Foundations of Natural Language Processing
Lecture 6c
Language Models: Smoothing

Alex Lascarides

Recap

- LMs are useful for many applications.
- To overcome sparse data, you can use small n in n-gram LMs to predict the probability of arbitrarily large word sequences.
- We can evaluate the quality of an n-gram LM using per word cross entropy

Now:

- Smoothing: A further strategy for tackling the sparse data problem.

Sparse data, again

Suppose now we build a *trigram* model from Moby Dick and evaluate the same sentence.

- But I spent three never occurs, so $P_{MLE}(\text{three} \mid \text{I spent}) = 0$
- which means the cross-entropy is infinite.
- Basically right: our model says I spent three should never occur, so our model is infinitely wrong/surprised when it does!
- Even with a unigram model, we will run into words we never saw before. So even with short N-grams, we need better ways to estimate probabilities from sparse data.

Smoothing

- The flaw of MLE: it estimates probabilities that make the training data maximally probable, by making everything else (unseen data) minimally probable.

- **Smoothing** methods address the problem by stealing probability mass from seen events and reallocating it to unseen events.

- Lots of different methods, based on different kinds of assumptions. We will discuss just a few.

Add-One (Laplace) Smoothing

- Just pretend we saw everything one more time than we did.

$$P_{\mathrm{MLE}}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}$$

$$\Rightarrow \qquad P_{+1}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i) + 1}{C(w_{i-2}, w_{i-1})} \qquad ?$$

Add-One (Laplace) Smoothing

- Just pretend we saw everything one more time than we did.

$$P_{\text{MLE}}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}$$

$$\Rightarrow \qquad P_{+1}(w_i|w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i) + 1}{C(w_{i-2}, w_{i-1})} \qquad ?$$

- NO! Sum over possible w_i (in vocabulary V) must equal 1:

$$\sum_{w_i \in V} P(w_i|w_{i-2}, w_{i-1}) = 1$$

- If increasing the numerator, must change denominator too.

Add-one Smoothing: normalization

- We want: $$\sum_{w_i \in V} \frac{C(w_{i-2}, w_{i-1}, w_i) + 1}{C(w_{i-2}, w_{i-1}) + x} = 1$$

- Solve for x:

$$\begin{aligned}
\sum_{w_i \in V} (C(w_{i-2}, w_{i-1}, w_i) + 1) &= C(w_{i-2}, w_{i-1}) + x \\
\sum_{w_i \in V} C(w_{i-2}, w_{i-1}, w_i) + \sum_{w_i \in V} 1 &= C(w_{i-2}, w_{i-1}) + x \\
C(w_{i-2}, w_{i-1}) + v &= C(w_{i-2}, w_{i-1}) + x \\
v &= x
\end{aligned}$$

where $v =$ vocabulary size.

Add-one example (1)

- *Moby Dick* has one trigram that begins with I spent (it's I spent in) and the vocabulary size is 17231.

- Comparison of MLE vs Add-one probability estimates:

	MLE	+1 Estimate
$\hat{P}(\text{three} \mid \text{I spent})$	0	0.00006
$\hat{P}(\text{in} \mid \text{I spent})$	1	0.0001

- $\hat{P}(\text{in}|\text{I spent})$ seems very low, especially since in is a very common word. But can we find better evidence that this method is flawed?

Add-one example (2)

- Suppose we have a more common bigram w_1, w_2 that occurs 100 times, 10 of which are followed by w_3.

	MLE	+1 Estimate
$\hat{P}(w_3 \mid w_1, w_2)$	$\frac{10}{100}$	$\frac{11}{17331}$
		≈ 0.0006

- Shows that the very large vocabulary size makes add-one smoothing steal *way* too much from seen events.

- In fact, MLE is pretty good for frequent events, so we shouldn't want to change these much.

Add-α (Lidstone) Smoothing

- We can improve things by adding $\alpha < 1$.

$$P_{+\alpha}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + \alpha}{C(w_{i-1}) + \alpha v}$$

- Like Laplace, assumes we know the vocabulary size in advance.
- But if we don't, can just add a single "unknown" (UNK) item, and use this for all unknown words during testing.
- Then: how to choose α?

Optimizing α (and other model choices)

- Use a three-way data split: **training** set (80-90%), **held-out** (or **development**) set (5-10%), and **test** set (5-10%)
  - Train model (estimate probabilities) on training set with different values of α
  - Choose the α that minimizes cross-entropy on development set
  - Report final results on test set.
- More generally, use dev set for evaluating different models, debugging, and optimizing choices. Test set simulates deployment, use it only once!
- Avoids overfitting to the training set and even to the test set.

