Part 1: n-gram Language Modeling

SOLUTIONS - do not distribute!

In this part of the assignment, we'll expand on the `SimpleTrigramLM` from the live session demo. We'll add smoothing to improve performance on unseen data, and explore some of the properties of the smoothed model.

If you haven't looked over the simple trigram LM in lm1.ipynb (../../../materials/simple_lm/lm1.ipynb), we recommend familiarizing yourself with it. This assignment will use a very similar set-up.

Add-k Smoothing

Recall our unsmoothed maximum likelihood estimate of $P(w_i \mid w_{i-1}, w_{i-2})$ where we use the raw distribution over words seen in a context in the training data:

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

Add-k smoothing is the simple refinement where we add $k > 0$ to each count C_{abc}, pretending we've seen every vocabulary word k extra times in each context. So we have:

$$\hat{P}_k(w_i = c \mid w_{i-1} = b, w_{i-2} = a) = \frac{C_{abc} + k}{\sum_{c'}(C_{abc'} + k)} = \frac{C_{abc} + k}{C_{ab} + k \cdot |V|}$$

where $|V|$ is the size of our vocabulary.

In the questions below and in the code, we'll refer to (w_{i-2}, w_{i-1}) as the *context*, and w_i as the current *word*. By convention, we'll somewhat interchangeably refer to the sequence (w_{i-2}, w_{i-1}, w) as abc.

Part (a): Short answer questions

Give brief answers to the following, in the cell below.

1. If we encounter a new context `(foo, bar)` unseen in the training data, what will the predicted *distribution* $\hat{P}_k(w \mid \text{foo}, \text{bar})$ be? How does your answer depend on k?
2. Is this a good estimate, or can we do better?
3. If we encounter a new word in a familiar context (i.e. `ab` is in the corpus, but `abq` is not), what will our predicted probability $\hat{P}_k(q \mid b, a)$ be? Give your answer in terms of C_{ab}, k, and $|V|$.
4. Based on your answer to question 3, in which context will your model predict a higher probability of *any* unknown word?
   Context (a): `<s> the ___`
   Context (b): `Mister Rogers ___`
   Assume $C_{\text{<s>, the}} = 10000$ and $C_{\text{Mister, Rogers}} = 47$.
5. Based on your knowledge of language, which of the contexts from question 4 *should* have a higher probability of an unknown word?

Answers for Part (a)

SOLUTIONS - do not distribute!

Please keep answers brief (1-2 lines).

Hint: You can use LaTeX to typeset math, e.g. `$ f(x) = x^2 $` will render as $f(x) = x^2$.

1. For a new (unseen) context, we have $C_{foo,bar} = 0$ *and* $C_{foo,bar,c} = 0$. So our smoothing equation reduces to:
$$\hat{P}_k(w = c \mid w_{i-1} = bar, w_{i-2} = foo) = \frac{C_{abc} + k}{C_{ab} + k \cdot |V|} = \frac{0 + k}{0 + k \cdot |V|} = \frac{1}{|V|}$$
This is a uniform distribution over the vocabulary, and the probability does not depend either on k or on the target word w.

2. This isn't a very good estimate! We have plenty of reason to believe that a new word would not follow the uniform distribution. For one: the distribution of unigrams follows Zipf's Law, which is very much non-uniform (see week 2 notebook). At minimum, we should be able to fall back to bigram or unigram counts, or even use something more sophisticated like type fertility.
3. Substituting $C_{abq} = 0$ into the equation for $\hat{P}_k$, we get:
$$\hat{P}_k(q| \, b, a) = \frac{k}{C_{ab} + k \cdot |V|}$$
Notice that the context count C_{ab} is in the denominator, which means the more times that we see the bigram ab, the lower the probability we attribute to a new trigram abq.

   Also note that C_{ab} and $k \cdot |V|$ combine *additively*. So if we have a not-unreasonable $V = 10000$ and $k = 0.1$, the smoothing term actually dominates for all contexts ab with counts $C_{ab} < 1000$ - quite a high bar! Clearly, this smoothing is very heavy-handed.

4. From the answer to (3), we see that the probability of an unknown word is higher for small C_{ab}. So, our smoothed model would predict unknown words more often for `Mister Rogers` ____ than for `<s> the` ____.

