Week 3: Language Modeling I

In this week's section, we'll explore basic techniques for language modeling.

We'll first dig into word distributions to illustrate the sparsity problem and the difficulty of prediction.

Then, we'll look at a simple N-gram language model and generate some sample sentences. In the homework, you'll extend this by implementing smoothing techniques that improve performance, and perform some more detailed analysis of the model.

Note: if you're viewing this in-browser, jump over to the NBViewer link. This will get around GitHub's JavaScript block and properly render the plots:
http://nbviewer.jupyter.org/github/datasci-w266/2018-spring-main/blob/master/materials/simple_lm/lm1.ipynb (http://nbviewer.jupyter.org/github/datasci-w266/2018-spring-main/blob/master/materials/simple_lm/lm1.ipynb)

We'll use Bokeh (https://bokeh.pydata.org/en/latest/docs/user_guide/quickstart.html) for plotting some data in this notebook. It's similar to `matplotlib`, but renders JavaScript-based plots that support more interactive control.

In [5]:
```
# Standard python helper libraries.
from __future__ import print_function
from __future__ import division
import os, sys, time
import collections
import itertools

# Numerical manipulation libraries.
import numpy as np
from scipy import stats, optimize

# NLTK is the Natural Language Toolkit, and contains several language datasets
# as well as implementations of many popular NLP algorithms.
# HINT: You should look at what is available here when thinking about your project!
import nltk

# Helper libraries (see the corresponding py files in this notebook's directory).
from common import utils, vocabulary
import segment

utils.require_package("tqdm")  # for nice progress bars
from tqdm import tqdm as ProgressBar

# Bokeh for plotting.
utils.require_package("bokeh")
import bokeh.plotting as bp
from bokeh.models import HoverTool
bp.output_notebook()
```

(https://bokeh.pydata.org) Bokeh 0.12.13 successfully loaded.

Corpus Statistics

NLTK includes a number of corpora that we can experiment with for this exercise. Different types of text can have very different N-gram distributions, and some are more difficult to model than others.

Let's start with the Brown corpus (http://www.essex.ac.uk/linguistics/external/clmt/w3c/corpus_ling/content/corpora/list/private/brown/brown.html), the first major computer-readable linguistic corpus. It consists of around 1 million words of American English, sampled from 15 different text categories ranging from news text to academic articles to popular fiction.

If you haven't yet run `nltk.download()`, the cell below will download the Brown corpus for you.

In [6]:
```
nltk.download('brown')
```

```
[nltk_data] Downloading package brown to
[nltk_data]     C:\Users\Nephila\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping corpora\brown.zip.
```

Out[6]: True

Let's start by looking at the all the words in the corpus, and looking at some basic statistics.

We've built a helper class, `Vocabulary`, that ingests a list of words, counts their frequencies, and assigns each one a numerical ID that will be useful later on.

In [7]:
```
corpus = nltk.corpus.brown

# "canonicalize_word" performs a few tweaks to the token stream of
# the corpus.  For example, it replaces digits with DG allowing numbers
# to aggregate together when we count them below.
# You can read the details in utils.py if you're really curious.
token_feed = (utils.canonicalize_word(w) for w in corpus.words())

# Collect counts of tokens and assign wordids.
vocab = vocabulary.Vocabulary(token_feed, progressbar=ProgressBar)
print("Vocabulary size: {:,}".format(vocab.size))

# Print out some (debugging) statistics to make sure everything went
# as we expected.  (Unsurprisingly, you should see "the" as the most popular word.)
print("Most common unigrams:")
for word, count in vocab.unigram_counts.most_common(10):
    print("\"{:s}\": {:,}".format(word, count))
```

