Introduction to Markov Models

Estimating the probability of phrases of words, sentences, etc.…

But first:
A few preliminaries on text preprocessing

What counts as a word? A tricky question....

Q1: How to estimate the probability of a given sentence $W$?

- A crucial step in speech recognition (and lots of other applications)
- First guess: bag of words: $\hat{P}(W) = \prod_{w \in W} P(w)$

Given word lattice:

<table>
<thead>
<tr>
<th>form</th>
<th>subsidy</th>
<th>for</th>
</tr>
</thead>
<tbody>
<tr>
<td>farm</td>
<td>subsidies</td>
<td>far</td>
</tr>
</tbody>
</table>

Unigram counts (in $1.7 \times 10^6$ words of AP text):

<table>
<thead>
<tr>
<th>form 183</th>
<th>subsidy 55</th>
<th>for 570</th>
</tr>
</thead>
<tbody>
<tr>
<td>farm 74</td>
<td>subsidy 15</td>
<td>for 18185</td>
</tr>
</tbody>
</table>

Most likely word string given $\hat{P}(W)$ isn’t quite right...

How to find Sentences??

The Obvious Heuristic For Sentence Boundaries:

/\?\{\*\}+/\n
But: I saw Mr. Jones visiting St. Peter’s basilica.
Patch: Delete the break after Mr. | Mrs. | Dr. | St. | Prof | … But:
- He left at 3 a.m. in the morning.
- He left at 3 a.m.
- In LISP, 2.0 and 2. stand for the same number.
Patch: Don’t break if the next word isn’t capitalized.
But: We saw Peter on Jones St. Peter’s brother was with him.

Predicting a word sequence II

- Next guess: products of bigrams
- For $W=w_1w_2w_3…w_n$, $\hat{P}(W) = \prod_{i=1}^{n} P(w_i | w_{i-1})$

Given word lattice:

<table>
<thead>
<tr>
<th>form</th>
<th>subsidy</th>
<th>for</th>
</tr>
</thead>
<tbody>
<tr>
<td>farm</td>
<td>subsidies</td>
<td>far</td>
</tr>
</tbody>
</table>

Bigram counts (in $1.7 \times 10^6$ words of AP text):

| form subsidy 0 | subsidy for 2 |
| form subsidies 0 | subsidy for 0 |
| farm subsidy 0 | subsidies for 6 |
| farm subsidies 4 | subsidies for 0 |

Much Better (if not quite right) …
(Q: the counts are our! Why?)
How can we estimate $P(W)$ correctly?

- Problem: Naïve Bayes model for bigrams violates independence assumptions.

Let’s do this right….

- Let $W = w_1 w_2 w_3 \ldots w_n$. Then, by the chain rule,
  
  $P(W) = P(w_1) \times P(w_2 | w_1) \times P(w_3 | w_1 w_2) \ldots \times P(w_n | w_1 \ldots w_{n-1})$

- We can estimate $P(w_i | w_{i-1})$ by the Maximum Likelihood Estimator
  
  $\frac{\text{Count}(w_i | w_{i-1})}{\text{Count}(w_{i-1})}$

  and so on…

and finally, Estimating $P(w_n | w_1 w_2 \ldots w_{n-1})$

Again, we can estimate $P(w_n | w_1 w_2 \ldots w_{n-1})$ with the MLE

$\frac{\text{Count}(w_n | w_1 \ldots w_{n-1})}{\text{Count}(w_1 \ldots w_{n-1})}$

So to decide pat vs. pot in “Heat up the oil in a large pot”, compute for pot

$\frac{\text{Count}("Heat up the oil in a large pot")}{\text{Count}("Heat up the oil in a large")}$

UNLESS OUR CORPUS IS REALLY HUGE

BOTH COUNTS WILL BE 0, yielding 0/0

The Web is HUGE!! (2016 version)

But what if we only have 100 million words for our estimates??

A BOTE Estimate of What We Can Estimate

What parameters can we estimate with 100 million words of training data??

Assuming (for now) uniform distribution over only 5000 words

<table>
<thead>
<tr>
<th>Event</th>
<th>Count</th>
<th>Estimate</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>words</td>
<td>5000</td>
<td>Excellent</td>
<td></td>
</tr>
<tr>
<td>word bigrams</td>
<td>25</td>
<td></td>
<td>OK</td>
</tr>
<tr>
<td>word trigrams</td>
<td>12.5</td>
<td></td>
<td>Terrible</td>
</tr>
</tbody>
</table>

So even with $10^9$ words of data, for even trigrams we encounter the sparse data problem.
The Markov Assumption: Only the Immediate Past Matters

The (First Order) Markov Assumption:
\[ P(w_i | w_1 \ldots w_{i-1}) = P(w_i | w_{i-1}) \]

Under this assumption, instead of
\[ P(W) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_2) \ldots P(w_n | w_{n-1}) \]
we estimate the probability of a string \( W \) by
\[ P(W) = P(w_1) \cdot P(w_2) \cdot P(w_3) \ldots P(w_n) \]

AND we have a correct practical estimation method for \( P(W) \) given the Markov assumption!