5. The behavior in (4) isn't ideal. Intuitively, you'd expect that `Mister Rogers` ____ will usually be completed by `"Neighborhood"` - whereas `<s> the` ____ could be completed by just about any noun. The Kneser-Ney smoothing model will help address this problem both through more careful smoothing, and the use of a type fertility model.

Part (b): Implementing the Add-k Model

Despite its shortcomings, it's worth implementing an add-k model as a baseline. Unlike the unsmoothed model, we'll be able to get some reasonable (or at least, finite) perplexity numbers which we can compare to the Kneser-Ney model below.

We've provided some skeleton code (similar to lm1.ipynb (../../../materials/simple_lm/lm1.ipynb)) in the `ngram_lm.py` file. In the `AddKTrigramLM` class, implement the following:

- `__init__(self, words)`, which computes the necessary corpus statistics C_{abc} and C_{ab}.
- `next_word_proba(self, word, seq, k)`, which computes $\hat{P}_k(w \mid w_{i-1}, w_{i-2})$

See the function docstrings and in-line comments for more details. In particular, you may want to use `collections.defaultdict` and `dict.get()` to simplify handling of unknown keys. See dict_notes.md (dict_notes.md) for a brief overview of how to use these.

Note on keys and word-order: Convention in the math is to write context in reverse order, as in $P(w \mid w_{i-1}, w_{i-2})$, but in the code it'll be much easier to write things left-to-right as in normal English: so for the context "`foo bar ___`", you'll want to use `("foo", "bar")` as a dict key.

No solutions for part (b)

We won't be releasing code solutions, as this makes it difficult to re-use the assignment in future iterations of the class. If you have specific questions on your code, please come to office hours or post a private question on Piazza.

In [2]:
```
reload(ngram_lm)
utils.run_tests(ngram_lm_test, ["TestAddKTrigramLM"])
```

```
test_context_totals (ngram_lm_test.TestAddKTrigramLM) ... ok
test_counts (ngram_lm_test.TestAddKTrigramLM) ... ok
test_next_word_proba_k_exists (ngram_lm_test.TestAddKTrigramLM) ... ok
test_next_word_proba_no_smoothing (ngram_lm_test.TestAddKTrigramLM) ... ok
test_no_mutate_on_predict (ngram_lm_test.TestAddKTrigramLM) ... ok
test_words (ngram_lm_test.TestAddKTrigramLM) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.011s

OK
```

Kneser-Ney Smoothing

In this part, we'll explore Kneser-Ney smoothing as a more sophisticated way of estimating unseen probabilities.

When building an n-gram model, we're limited by the model order (e.g. trigram, 4-gram, or 5-gram) and how much data is available. Within that, we want to use as much information as possible. Within, say, a trigram context, we can compute a number of different statistics that might be helpful. Let's review a few goals:

1. If we don't have good n-gram estimates, we want to back off to (n-1) grams.
2. If we back off to (n-1) grams, we should do it "smoothly".
3. Our counts C_{abc} are probably *overestimates* for the n-grams we observe (see *held-out reweighting*).
4. Type fertilities tell us more about $P(w_{new} \mid \text{context})$ than the unigram distribution does.

Kneser-Ney smoothing combines all four of these ideas.

Absolute discounting - which follows from 3. - gives us an easy way to backoff (1. and 2.), by distributing the subtracted probability mass among the backoff distribution $\tilde{P}(c \mid b)$. The amount to redistribute, δ, is a hyperparameter selected based on a cross-validation set in the usual way, although for this assignment we'll just let $\delta = 0.75$.

$$P_{ad}(c \mid b, a) = \frac{C_{abc} - \delta}{C_{ab}} + \alpha_{ab}\tilde{P}(c \mid b)$$

Where α_{ab} is a backoff factor, derived from the counts, that guarantees that the probabilities are normalized: $\sum_{c'} P_{ad}(c' \mid b, a) = 1$. This definition is recursive: if we let $\tilde{P}(c \mid b) = P_{ad}(c \mid b)$, then the backoff distribution can also back off to even lower n-grams.

Note: we need the numerator above to positive, so it should actually read $\max(0, C_{abc} - \delta)$.