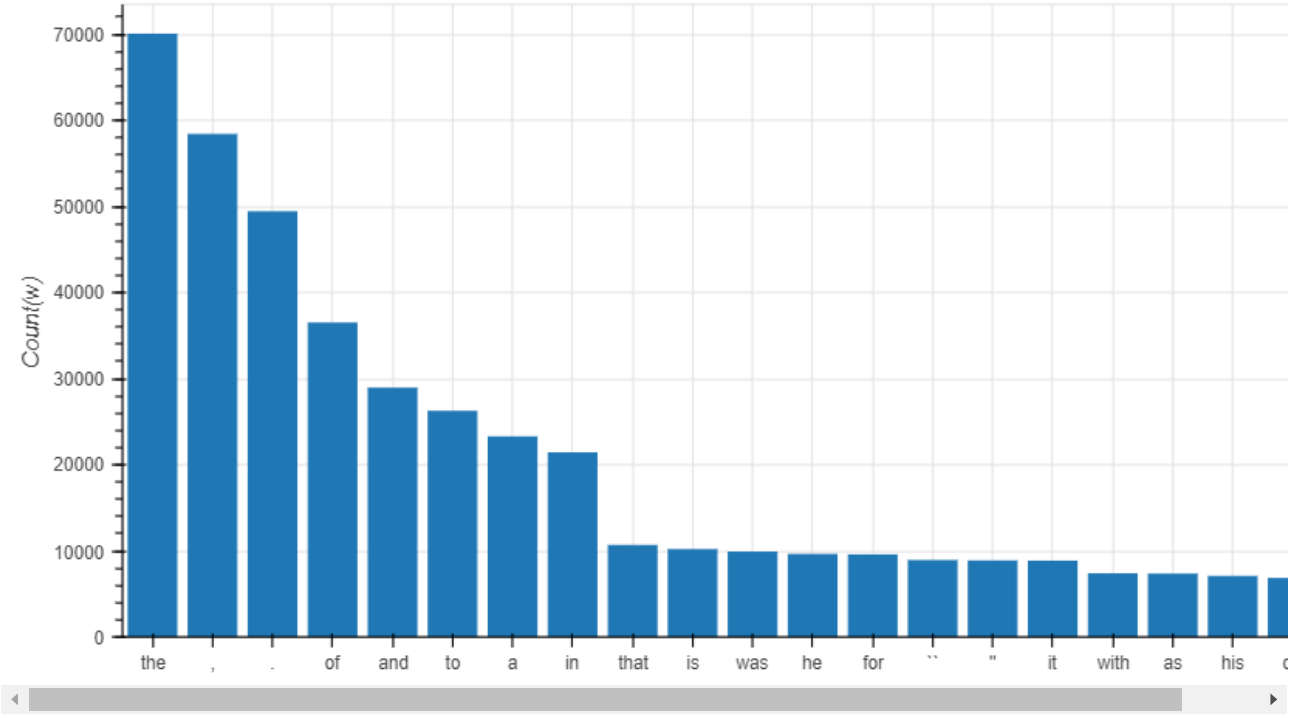
```
1161192it [00:14, 80298.16it/s]

Vocabulary size: 48,174
Most common unigrams:
"the": 69,971
",": 58,334
".": 49,346
"of": 36,412
"and": 28,853
"to": 26,158
"a": 23,195
"in": 21,337
"that": 10,594
"is": 10,109
```

`Vocabulary.unigram_counts` is a dictionary (actually, `collections.Counter`) of the unigram frequencies $c(w)$. Let's look at a plot of the top frequencies:

In [8]:
```
words, counts = zip(*vocab.unigram_counts.most_common(20))

hover = HoverTool(tooltips=[("word", "@x"), ("count", "@top")], mode="vline")
fig = bp.figure(x_range=words, plot_width=800, plot_height=400, tools=[hover])
fig.vbar(x=words, width=0.8, top=counts, hover_fill_color="firebrick")
fig.y_range.start = 0
fig.yaxis.axis_label = "Count(w)"
bp.show(fig)
```



You'll notice that it falls off very quickly! It also flattens out a lot after the initial dip. Recall that word frequencies tend to follow **Zipf's law**, in that they are rougly proportional to $\frac{1}{\text{rank}(w)}$, where rank = 1 for the most common word, 2 for the second-most, etc. We can test this directly with a numerical fit:

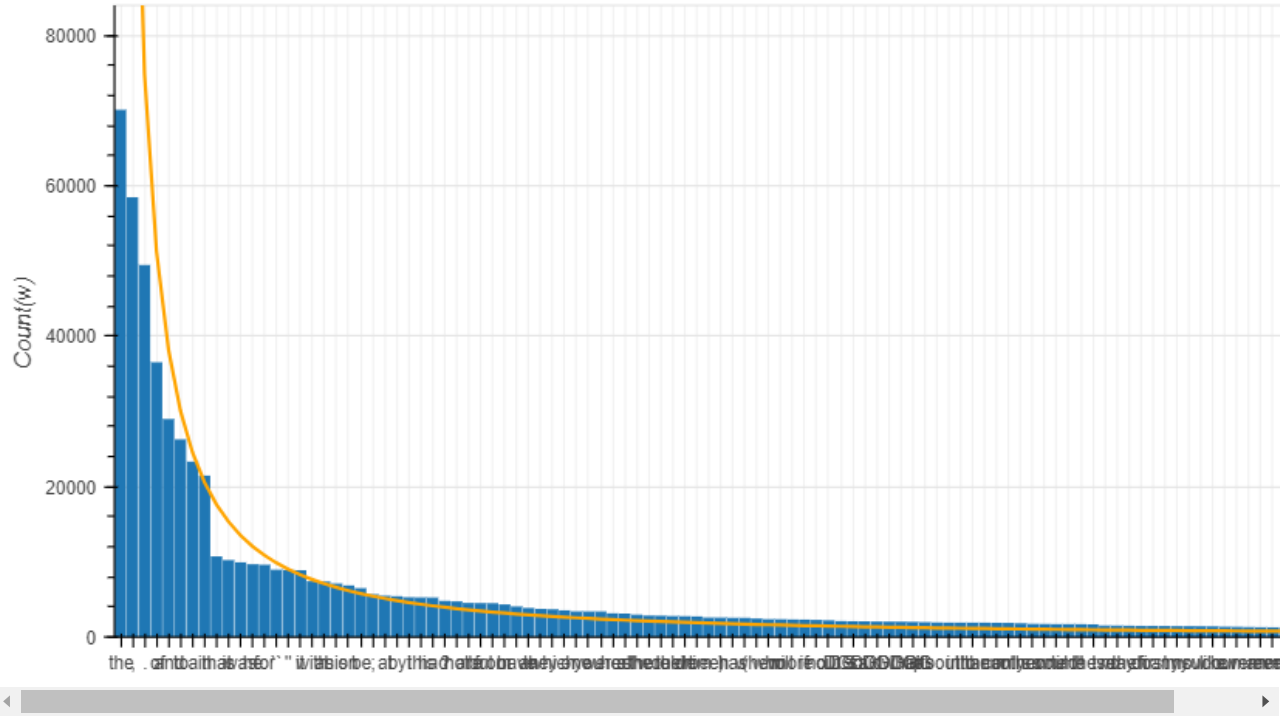
In [9]:
```
# This next line splits the pairs of <word, count> in the vocabulary into two lists:
# 1.  a list of words (types)
# 2.  a list of counts (per type)
# with the property that the ith word in the list has its corresponding count in the ith counts.
words, counts = zip(*vocab.unigram_counts.most_common(vocab.size))
counts = np.array(counts, dtype=float)  # Avoid integer math.
rank = 1 + np.arange(len(counts))  # rank is an array of [1, 2, 3, 4, ..., num_types]
N = np.sum(counts)  # N = total # of tokens seen.
p = counts / N  # p is an array the length of `words`.  #_times_word_seen / total_#_words

# Fit a power law curve to the histogram above.
# Optimize least-squares in log space.
# See http://nlp.stanford.edu/IR-book/html/htmledition/zipfs-law-modeling-the-distribution-of-term
fit_func = lambda c: (np.log(c[0]*p) - np.log(c[0] * rank**c[1]))
(a,b), _ = optimize.leastsq(fit_func, np.array([p[0], -1.0]))
print(u"Power law exponent: \u03B2 = {:.02f}".format(b))
p_pred = (a * rank**b) / sum(a * rank**b)  # predict probabilities
c_pred = N * p_pred  # predict counts


# Plot counts, with fit curve.
nplot = 100
fig = bp.figure(x_range=words[:nplot], plot_width=800, plot_height=400)
bars = fig.vbar(x=words[:nplot], width=0.8, top=counts[:nplot], hover_fill_color="firebrick")
fig.add_tools(HoverTool(tooltips=[("word", "@x"), ("count", "@top")], renderers=[bars], mode="vlin
fig.line(x=words[:nplot], y=np.round(c_pred)[:nplot], color="Orange", line_width=2)
fig.y_range.start = 0
fig.y_range.end = 1.2*max(counts)
fig.yaxis.axis_label = "Count(w)"
fig.xgrid.grid_line_alpha = 0.5
bp.show(fig)
```

```
C:\Users\Nephila\Anaconda3\lib\site-packages\ipykernel_launcher.py:14: RuntimeWarning: invalid v
alue encountered in log
```

```
Power law exponent: β = -1.32
```



Why should we care about the form of this distribution? Power-law distributions like Zipf's law (https://en.wikipedia.org/wiki/Zipf%E2%80%93Mandelbrot_law) have a **long tail**, which means that a large fraction of the

tokens (words on the page) belong to types (words in the dictionary) that appear quite rarely.

