Language models

- Language models tell us $P(\mathbf{w}) = P(w_1 \ldots w_n)$
  Roughly: Is this string of words a "good" one in my language?
- Used in noisy channel model for many applications.

Example uses of language models

- Machine translation: reordering, word choice.
  
  \[
  P_{\text{LM}}(\text{the house is small}) \quad \text{>} \quad P_{\text{LM}}(\text{small the is house})
  \]
  
  \[
  P_{\text{LM}}(\text{I am going home}) \quad \text{>} \quad P_{\text{LM}}(\text{I am going house})
  \]
  
  \[
  P_{\text{LM}}(\text{We'll start eating}) \quad \text{>} \quad P_{\text{LM}}(\text{We shall commence consuming})
  \]

- Speech recognition: word choice:
  
  \[
  P_{\text{LM}}(\text{some morphosyntactic analyses}) \quad \text{>} \quad P_{\text{LM}}(\text{some more faux syntactic analyses})
  \]
  
  \[
  P_{\text{LM}}(\text{I put it on today}) \quad \text{>} \quad P_{\text{LM}}(\text{I putted onto day})
  \]

N-Gram Language Models

- Given: a string of English words $\mathbf{w} = w_1, w_2, w_3, \ldots, w_n$
- Compute $P(\mathbf{w})$ using chain rule + Markov assumption. E.g., a one-word history gives us a bigram model:
  
  \[
  P(w_1, w_2, w_3, \ldots, w_n) \approx P(w_1) \ P(w_2|w_1) \ P(w_3|w_2) \ldots P(w_n|w_{n-1})
  \]
Estimating N-Gram Probabilities

- Maximum likelihood (relative frequency) estimation for bigrams:
  \[ P_{ML}(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)} \]
- Collect counts over a large text corpus
- Millions to billions of words are easy to get
  (trillions of English words available on the web)

Example: 3-Gram

- Counts for trigrams and estimated word probabilities

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
<th>Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>paper</td>
<td>801</td>
<td>0.458</td>
</tr>
<tr>
<td>group</td>
<td>640</td>
<td>0.367</td>
</tr>
<tr>
<td>light</td>
<td>110</td>
<td>0.063</td>
</tr>
<tr>
<td>party</td>
<td>27</td>
<td>0.015</td>
</tr>
<tr>
<td>ecu</td>
<td>21</td>
<td>0.012</td>
</tr>
<tr>
<td>the green</td>
<td>total: 1748</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
<th>Prob.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cross</td>
<td>123</td>
<td>0.547</td>
</tr>
<tr>
<td>tape</td>
<td>31</td>
<td>0.138</td>
</tr>
<tr>
<td>army</td>
<td>9</td>
<td>0.040</td>
</tr>
<tr>
<td>flag</td>
<td>6</td>
<td>0.111</td>
</tr>
<tr>
<td>the red</td>
<td>total: 225</td>
<td></td>
</tr>
</tbody>
</table>

- 225 trigrams in the Europarl corpus start with the red
- 123 of them end with cross
- Maximum likelihood probability is \( \frac{123}{225} = 0.547 \).

How good is the LM?

- A good model \( \mathcal{M} \) assigns a text of real English \( \hat{w} \) a high probability.
- Can be measured with cross entropy:
  \[ H_M(w_1 \ldots w_n) = -\frac{1}{n} \log P_M(w_1 \ldots w_n) \]
  - Avg neg log probability our model assigns to each word we saw
- Or, perplexity:
  \[ PP_M(\hat{w}) = 2^{H_M(\hat{w})} \]
### Comparison 1–4-Gram