Type fertility is item 4. Instead of falling back to the unigram distribution at the end, we'll define $\hat{P}(w)$ as proportional to the type fertility of w, or the *number of unique preceding words* w_{i-1}. In the following equation, the word we are estimating the probability of is c. b' are the set of words we've found occurring before c in the training data.

$$\hat{P}_{tf}(c) \propto \mid b' : C_{b'c} > 0 \mid = tf(c)$$

In order to make this a valid probability distribution, we need to normalize it with a factor $Z_{tf} = \sum_w tf(w)$, so we have $\hat{P}_{tf}(w) = \frac{tf(w)}{Z_{tf}}$

KN Equations

Putting it all together, we have our equations for a KN trigram model:

$$P_{kn}(c \mid b, a) = \frac{\max(0, C_{abc} - \delta)}{C_{ab}} + \alpha_{ab}P_{kn}(c \mid b)$$

where the bigram backoff is:

$$P_{kn}(c \mid b) = \frac{\max(0, C_{bc} - \delta)}{C_b} + \alpha_b P_{kn}(c)$$

and the unigram (type fertility) backoff is:

$$P_{kn}(c) = \frac{tf(c)}{Z_{tf}} \quad \text{where} \quad tf(c) = |\ b' : C_{b'c} > 0\ |$$

Note that there is only one free parameter in in the above equations: δ. You'll compute α_{ab} and α_b in the exercise below.

Part (c): Short answer questions

Give brief answers to the following, in the cell below.

1. Compute the value of α_b such that $P_{kn}(c \mid b)$ is properly normalized. Give your answer in terms of the discount term δ, the context total C_b, and $\text{nnz}(b) = |\ c' : C_{bc'} - \delta > 0\ |$, the number of bigrams `bc'` with positive (discounted) count.
   Hint: solve $\sum_{c'} P_{kn}(c' \mid b) = 1$ using the fact that $\sum_{c'} P_{kn}(c') = 1$ and $\sum_{c'} C_{bc'} = C_b$.
   Note that if you replace b with the context ab, you can re-use this result to define α_{ab}.

2. Based on your answer to question 1, in which case do you expect the KN model to rely more on the backoff distribution (i.e. higher α)? Assume $\delta = 0.75$.
   Case (a): context `<s> my name is ___`, which occurs 1000 times ending with 632 unique words.
   Case (b): context `Mister Rogers ___`, which occurs 5 times and always ends with "Neighborhood".
   Explain *briefly* why this is the case, and why this is reasonable behavior given your intuition.

Answers for Part (c)

SOLUTIONS - do not distribute!

You can use LaTeX to typeset math, e.g. `$ f(x) = x^2 $` will render as $f(x) = x^2$. For Question 1, **please show your work!** You can use LaTeX, or attach an image to your submission. To add an image to the notebook, use Markdown: `![alt_text](partc1.png)`

1. Beginning with the absolute discounting equation:
$$P_{ad}(c \mid b) = \frac{\max(0, C_{bc} - \delta)}{C_b} + \alpha_b \tilde{P}(c)$$
We can sum both sides with respect to the target word c:
$$\sum_c P_{ad}(c \mid b) = \sum_c \frac{\max(0, C_{bc} - \delta)}{C_b} + \sum_c \alpha_b \tilde{P}(c)$$
We let the left side $= 1$ to enforce that our distribution is normalized, and also collapse the second term on the right side using the fact that $tildeP(c)$ is normalized:
$$1 = \sum_c \frac{\max(0, C_{bc} - \delta)}{C_b} + \alpha_b$$
Now for the remaining sum, we can re-write to ignore the zeros:
$$\sum_c \frac{\max(0, C_{bc} - \delta)}{C_b} = \frac{1}{C_b} \sum_c \max(0, C_{bc} - \delta) = \frac{1}{C_b} \sum_{c\colon C_{bc} - \delta > 0} (C_{bc} - \delta)$$
and split it up to simplify further:
$$\frac{1}{C_b} \sum_{c\colon C_{bc} - \delta > 0} (C_{bc} - \delta) = \frac{1}{C_b} \left[\sum_{c\colon C_{bc} - \delta > 0} C_{bc} - \sum_{c\colon C_{bc} - \delta > 0} \delta \right]$$
Now if we assume that $\delta < 1$, then $\sum_{c\colon C_{bc} - \delta > 0} C_{bc} = C_b$. So we can simplify to:
$$\frac{1}{C_b} \left[\sum_{c\colon C_{bc} - \delta > 0} C_{bc} - \sum_{c\colon C_{bc} - \delta > 0} \delta \right] = \frac{1}{C_b} [C_b - nnz(b) \cdot \delta] = 1 - \frac{nnz(b) \cdot \delta}{C_b}$$
Now we can substitute this back in to our first identity, to get:
$$1 = 1 - \frac{nnz(b) \cdot \delta}{C_b} + \alpha_b$$
which gives us our final answer:
$$\alpha_{\mathbf{b}} = \frac{\mathbf{nnz(b)} \cdot \delta}{\mathbf{C_b}}$$

   Note that $\alpha \in [0, 1]$, and can be interpreted as the fraction of probability mass that we're giving to the backoff distribution.