We can look at this directly by plotting the cumulative distribution function, *as a function of the count* $c(w)$:

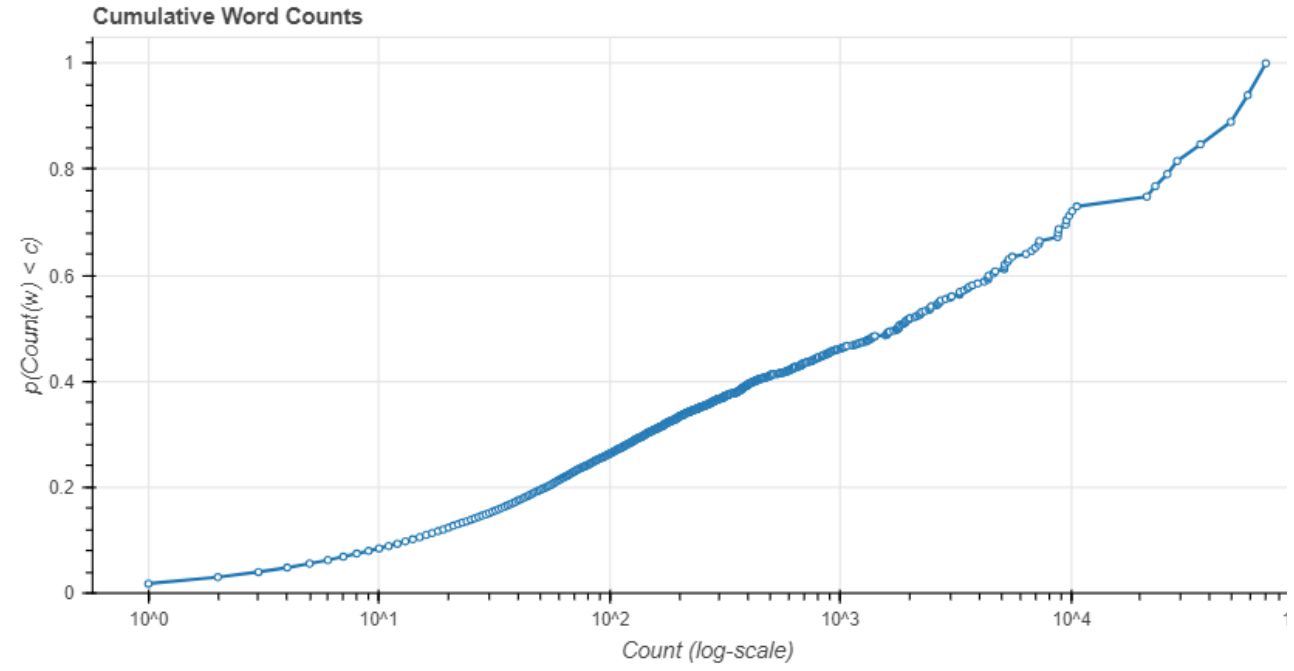
$$f(k) = \sum_{i \in \text{words}}^{|\text{corpus}|} \mathbf{1}\{c(w_i) <= k\} = \sum_{j \in \text{types}}^{|V|} c(w_j)\, \mathbf{1}\{c(w_j) <= k\}$$

This asks: for a given word on the page, how likely is $c(w) <= k$?

The Brown corpus has about 1 million words (tokens), about 7% of which are the token "the" and 6% of which are commas. So, we'll plot our data on a log scale so that we can see what's going on for $k = 1, \ldots, 100$, as well as the higher counts ($c(\text{the}) = 6.9 \cdot 10^4$).

In [10]:
```
# We'll use the histogram function with variable bins in order to get a stair-step plot.
b_shift = 0.5  # So counts don't fall on bin boundaries.
# Weights give us distribution by token; remove this to get distribution by type.
h, bins = np.histogram(counts, weights=counts, bins=b_shift+np.concatenate([[0], np.unique(counts)

fig = bp.figure(plot_width=800, plot_height=400, x_axis_type="log", title="Cumulative Word Counts"
l = fig.line(x=bins[1:]-b_shift, y=np.cumsum(1.0*h)/np.sum(h), line_width=2)
fig.circle(x=bins[1:]-b_shift, y=np.cumsum(1.0*h)/np.sum(h), fill_color="white", size=4)
fig.add_tools(HoverTool(tooltips=[("count", "@x"), ("CDF", "@y")], renderers=[l], mode="vline"))
fig.y_range.start = 0
fig.yaxis.axis_label = "p(Count(w) < c)"
fig.xaxis.axis_label = "Count (log-scale)"
bp.show(fig)
```



Exercise: the cell above plots the distribution based on *tokens*, or words on the page. Modify the histogram code to make a similar plot, but based on *types*, or distinct words in the vocabulary. If we pick a random word in the dictionary, how many times are we likely to see it in a 1M word corpus?