<table>
<thead>
<tr>
<th>word</th>
<th>unigram</th>
<th>bigram</th>
<th>trigram</th>
<th>4-gram</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>6.684</td>
<td>3.197</td>
<td>3.197</td>
<td>3.197</td>
</tr>
<tr>
<td>would</td>
<td>8.342</td>
<td>2.884</td>
<td>2.791</td>
<td>2.791</td>
</tr>
<tr>
<td>like</td>
<td>9.129</td>
<td>2.026</td>
<td>1.031</td>
<td>1.290</td>
</tr>
<tr>
<td>to</td>
<td>5.081</td>
<td>0.402</td>
<td>0.144</td>
<td>0.113</td>
</tr>
<tr>
<td>commend</td>
<td>15.487</td>
<td>12.335</td>
<td>8.794</td>
<td>8.633</td>
</tr>
<tr>
<td>the</td>
<td>3.885</td>
<td>1.402</td>
<td>1.084</td>
<td>0.880</td>
</tr>
<tr>
<td>rapporteur</td>
<td>10.840</td>
<td>7.319</td>
<td>2.763</td>
<td>2.350</td>
</tr>
<tr>
<td>on</td>
<td>6.765</td>
<td>4.140</td>
<td>4.150</td>
<td>1.862</td>
</tr>
<tr>
<td>his</td>
<td>10.678</td>
<td>7.316</td>
<td>2.367</td>
<td>1.978</td>
</tr>
<tr>
<td>work</td>
<td>9.993</td>
<td>4.816</td>
<td>3.498</td>
<td>3.294</td>
</tr>
<tr>
<td>.</td>
<td>4.896</td>
<td>3.020</td>
<td>1.785</td>
<td>1.510</td>
</tr>
<tr>
<td>&lt;/s&gt;</td>
<td>4.828</td>
<td>0.005</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>average</td>
<td>8.051</td>
<td>4.072</td>
<td>2.634</td>
<td>2.251</td>
</tr>
<tr>
<td>perplexity</td>
<td>265.136</td>
<td>16.817</td>
<td>6.206</td>
<td>4.758</td>
</tr>
</tbody>
</table>

---

### Unseen N-Grams

- What happens when I try to compute $P(\text{consuming}|\text{shall commence})$?
  - We may have seen $\text{shall commence}$ in our corpus
  - We have never seen $\text{shall commence consuming}$ in our corpus
  - $P(\text{consuming}|\text{shall commence}) = 0$

- Any sentence with $\text{shall commence consuming}$ will be assigned probability 0

  The guests shall commence consuming supper
  Green inked shall commence consuming garden the

---

### The problem with MLE

- MLE estimates probabilities that make the observed data maximally probable
- by assuming anything unseen cannot happen
- It over-fits the training data
- **Smoothing** methods reassign some probability mass from observed to unobserved events
Add-One Smoothing

- For all possible bigrams, add one more count.

\[ P_{ML}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i)}{C(w_{i-1})} \]

\[ \Rightarrow P_{+1}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1})} \]

Add-One Smoothing: normalization

- We want:

\[ \sum_{w_i \in V} \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + x} = 1 \]

- Solve for \( x \):

\[ \sum_{w_i \in V} \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + x} = C(w_{i-1}) + x \]

\[ \sum_{w_i \in V} C(w_{i-1}, w_i) + \sum_{w_i \in V} 1 = C(w_{i-1}) + x \]

\[ C(w_{i-1}) + v = C(w_{i-1}) + x \]

- So, \( P_{+1}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + 1}{C(w_{i-1}) + v} \) where \( v \) = vocabulary size.

Add-One Smoothing: effects

- Large vocabulary size means \( v \) is often much larger than \( C(w_{i-1}) \), overpowers actual counts.

- Ex: in Europarl, \( v = 86,700 \) word types (30m tokens, max \( C(w_{i-1}) = 2m \)).

- Compute some example probabilities:

<table>
<thead>
<tr>
<th>( C(w_{i-1}, w_i) )</th>
<th>( P_{ML} )</th>
<th>( P_{+1} \approx )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1/100</td>
<td>1/970</td>
</tr>
<tr>
<td>10</td>
<td>1/10k</td>
<td>1/10k</td>
</tr>
<tr>
<td>1</td>
<td>1/10k</td>
<td>1/100</td>
</tr>
<tr>
<td>0</td>
<td>1/97k</td>
<td>1/87k</td>
</tr>
<tr>
<td>100</td>
<td>1/100</td>
<td>1/970</td>
</tr>
<tr>
<td>10</td>
<td>1/10k</td>
<td>1/43k</td>
</tr>
<tr>
<td>1</td>
<td>1/100</td>
<td>1/43k</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1/87k</td>
</tr>
</tbody>
</table>
The problem with Add-One smoothing

- All smoothing methods “steal from the rich to give to the poor”
- Add-one smoothing steals way too much
- ML estimates for frequent events are quite accurate, don’t want smoothing to change these much.

