working through an example

Training corpus consisting of five separate pieces of text (pretend _ is a space):

RATION_	ORATION_	AT_	SAMPLE_	POTION_

Vocabulary:

A, R, T, I, O, N, _, S, M, P, L, E

IO

AT: 3	ON: 3	SA: 1	...
TI: 3	N_: 3	AM: 1	
IO: 3	RA: 2	MP: 1	

2. Pick any **one** of the highest-count token pairs and add it to the vocabulary as its own new token

Working through an example

Training corpus consisting of five separate pieces of text (pretend _ is a space):

RATION_	ORATION_	AT_	SAMPLE_	POTION_

Vocabulary:

A, R, T, I, O, N, _, S, M, P, L, E

IO

AT: 3	ON: 3	SA: 1	...
TI: 3	N_: 3	AM: 1	
IO: 3	RA: 2	MP: 1	

3. Replace all instances of that pair in the training corpus with the newly added token

Now just repeat!

Training corpus consisting of five separate pieces of text (pretend _ is a space):

RATION_	ORATION_	AT_	SAMPLE_	POTION_
RA, AT, TIO, ION, N_	OR, RA, AT, TIO, ION, N_	AT, T_	SA, AM, MP, PL, LE, E_	PO, OT, TIO, ION, N_

Vocabulary:

A, R, T, I, O, N, _, S, M, P, L, E

IO

AT: 3	N_: 3	AM: 1	...
TIO: 3	RA: 2	MP: 1	
ION: 3	SA: 1	PL: 1	

1. Count the number of times each current token pair occurs in the training corpus

Now just repeat!

Training corpus consisting of five separate pieces of text (pretend _ is a space):

RATION_	ORATION_	AT_	SAMPLE_	POTION_

Vocabulary:

A, R, T, I, O, N, _, S, M, P, L, E

IO

TIO

AT: 3	N_: 3	AM: 1	...
TIO: 3	RA: 2	MP: 1	
ION: 3	SA: 1	PL: 1	

2. Pick any **one** of the highest-count token pairs and add it to the vocabulary as its own new token

Now just repeat!

Training corpus consisting of five separate pieces of text (pretend _ is a space):

RATION_ ORATION_ AT_ SAMPLE_ POTION_

Vocabulary:

A, R, T, I, O, N, _, S, M, P, L, E

IO

TIO

AT: 3	N_: 3	AM: 1	...
TIO: 3	RA: 2	MP: 1	
ION: 3	SA: 1	PL: 1	

3. Replace all instances of that pair in the training corpus with the newly added token

Repeat until when?

For original compression purposes:

- Until there are no more token pairs that appear more than once across your training corpus

For NLP purposes, though:

- Until you've reached some predetermined cap on the number of vocabulary words you're willing to have (perhaps 10^4 or something)

What kind of vocabulary do we end up with?

Training corpus consisting of five separate pieces of text (pretend _ is a space):

RATION_ ORATION_ AT_ SAMPLE_ POTION_

Vocabulary resulting from continuing rounds of BPE algorithm until no token pair is repeated:

A, R, T, I, O, N, _, S, M, P, L, E
IO
TIO
N_
TION_
ATION_
RATION_

Getting text into its BPE-encoded form (after vocabulary has been fixed)

Important to keep track of not only *which* new tokens we added while performing BPE, but also *in which order* we added them.

A, R, T, I, O, N, _, S, M, P, L, E
IO
TIO
N_
TION_
ATION_
RATION_

To apply our BPE vocabulary to some new text, split that new text into its component characters and then apply all merge operations *in the order they were added to the vocabulary.*

Different variants of creating BPE vocabularies

Do we allow merges that could create tokens with whitespace in their middle?

Suppose our training corpus was

The White House reported…   The House decided that…

Pros of allowing merging over whitespace:

- Potentially allows single token for multiword expression (e.g., “White House”)

Pros of *not* allowing merging over whitespace:

- Guarantees a nice mapping of word-level annotations to your potentially subword-tokenized text

Language models: Conclusion (for now)

Which of this material only applied to *simple* language models?

Just the n-gram material! *(Since the Markovian assumption is, well, false in a language context)*

For current state-of-the-art neural language models, all of the following still apply:

- Task definition of language modeling
- Evaluation via perplexity
- Vocabulary creation considerations

Language modeling will make reappearances later in the course…

Next class

Lexical semantics!