Exercise (more involved): compute the same function for the bigram and trigram distributions, and overlay it on a version of the plot above. (This should be similar to a graph from the async.) What does this tell you about the sparsity problem?

Language Modeling

With a basic idea of what our corpus looks like, we can now embark on the task of modeling it. Recall from the async that language modeling is the task of predicting the next word, given the preceding history:

$$P(w_i|w_{i-1}, \ldots, w_0)$$

Perplexity

How hard is this task? Let's use the unigram distribution as a starting point for our model. We saw that the distribution is very uneven - some words (types) are very common, while others are very rare. If the corpus (or a sample thereof) is our test set, we're going to see our training labels according to the distribution $y \sim p(w)$. A smarter model might take some features such as immediately previous words and do a better job estimating the next word (let's call such features x in general). We can summarize how uncertain our model is by looking at the entropy:

In [11]:
```
print("Unigram entropy: {:.03f} bits".format(stats.entropy(p, base=2)))
```

```
Unigram entropy: 10.066 bits
```

Bits are still a little unintuitive, but we can do better. Recall from Assignment 1 that the entropy of a *uniform* distribution over n elements is $\log_2 n$. So given some distribution with entropy $H(P) = k$ bits we can say that distribution P is *as uncertain as* a uniform distribution over 2^k elements.

If we apply this to our unigram distribution, we have:

In [12]:
```
print("Distinct unigrams: {:d}".format(len(p)))
print("As uncertain as:   {:.02f}".format(2**stats.entropy(p, base=2)))
```

```
Distinct unigrams: 48171
As uncertain as:   1071.78
```

Now in the (real) machine learning setting, we can't really measure the entropy of the underlying distribution $P(y \mid x) = P(w_i|w_{i-1}, \ldots, w_0)$. But, we can approximate the cross-entropy between the true distribution (for which we have samples $y_i \sim P(y \mid x_i)$) and our *predicted* distribution $\hat{P}(y \mid x_i)$:

$$\mathrm{CE}(P, \hat{P}) = -\sum_{y,x} P(y \mid x) \log_2 \hat{P}(y \mid x) \approx -\frac{1}{N} \sum_{i}^{N} \log_2 \hat{P}(y_i \mid x_i)$$

This has the same units (bits), and we can exponentiate it in the same way to give us a measure of "how confused" the model is. This quantity is the **perplexity** of the model. A perplexity of k tells us that the model is as uncertain as if it had to choose from k elements with equal probability.

So suppose we used the unigram model as our language model. We'll cheat for now and just use $\hat{P}_{\text{unigram}}(w) = \tilde{p}(w)$ (where $\tilde{p}$ denotes an estimate from a finite sample). Then our (training set) perplexity is:

In [13]:
```
# scipy.stats.entropy with two arguments
def cross_entropy(p, q):
    return -1*np.sum(p * np.log2(q))

print("Unigram language model")
print("Expected full-corpus perplexity: {:.02f}".format(2**cross_entropy(p, q=p)))
```

```
Unigram language model
Expected full-corpus perplexity: 1071.78
```

If we want to sanity-check it, we can do our usual machine learning loss calculation: look at all the examples (words), and take the average loss:

In [14]:
```
word_to_prob = {w:p[i] for i,w in enumerate(words)}
probs = [word_to_prob[utils.canonicalize_word(w)]
         for w in ProgressBar(corpus.words())]
perplexity = 2**(-1*np.sum(np.log2(probs))/len(probs))
print("Perplexity (full-corpus): {:.02f}".format(perplexity))
```

```
100%|████████████████████████████████████████████████████████████| 1161192/1161192 [00:1
0<00:00, 111204.00it/s]

Perplexity (full-corpus): 1071.78
```

In the two cells below, we test out our simple model by taking a piece of text with all the spaces removed, enumerating all possible ways of splitting it (efficiently using dynamic programming - more on that later in the course), scoring each, and then returning the highest scoring sequence.

You can see we can apply even this **really** simple model to a neat application[1]!

[1] Peter Norvig, a director of research at Google implemented this model here: http://norvig.com/ngrams/ch14.pdf (http://norvig.com/ngrams/ch14.pdf) and extends it pretty far. By the end, he has managed to decrypt WWI encryption using only this simple language model!

In [15]:
```
segment.segment('hellotherehowareyou', vocab)
```

Out[15]:
```
['hello', 'there', 'how', 'are', 'you']
```

In [16]:
```
segment.segment('danceajig', vocab)
```

Out[16]:
```
['dance', 'a', 'jig']
```

In [17]:
```
segment.segment('thecatinthehat', vocab)
```

Out[17]:
```
['the', 'cat', 'in', 'the', 'hat']
```

N-Gram Language Models

The unigram model isn't a very good one - it doesn't model any previous context at all. On the other hand, we can't model *all* of the preceding words, because that history could get prohibitively long and extremely sparse. As a compromise, we'll make a *Markov assumption* and limit ourselves to a finite history of n words.