   Intuitively, this should look reasonable: the more discounting (δ) we apply, the more we rely on the backoff distribution. Conversely, the higher the counts C_b (all else being equal), the less we'll backoff.

2. Using your formula from (1), you can calculate this directly:
(a) $nnz(<\texttt{s}>\texttt{my name is}) = 632$, and $C_{\texttt{<s> my name is}} = 1000$, so we have $\alpha = \frac{632 \cdot 0.75}{1000} = 0.474$.
(b) $nnz(\texttt{Mister Rogers}) = 1$, and $C_{\texttt{Mister Rogers}} = 5$, so we have $\alpha = \frac{1 \cdot 0.75}{5} = 0.15$.
So, even though context (a) is observed many more times, we rely much more heavily on the backoff

distribution in that case. This should make sense, given that context (a) is likely to support a diverse set of completions - and it's unlikely that we observed them all in our corpus.

Part (d): Implementing the KN Model

Implement the `KNTrigramLM` in `ngram_lm.py`. As with the add-k model, we've provided some starter code for you; you need only fill in the marked blocks. `KNTrigramLM` also conforms to the exact same interface as the add-k model.

You should:

- Finish the implementation of `__init__(self, words)` to compute the necessary quantities
- Implement the `kn_interp(...)` function, which interpolates between n-gram counts and a backoff distribution according to the absolute discounting rule (see definitions of $P_{kn}(c \mid a, b)$ and $P_{kn}(c \mid b)$). You'll need your definition of α from (c).1. here.

As before, see the function docstrings and in-line comments for more details.

When you're done implementing it, run the cells below to train your model, sample some sentences, and evaluate your model on the dev set. Then jump to Part (e) for a few last questions.

No solutions for part (d)

We won't be releasing code solutions, as this makes it difficult to re-use the assignment in future iterations of the class. If you have specific questions on your code, please come to office hours or post a private question on Piazza.

We've left the output of our solution model (`KNTrigramLM`) below, for your reference.

In [3]:
```
reload(ngram_lm)
utils.run_tests(ngram_lm_test, ["TestKNTrigramLM"])
```

```
test_context_nnz (ngram_lm_test.TestKNTrigramLM) ... ok
test_context_totals (ngram_lm_test.TestKNTrigramLM) ... ok
test_counts (ngram_lm_test.TestKNTrigramLM) ... ok
test_kn_interp (ngram_lm_test.TestKNTrigramLM) ... ok
test_next_word_proba (ngram_lm_test.TestKNTrigramLM) ... ok
test_no_mutate_on_predict (ngram_lm_test.TestKNTrigramLM)
Don't allow modifications to the LM during inference. ... ok
test_type_contexts (ngram_lm_test.TestKNTrigramLM) ... ok
test_type_fertility (ngram_lm_test.TestKNTrigramLM) ... ok
test_words (ngram_lm_test.TestKNTrigramLM) ... ok
test_z_tf (ngram_lm_test.TestKNTrigramLM) ... ok

----------------------------------------------------------------------
Ran 10 tests in 0.013s

OK
```

Select your model

Select either `AddKTrigramLM` or `KNTrigramLM` in the cell below. If switching models, you only need to re-run the cells below here - no need to re-run the preprocessing.

In [6]:
```
reload(ngram_lm)

# Uncomment the line below for the model you want to run.
# Model = ngram_lm.AddKTrigramLM
Model = ngram_lm.KNTrigramLM

t0 = time.time()
print("Building trigram LM... ", end="")
lm = Model(train_tokens)
print("done in {:.02f} s".format(time.time() - t0))
lm.print_stats()
```

```
Building trigram LM... done in 6.20 s
=== N-gram Language Model stats ===
  30,000 unique 1-grams
 357,896 unique 2-grams
 732,467 unique 3-grams
Optimal memory usage (counts only): 24.21 MB
```

Change `params` to change the smoothing factor. `AddKTrigramLM` will ignore the value of `delta`, and `KNTrigramLM` will ignore `k`.