Visualizing an \( n \)-gram based language model: the Shannon/Miller/Selfridge method

Corpus: “the mouse ran up the clock. The spider ran up the waterspout.”
\( P(\text{the}) = 4/12 \quad P(\text{ran}) = 6/12 \quad P(\text{up}) = 8/12 \)
\( P(\text{mouse}) = P(\text{clock}) = P(\text{spider}) = P(\text{waterspout}) = 1/12 \)

Arbitrarily fix an order: \( w_1 = \text{the}, w_2 = \text{ran}, w_3 = \text{up}, w_4 = \text{mouse}, \ldots \)

CDF for a very small English corpus

Review (and crucial for upcoming homework): Cumulative Distribution Functions (CDFs)

The CDF of a random variable \( X \) is denoted by \( F_X(x) \) and is defined by
\[ F_X(x) = \Pr(X \leq x) \]

• \( F \) is monotonic nondecreasing: \( \forall x \leq y, F(x) \leq F(y) \)

• If \( X \) is a discrete random variable that attains values \( x_1, x_2, \ldots, x_k \) with probabilities \( p(x_1), p(x_2), \ldots \), then
\[ F_X(x_j) = \sum_{i=1}^{j} p(x_i) \]

Markov Models

A bigram can be viewed as a Markov model, a probabilistic FSA with

• \( S \), a set of states, one for each word \( w_i \) in the vocabulary.
• \( A \), a transition matrix where \( a(i,j) \) is the probability of going from state \( w_i \) to state \( w_j \).

The probability \( a(i,j) \) can be estimated by
\[ a(i,j) = \frac{\text{Count}(w_i, w_j)}{\text{Count}(w_i)} \]

• \( \Pi \), a vector of initial state probabilities, where \( \pi(i) \) is the probability of the first word being \( w_i \).

The Markov Assumption: Estimation

We estimate the probability of each \( w_i \) given previous context by
\[ P(w_i | w_1 w_2 \ldots w_{i-1}) = P(w_i | w_{i-1}) \]
which can be estimated by
\[ \frac{\text{Count}(w_i, w_{i-1})}{\text{Count}(w_{i-1})} \]

So we’re back to counting only unigrams and bigrams!!
Visualizing an n-gram based language model: the Shannon/Miller/Selfridge method

- To generate a sequence of \( n \) words given a 1st order Markov model (i.e., conditioned on one previous word):
  - Fix some ordering of the vocabulary \( v_1, v_2, v_3, \ldots, v_k \).
  - Use unigram method to generate an initial word \( w_1 \).
  - For each remaining position \( i, 2 \leq i \leq n \):
    - Choose a random value \( r_i \) between 0 and 1.
    - Choose \( w_i = \) the first \( v_j \) such that
      \[
      \sum_{m=1}^{j-1} P(v_m | w_{m-1}) \geq r_i.
      \]

The Shannon/Miller/Selfridge method trained on Shakespeare

Wall Street Journal just isn't Shakespeare

Shakespeare as corpus

- \( N=884,647 \) tokens, \( V=29,066 \)
- Shakespeare produced 300,000 bigram types out of \( V^2 = 844 \) million possible bigrams.
- So 99.96% of the possible bigrams were never seen (have zero entries in the table).
- Quadgrams worse: What's coming out looks like Shakespeare because it is Shakespeare

The Sparse Data Problem

English word frequencies well described by Zipf's Law

- Zipf (1949) characterized the relation between word frequency and rank as:
  \[
  f \cdot r = C \quad \text{(for constant } C)\]
  \[
  r = C/f \quad \text{and } \log(r) = \log(C) - \log(f) \]
- Purely Zipfian data plots as a straight line on a log-log scale
Exploiting Zipf to do Language ID

#The following filters out arabic words that are also frequent in Spanish and English...
arabic_top_12 = ['7ata', 'ana', 'ma', 'w', 'bs', 'fe', 'b3d', '3adou', 'mn', 'kan', 'men', 'ahmed']

#The following filters out urdu words common in English
urdu_top_17 = ['hai', 'ko', 'ki', 'main', 'na', 'se', 'ho', 'bhi', 'mein', 'ka', 'lum', 'nahi', 'men', 'jo', 'wo', 'dil', 'hain']

spanish_top_16 = ['de', 'la', 'que', 'el', 'en', 'y', 'es', 'un', 'los', 'por', 'se', 'para', 'con']

english_top_20 = ['the', 'to', 'of', 'in', 'i', 'a', 'is', 'and', 'you', 'for', 'on', 'it', 'that', 'are', 'with', 'am', 'my', 'be', 'at', 'not', 'we']

All the code you need....

```python
# Get best language as string: lid_pick_best(lid_process_tweet(tweet))
counts = collections.Counter()
def lid_process_tweet(tweet):
counts.clear()
for word in re.split(r'\[.\?!,\]s+', tweet.encode('ascii', 'replace').strip().lower()):
    if not re.match(r'http://', word):
        for lang in languages:
            if word in topwords[lang]:
                #("english", "arabic", . . .)
                counts[lang] += 1  # dict of word lists indexed by lang

return counts.most_common()

def lid_pick_best(count_list):
    if count_list:
        return count_list[0][0]
    else:
        return 'UNKNOWN'
```

Lots of area under the tail of this curve!