For now, we'll build a trigram model, which considers the two preceding words:

$$P(w_i \mid w_{i-1}, \ldots, w_0) \approx P(w_i \mid w_{i-1}, w_{i-2})$$

Exercise: how many possible trigrams are there?

We'll need to store a table of the probabilities, indexed by triples (w_i, w_{i-1}, w_{i-2}). If we store every possible combination, it can get quite large. Assuming 8 bytes for each entry, we have:

In [21]:
```
import psutil
print("Vocab size:     {:10,} words".format(vocab.size))
print("Unigrams need:  {:12.2f} kB".format(8 * (vocab.size ** 1) / (2**10)))
print("Bigrams  need:  {:12.2f} MB".format(8 * (vocab.size ** 2) / (2**20)))
print("Trigrams need:  {:12.2f} MB".format(8 * (vocab.size ** 3) / (2**20)))
print("Available:      {:12.2f} MB".format(psutil.virtual_memory().available / (2**20)))
```

```
Vocab size:         48,174 words
Unigrams need:        376.36 kB
Bigrams  need:      17705.80 MB
Trigrams need:  852959083.65 MB
Available:          1715.93 MB
```

Well, that's no good. We can store all possible unigrams, but after that we're toast.

Thankfully, we don't have to store everything. Recall that most words only appear a handful of times, and that there are plenty of words in the English language that we don't see in our corpus at all. For bigrams and trigrams, the table will be quite sparse. We need only store the entries that we actually observe; the rest we can take to be zero, or estimate their values on the fly through smoothing or backoff.

Observation: When building language models, be sure to only keep non-zero counts in your datastructures and assume anything missing is 0.

Exercise: for a corpus of 1 million words and a vocabulary of 10000, what is the maximum number of bigrams that can be actually *observed*? How about trigrams?

Constructing our Model

We'll represent our model with a nested map `context => word => probability`, where word is w_i and for our trigram model, the context is the two preceding words (w_{i-2}, w_{i-1}).

First, we'll go through the corpus and compute raw trigram counts $c(abc)$, which we'll then normalize into probabilities:

$$P_{abc} = P(w_i = c \mid w_{i-1} = b, w_{i-2} = a) = \frac{\text{c(abc)}}{\sum_{c'} \text{c(abc}')} = \frac{C_{abc}}{\sum_{c'} C_{abc'}}$$

Here's the code for our model:

In [22]:
```
from collections import defaultdict

hi = defaultdict(lambda: defaultdict(lambda: 0.0))
hi[('alice', 'lam')]['sister'] += 1
hi[('alice', 'lam')]['chibo'] += 5
hi[('houston','lam')]['brother'] += 1
```

In [23]:
```
hello = defaultdict(lambda: set())
```

In [24]:
```
hello.update({'chibo':('alice','lam')})
```

In [25]:
```
hello.update({'chibo':('awesome')})
```

In [30]:
```
hello
```

Out[30]:
```
defaultdict(<function __main__.<lambda>>, {'chibo': 'awesome'})
```

In [31]:
```
k = 2
```

In [32]:
```
for context1, ctr1 in hi.items():
    print (context1)
    print (ctr1)
    print (ctr1.values())
    print (len(ctr1.values()))
    print (sum(ctr1.values()))
    for word in ctr1.keys():
        print (word)
```

```
('alice', 'lam')
defaultdict(<function <lambda>.<locals>.<lambda> at 0x0000023E48890598>, {'sister': 1.0, 'chib
o': 5.0})
dict_values([1.0, 5.0])
2
6.0
sister
chibo
('houston', 'lam')
defaultdict(<function <lambda>.<locals>.<lambda> at 0x0000023E488907B8>, {'brother': 1.0})
dict_values([1.0])
1
1.0
brother
```

In [33]:
```
len(hi['alice','lam'].items())
```

Out[33]:
```
2
```

In [34]:
```
context1, ctr1 = hi['alice','lam']
```

In [35]:
```
ctr1
```

Out[35]:
```
'chibo'
```

In [52]:
```
hi['alice','lam'].values()
```

Out[52]:
```
0.375
```

In [54]:
```
(hi['alice','lam'].get('sister', 0.0) + k)  / (sum(hi['alice','lam'].values()) + k * 2)
```