In [7]:
```
lm.set_live_params(k = 0.001, delta=0.75)
```

Sampling Sentences

In [7]:
```
max_length = 20
num_sentences = 5

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

```
<s> <s> <s> the plan ? ? </s>
[4 tokens; log P(seq): -17.22]

<s> <s> go causes jackie i examined far greater . </s>
[8 tokens; log P(seq): -94.68]

<s> <s> for he had a job . </s>
[6 tokens; log P(seq): -28.28]

<s> <s> the president kennedy's snatches , two other large cities of this gro
up which an award can , with respect to
[20 tokens; log P(seq): -121.08]

<s> <s> washington on . </s>
[3 tokens; log P(seq): -27.91]
```

Scoring on Held-Out Data

Your KN model should get a perplexity of around 280 on the dev set with $\delta = 0.75$. The add-k smoothing model will perform... somewhat worse :)

In [9]:
```
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: 17.14
Test perplexity: 283.93
```

Part (e): Additional Questions

Answer the following in the cell below.

1. What is the average number of times our model sees any particular trigram, averaged across the trigrams we observed at least once (i.e. ignoring the zeros for trigrams we never observed)? How about averaged across *all possible* trigrams (i.e. including hypothetical ones we never observed)? (*Hint:* you don't need to write any code for this - it should be a quick calculation.)

2. Based on your answer above, do you think that a 4-gram or 5-gram model would perform better than the trigram model on the 1 million word Brown corpus? How about on a 42-billion word Wikipedia corpus?

3. Which model generates more "realistic" sentences - `AddKTrigramLM`, `KNTrigramLM`, or the unsmoothed `SimpleTrigramLM` from the demo notebook? Is this in-line with their perplexity on the dev set?

Note: in the next assignment, we'll implement a neural network model that avoids the sparsity problem altogether and can achieve significantly better generalization even on a small dataset like the Brown corpus.

Answers for Part (e)

SOLUTIONS - do not distribute!

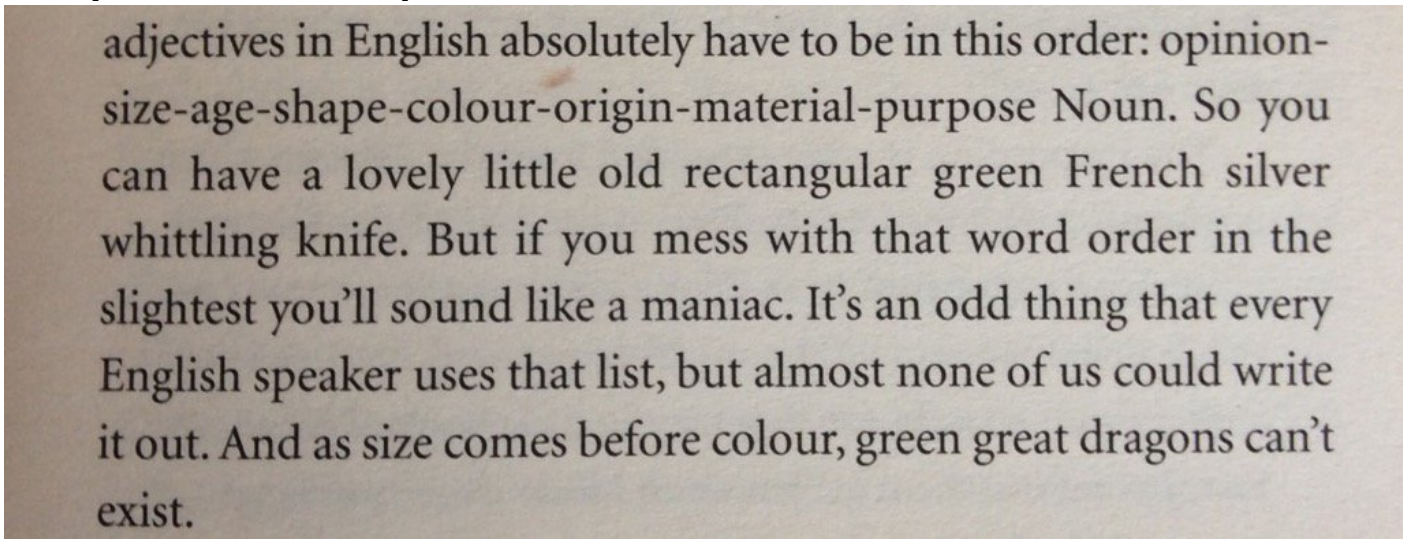
1. In the first part, we want the expected count $\mathrm{E}[C_{abc}]$ of any trigram abc, assuming that trigram is observed at least once. Formally this is equal to:
$$\frac{\sum_{abc\,:\,C_{abc}>0} C_{abc}}{\sum_{abc\,:\,C_{abc}>0} 1}$$
The denominator is the number of unique trigrams observed. The numerator is just the total number of trigrams we observed, which is equal to $|C|$, the number of the tokens in the corpus (*technically, it's* $|C|-2$*, but we'll ignore that minor correction*). So for the Brown corpus, we have 1.16M tokens and 733k unique trigrams, for $\mathrm{E}[C_{abc}] = 1.58$. So on average, we saw each *observed* trigram less than twice!
If we average across *all possible* trigrams, then we set the denominator to $|V|^3$. For $|V| =$ 30k, we get $\mathrm{E}[C_{abc}] = \frac{1.16\cdot 10^6}{27\cdot 10^{12}} = 0.043 \cdot 10^{-6}$! We'd need a tremendous amount of data to see all trigrams - although we're helped somewhat by the fact that most are nonsensical and will never appear.