Better smoothing: Good-Turing

- Previous methods changed the denominator, which can have big effects even on frequent events.

- Good-Turing changes the numerator. Think of it like this:
  - MLE divides count c of N-gram by count n of history:

$$P_{\mathrm{MLE}} = \frac{c}{n}$$

  - Good-Turing uses **adjusted counts** c^* instead:

$$P_{\mathrm{GT}} = \frac{c^*}{n}$$

Good-Turing in Detail

- Push every probability total down to the count class below.
- Each *count* is reduced slightly (Zipf): we're discounting!

c	N_c	P_c	$P_c[\textit{total}]$	$c*$	$P*_c$	$P*_c[\textit{total}]$
0	N_0	0	0	$\frac{N_1}{N_0}$	$\frac{\frac{N_1}{N_0}}{N}$	$\frac{N_1}{N}$
1	N_1	$\frac{1}{N}$	$\frac{N_1}{N}$	$2\frac{N_2}{N_1}$	$\frac{2\frac{N_2}{N_1}}{N}$	$\frac{2N_2}{N}$
2	N_2	$\frac{2}{N}$	$\frac{2N_2}{N}$	$3\frac{N_3}{N_2}$	$\frac{3\frac{N_3}{N_2}}{N}$	$\frac{3N_3}{N}$

- c: count
  N_c: number of different items with count c
  P_c: MLE estimate of prob. of that item
  $P_c[\textit{total}]$: MLE total probability mass for *all* items with that count.
  $c*$: Good-Turing smoothed version of the count
  $P*_c$ and $P*_c[\textit{total}]$: Good-Turing versions of P_c and $P_c[\textit{total}]$

Some Observations

- Basic idea is to arrange the discounts so that the amount we *add* to the total probability in row 0 is matched by all the discounting in the other rows.
- Note that we only know N_0 if we actually know what's missing.
- And we can't always estimate what words are missing from a corpus.
- But for bigrams, we often assume $N_0 = V^2 - N$, where V is the different (observed) words in the corpus.

Good-Turing Smoothing: The Formulae

Good-Turing discount depends on (real) adjacent count:

$$
\begin{aligned}
c* = &\quad (c+1)\frac{N_{c+1}}{N_c} \\
P*_c = &\quad \frac{c*}{N} \\
= &\quad \frac{(c+1)\frac{N_{c+1}}{N_c}}{N}
\end{aligned}
$$

- Since counts tend to go down as c goes up, the multiplier is < 1.
- The sum of all discounts is $\frac{N_1}{N_0}$. We need it to be, given that this is our GT count for row 0!

Good-Turing for 2-Grams in Europarl

Count	Count of counts	Adjusted count	Test count
c	N_c	c^*	t_c
0	7,514,941,065	0.00015	0.00016
1	1,132,844	0.46539	0.46235
2	263,611	1.40679	1.39946
3	123,615	2.38767	2.34307
4	73,788	3.33753	3.35202
5	49,254	4.36967	4.35234
6	35,869	5.32928	5.33762
8	21,693	7.43798	7.15074
10	14,880	9.31304	9.11927
20	4,546	19.54487	18.95948

t_c are average counts of bigrams in test set that occurred c times in corpus: fairly close to estimate c^*.

Good-Turing justification: 0-count items

- Estimate the probability that the next observation is previously unseen (i.e., will have count 1 once we see it)

$$P(\text{unseen}) = \frac{N_1}{n}$$

This part uses MLE!

- Divide that probability equally amongst all unseen events

$$P_{\text{GT}} = \frac{1}{N_0}\frac{N_1}{n} \quad \Rightarrow \quad c^* = \frac{N_1}{N_0}$$

Good-Turing justification: 1-count items

- Estimate the probability that the next observation was seen once before (i.e., will have count 2 once we see it)

$$P(\text{once before}) = \frac{2N_2}{n}$$

- Divide that probability equally amongst all 1-count events

$$P_{\text{GT}} = \frac{1}{N_1}\frac{2N_2}{n} \quad \Rightarrow \quad c^* = \frac{2N_2}{N_1}$$

- Same thing for higher count items

Summary

- We need smoothing to deal with unseen N-grams.

- Add-1 and Add-α are simple,
  but may not work very well (it all depends. . .).

- Good-Turing is more sophisticated,
  it may yield better models.

Next time: Alternative, perhaps better, approaches to smoothing.