Out[54]:
```
0.3
```

In [37]:
```
for context, ctr in hi.items():
#    print (context)
    print(ctr)
    print (sum(ctr.values()))
#    print (sum(ctr.items()))
```

```
defaultdict(<function <lambda>.<locals>.<lambda> at 0x0000023E48890598>, {'sister': 1.0, 'chib
o': 5.0})
6.0
defaultdict(<function <lambda>.<locals>.<lambda> at 0x0000023E488907B8>, {'brother': 1.0})
1.0
```

In [38]:
```
from collections import defaultdict

def normalize_counter(c):
    """Given a dictionary of <item, counts>, return <item, fraction>."""
    total = sum(c.values())
    return {w:float(c[w])/total for w in c}

class SimpleTrigramLM(object):
    def __init__(self, words):
        """Build our simple trigram model."""
        # Raw trigram counts over the corpus.
        # c(w | w_1 w_2) = self.counts[(w_2,w_1)][w]
        self.counts = defaultdict(lambda: defaultdict(lambda: 0.0))

        # Iterate through the word stream once.
        w_1, w_2 = None, None
        for word in words:
            if w_1 is not None and w_2 is not None:
                # Increment trigram count.
                self.counts[(w_2,w_1)][word] += 1
            # Shift context along the stream of words.
            w_2 = w_1
            w_1 = word

        # Normalize so that for each context we have a valid probability
        # distribution (i.e. adds up to 1.0) of possible next tokens.
        self.probas = defaultdict(lambda: defaultdict(lambda: 0.0))
        for context, ctr in self.counts.items():
            self.probas[context] = normalize_counter(ctr)

    def next_word_proba(self, word, seq):
        """Compute p(word | seq)"""
        context = tuple(seq[-2:])  # last two words
        return self.probas[context].get(word, 0.0)

    def predict_next(self, seq):
        """Sample a word from the conditional distribution."""
        context = tuple(seq[-2:])  # last two words
        pc = self.probas[context]  # conditional distribution
        words, probs = zip(*pc.items())  # convert to list
        return np.random.choice(words, p=probs)

    def score_seq(self, seq, verbose=False):
        """Compute log probability (base 2) of the given sequence."""
        score = 0.0
        count = 0
        # Start at third word, since we need a full context.
        for i in range(2, len(seq)):
            if (seq[i] == "<s>" or seq[i] == "</s>"):
                continue  # Don't count special tokens in score.
            s = np.log2(self.next_word_proba(seq[i], seq[i-2:i]))
            score += s
            count += 1
            # DEBUG
            if verbose:
                print("log P({:s} | {:s}) = {.03f}".format(seq[i], " ".join(seq[i-2:i]), s))
        return score, count
```

Training

Let's train our model. We'll do a proper train-test split this time, so we can evaluate on unseen data. This means that we also have to fix our vocabulary based on the *training* set - which means that some unseen words in the test set will get replaced by `<unk>`. The `Vocabulary` helper class will take care of this for us.

In [39]:
```
train_sents, test_sents = utils.get_train_test_sents(corpus, split=0.8, shuffle=True)
```

```
Loaded 57,340 sentences (1.16119e+06 tokens)
Training set: 45,872 sentences (929,539 tokens)
Test set: 11,468 sentences (231,653 tokens)
```

In [40]:
```
vocab = vocabulary.Vocabulary(utils.canonicalize_word(w) for w in ProgressBar(utils.flatten(train_
print("Train set vocabulary: %d words" % vocab.size)
```

```
100%|████████████████████████████████████████████████████████████| 929539/929539 [00:0
5<00:00, 158868.46it/s]

Train set vocabulary: 43707 words
```

A note on Preprocessing

We didn't do this for the unigram LM, but when modeling sentences it's helpful to add special beginning-of-sentence (`<s>`) and end-of-sentence (`</s>`) tokens.

This lets the model estimate the probability of a word appearing at the beginning of a sentence, and lets it model the end of a sentence properly, since periods or other punctuation aren't always an accurate guide (e.g. `"Dr."` or `"Yahoo!"`).