Add-α Smoothing

- Add $\alpha < 1$ to each count
  \[ P_{+\alpha}(w_i|w_{i-1}) = \frac{C(w_{i-1}, w_i) + \alpha}{C(w_{i-1}) + \alpha v} \]
- Simplifying notation: $c$ is n-gram count, $n$ is history count
  \[ P_{+\alpha} = \frac{c + \alpha}{n + \alpha v} \]
- What is a good value for $\alpha$?

Optimizing $\alpha$

- Divide corpus into 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 $\alpha$
- Choose the value of $\alpha$ that minimizes perplexity on development set
- Report final results on test set

A general methodology

- Training/dev/test split is used across machine learning
- Development set used for evaluating different models, debugging, optimizing parameters (like $\alpha$)
- Test set simulates deployment; only used once final model and parameters are chosen. (Ideally: once per paper)
- Avoids overfitting to the training set and even to the test set
Adjusted Counts

• Previously, we estimated probabilities based on actual counts

\[ P_{\text{ML}} = \frac{c}{n} \]

• Then, we changed the formula to estimate smoothed probabilities

\[ P_{+\alpha} = \frac{c + \alpha}{n + \alpha v} \]

• Another view: we adjusted the counts \( c \)

\[ P_{+\alpha} = \frac{c^{*}}{n} \Rightarrow c^{*} = n P_{+\alpha} = (c + \alpha) \frac{n}{n + \alpha v} \]

Good-Turing Smoothing

• Adjust actual counts \( c \) to expected counts \( c^{*} \) with formula

\[ c^{*} = (c + 1) \frac{N_{c} + 1}{N_{c}} \]

- \( N_{c} \) number of n-grams that occur exactly \( c \) times in corpus
- \( N_{0} \) total number of unseen n-grams

Good-Turing for 2-Grams in Europarl

<table>
<thead>
<tr>
<th>Count</th>
<th>Count of counts</th>
<th>Adjusted count ( c^{*} )</th>
<th>Test count ( t_{c} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7,514,941,065</td>
<td>0.00015</td>
<td>0.00016</td>
</tr>
<tr>
<td>1</td>
<td>1,132,844</td>
<td>0.46539</td>
<td>0.46235</td>
</tr>
<tr>
<td>2</td>
<td>263,611</td>
<td>1.40679</td>
<td>1.39946</td>
</tr>
<tr>
<td>3</td>
<td>123,615</td>
<td>2.38767</td>
<td>2.34307</td>
</tr>
<tr>
<td>4</td>
<td>73,788</td>
<td>3.33753</td>
<td>3.35202</td>
</tr>
<tr>
<td>5</td>
<td>49,254</td>
<td>4.36967</td>
<td>4.35234</td>
</tr>
<tr>
<td>6</td>
<td>35,869</td>
<td>5.32928</td>
<td>5.33762</td>
</tr>
<tr>
<td>8</td>
<td>21,693</td>
<td>7.43798</td>
<td>7.15074</td>
</tr>
<tr>
<td>10</td>
<td>14,880</td>
<td>9.31304</td>
<td>9.11927</td>
</tr>
<tr>
<td>20</td>
<td>4,546</td>
<td>19.54487</td>
<td>18.95948</td>
</tr>
</tbody>
</table>

\( t_{c} \) are average counts of n-grams in test set that occurred \( c \) times in corpus

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

\[ P_{\text{GT}} = \frac{1}{N_{0}} \frac{N_{1}}{n} \Rightarrow c^{*} = \frac{N_{1}}{N_{0}} \]
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

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

- Same thing for higher count items

Problems with Good-Turing

- Assumes we know the vocabulary size (no unseen words) [but see J&M 4.3.2]
- Doesn’t allow “holes” in the counts (if \( N_i > 0, N_{i-1} > 0 \)) [but see J&M 4.5.3]
- Applies discounts even to high-frequency items [but see J&M 4.5.3]
- Divides shifted probability mass evenly between all items of same frequency.