2. We're already stretching the limits of a trigram model on the 1M word Brown corpus. You could probably extract some additional performance out of 4-gram or 5-gram models, but the longer n-grams could only be used in a small fraction of contexts where there are sufficient counts. On the much larger Wikipedia corpus, higher-order models should perform somewhat better.

3. We'd expect the unsmoothed `SimpleTrigramLm` to generate the "best" sentences, because it's most faithfully representing the input data (by maximum likelihood). When we smooth the model to reduce overfitting, we introduce some bias that moves us away from the "true" distribution of natural language.

Just for fun: Linguistic Curiosities

You might have seen this floating around the internet:



adjectives in English absolutely have to be in this order: opinion-size-age-shape-colour-origin-material-purpose Noun. So you can have a lovely little old rectangular green French silver whittling knife. But if you mess with that word order in the slightest you'll sound like a maniac. It's an odd thing that every English speaker uses that list, but almost none of us could write it out. And as size comes before colour, green great dragons can't exist.

source: https://twitter.com/MattAndersonBBC/status/772002757222002688?lang=en (https://twitter.com/MattAndersonBBC/status/772002757222002688?lang=en)

Let's see if it holds true, statistically at least. Note that log probabilities are always negative, so the smaller magnitude is better. And remember the log scale: a difference of score of 8 units means one utterance is $2^8 = 256$ times more likely!

In [10]:
```
def preprocess_for_scoring(sentence, vocab):
    # Pre-process words, replace anything the model doesn't know with <unk>
    words = [utils.canonicalize_word(w, wordset=vocab.wordset)
             for w in sentence]
    # Pad sequence with start and end markers
    return [u"<s>", u"<s>"] + words + [u"</s>"]

s0 = preprocess_for_scoring("square green plastic toys".split(), vocab)
s1 = preprocess_for_scoring("plastic green square toys".split(), vocab)
```

In [11]:
```
print("s0 score: {:.02f}".format(lm.score_seq(s0)[0]))
print("s1 score: {:.02f}".format(lm.score_seq(s1)[0]))
```

```
s0 score: -52.01
s1 score: -60.96
```

In [12]:
```
noun = "toys"
adjectives = ["square", "green", "plastic"]
results = []
for adjs in itertools.permutations(adjectives):
    words = list(adjs) + [noun]
    seq = preprocess_for_scoring(words, vocab)
    score, _ = lm.score_seq(seq)
    results.append((score, words))

# Sort results
for score, words in sorted(results, reverse=True):
    print("\"{:s}\" : {:.02f}".format(" ".join(words), score))
```

```
"green square plastic toys" : -51.95
"square green plastic toys" : -52.01
"plastic square green toys" : -60.96
"plastic green square toys" : -60.96
"green plastic square toys" : -61.25
"square plastic green toys" : -61.31
```