Our padded sentences will look like this:

```
<s> <s> the cat sat in the hat . </s>
```

Exercise: why do we add two `<s>` tokens at the beginning? (*Hint: this is specific to our trigram model.*)

In [41]:
```
def sents_to_tokens(sents):
    """Returns an flattened list of the words in the sentences, with padding for a trigram model."
    padded_sentences = (["<s>", "<s>"] + s + ["</s>"] for s in sents)
    # This will canonicalize words, and replace anything not in vocab with <unk>
    return np.array([utils.canonicalize_word(w, wordset=vocab.wordset)
                     for w in ProgressBar(utils.flatten(padded_sentences))], dtype=object)

train_tokens = sents_to_tokens(train_sents)
test_tokens = sents_to_tokens(test_sents)

t0 = time.time()
print("Building trigram LM...",)
lm = SimpleTrigramLM(train_tokens)
print("done in %.02f s" % (time.time() - t0))
```

```
100%|████████████████████████████████████████████████████████████| 1067155/1067155 [00:0
1<00:00, 661189.39it/s]
100%|████████████████████████████████████████████████████████████| 266057/266057 [00:0
0<00:00, 591244.92it/s]

Building trigram LM...
done in 4.94 s
```

Sampling Sentences

Before we run scoring, let's look at some generated sentences from our model. We'll generate them sequentially, one token at a time:

In [42]:
```
max_length = 20
num_sentences = 5

for _ in range(num_sentences):
    seq = ["<s>", "<s>"]
    for i in range(max_length):
        seq.append(lm.predict_next(seq))
        # Stop at end-of-sentence
        if seq[-1] == "</s>": break
    print(" ".join(seq))
    print("[{1:d} tokens; log P(seq): {0:.02f}]".format(*lm.score_seq(seq)))
    print("")
```

```
<s> <s> and , therefore , it could be shown to undergo a contraction which is related to a minim
um , short
[20 tokens; log P(seq): -65.06]

<s> <s> london explains that the organic active . </s>
[7 tokens; log P(seq): -30.36]

<s> <s> the strange darkness , was intrigued by the progressive continuity to sensory impression
s . </s>
[14 tokens; log P(seq): -43.46]

<s> <s> he hadn't decided between mizell and vern law for the lambert tree award and the door .
</s>
[17 tokens; log P(seq): -40.76]

<s> <s> there were many cures for warts . </s>
[7 tokens; log P(seq): -16.91]
```

Scoring our model

We'll score our model by running the `score_seq` function, which computes

$$\mathrm{CE}_{\text{total}}(y, \hat{y}) = \sum_{i=2}^{N} \log_2 \hat{p}(w_i \mid w_{i-1}, w_{i-2})$$

This is the cross-entropy loss, which is equal to -1 times the log-likelihood of the data under our model. As usual, we'll exponentiate to get the perplexity score:

In [43]:
```
log_p_data, num_real_tokens = lm.score_seq(train_tokens)
print("Train perplexity: {:.02f}".format(2**(-1*log_p_data/num_real_tokens)))

log_p_data, num_real_tokens = lm.score_seq(test_tokens)
print("Test perplexity: {:.02f}".format(2**(-1*log_p_data/num_real_tokens)))
```

```
Train perplexity: 7.44

C:\Users\Nephila\Anaconda3\lib\site-packages\ipykernel_launcher.py:51: RuntimeWarning: divide by
zero encountered in log2

Test perplexity: inf
```

What's going on here? Our model gets an absurdly low perplexity on the training data, but a perplexity of *infinity* on the test data.

Answer: the n-gram model badly overfits without any smoothing.

Smoothing and Handling the Unknown

Our simple model doesn't have any real mechanism for handling unknown words - if we feed something unseen to `score_seq`, it will assign it a probability of zero:

In [44]:
```
lm.score_seq(["<s>", "i", "love", "w266", "</s>"])[0]
```

```
C:\Users\Nephila\Anaconda3\lib\site-packages\ipykernel_launcher.py:51: RuntimeWarning: divide by
zero encountered in log2
```

Out[44]:
```
-inf
```

This is the cause of the infinite perplexity, above: $\log_2 0 = -\infty$.

Exercise: besides unknown words, when else would the un-smoothed trigram model predict $p(w \mid w_{i-1}, w_{i-2}) = 0$?

Is assuming zero probabilities realistic? Let's look back at our unigram distribution:

In [45]:
```
print("% words seen only once: {:.02%}".format(sum(counts * (counts == 1)) / sum(counts)))
```

```
% words seen only once: 1.80%
```

About 1 in 50 words were seen only once in our corpus, so we might expect a comparable fraction of words in a new sample to also be previously-unseen:

In [46]:
```
print("% <unk> in test set: {:.02%}".format(np.sum(np.array(test_tokens) == "<unk>") / len(test_to
```

```
% <unk> in test set: 1.83%
```

If we want to use our language model in the wild, we'll need to implement some kind of smoothing to hedge our bets whenever these come up. This will be a major focus of Assignment 2, in which you'll build on the `SimpleTrigramLM` by implementing Laplace (add-k) and Kneser-Ney smoothing.