1 Forward and Backward Bigram Models

In the following, we use two perplexity terms: **sentence perplexity** (called "Perplexity" in the sample trace file) and **non-terminating perplexity** (called "Word Perplexity" in the sample trace file and assignment). Both of these are normalized by the number of words, so it is confusing to call one "Word Perplexity" and the other just "Perplexity"; the significant difference is that sentence perplexity models both sentence boundaries, while non-terminating perplexity *does* not model the terminating boundary.

We can implement the backward bigram model with a tiny change to the forward bigram model: instead of conditioning each bigram on the first token in the bigram, simply condition it on the second token. We use the same weights as in the provided library: $\frac{1}{10}$ on the unigram probability, $\frac{9}{10}$ on the bigram probability. We perform smoothing for both forward and backward bigram models by replacing the leftmost token (instance) of each type with the special "<UNK>" token. For this choice of smoothing, there is no difference in sentence perplexity between the forward bigram and backward bigram models. This makes sense for evaluation of training data, but it is potentially surprising that performance is equal for both backward and forward models on the test dataset.

If we instead choose to replace the *last* occurrence of each word type with the <UNK> token, the perplexity of the backward bigram is slightly different. For a large corpus, like the provided WSJ dataset, this difference in (log) perplexity is less than 1%—small compared to the other differences we discuss.

The big difference is in the non-terminating perplexity, when we condition on the initial sentence boundary marker, but do not predict the termination of the sentence (the end in the forward case, the beginning in the backward case). This should indicate whether a language is left-branching or right-branching. If we are working from left to right and we find it harder to predict where the sentence ends, than working from right to left and trying to predict where the sentence begins, that means the language is more right branching. This is the case with English, and the logic follows through in the more diverse datasets, *Wall Street Journal* and *Brown*. In these two datasets, we see that non-terminating perplexity is somewhat lower than complete sentence perplexity, about a 3% reduction in log perplexity. This reduction simply shows that our backward bigram model is slightly better at predicting where the beginning of the sentence is, compared to how good our forward bigram model is at predicting where the sentence ends.

However, with the Atis dataset (airline booking conversations), the forward bigram model is better. One hypothesis for this might be that in spoken discourse, like telephone conversations, one tends to insert greetings or other arbitrary interjections at the beginning of a sentence.

1.1 Results

Here are the forward vs. backward bigram model results for non-terminating perplexity (log scale):

WSJ	Forward model	Backward model
train	88.89	86.65
test	275.11	266.58
Brown	Forward model	Backward model
train	113.36	110.78
test	310.72	299.89
Atis	Forward model	Backward model
train	10.59	11.68
test	24.05	27.12

2 Bidirectional Bigram Models

The next step is to combine these two models. Not only does this let us reasonably predict both beginning of sentence and end of sentence markers, it factors in a sort of trigram model, but conditioning the center word on the surrounding two words while smoothing over the sparsity of trigram data. The datasets we're using here are diverse but relatively small (only about a million words), so they are not well-suited to trigram models. Trigram models can be used in a variety of ways; the primary way is to use the previous two words to predict the next one in a sequence (like the forward bigram model). Or we could use the tokens before and after a center token to predict that token. This is similar to the bidirectional model, but the trigram model is less flexible, since it only knows to apply the predictions from trigrams that exactly match the form <left, ?, right>. The bidirectional bigram, instead, looks at the combinations, <left, ?> and <?, right>, and combines the predictions.

The implementation of this model requires no new code, only a recalculation of the final log perplexity. Because the forward and backward bigram models produce mostly equivalent perplexities, we'll weight the two equally, so the final perplexity is derived from the following formula:

$$\frac{1}{10}p_{unigram} + \frac{9}{10}\left(\frac{1}{2}p_{forward-bigram} + \frac{1}{2}p_{backward-bigram}\right)$$

This new model produces the following results (again, log-scale non-terminating perplexity):

WSJ	Forward model	Backward model	Bidirectional model
train	88.89	86.65	26.09
test	275.11	266.58	108.70
Brown	Forward model	Backward model	Bidirectional model
train	113.36	110.78	29.89
test	310.72	299.89	127.32
Atis	Forward model	Backward model	Bidirectional model
train	10.59	11.68	5.17
test	24.05	27.12	14.73

This is a pretty impressive improvement! This new model over-fits more than before, but the much lower perplexity demonstrates that this model is superior.

3 Smoothing Out-of-Vocabulary (OOV) Tokens

As noted in my Question of the Week, we can artificially drop perplexity due to the way the provided BigramModel class handles out of vocabulary words. Using the out-of-the-box forward bigram model:

```
$ java -cp . nlp.lm.BigramModel \
     ../data/penn-treebank3/tagged/pos/brown/ 0.1
# Train Sentences = 47207 (# words = 1079440)
# Test Sentences = 5245 (# words = 93530)
Training...
Perplexity = 93.51927720192262
Word Perplexity = 113.35954408376686
Testing...
Perplexity = 231.30243689356243
Word Perplexity = 310.66735613437913
$ java -cp . nlp.lm.BigramModel
      ../data/penn-treebank3/tagged/pos/brown/ 0.9
```

```
# Train Sentences = 5245 (# words = 127667)
# Test Sentences = 47207 (# words = 1045303)
Training...
Perplexity = 60.29405209041766
Word Perplexity = 71.00256174673837
Testing...
Perplexity = 222.516464255008
Word Perplexity = 282.17933493106415
```

Using more extreme training / test splits (like 0.99) produces an even more striking drop in perplexity. Of course, it's ridiculous to train on one tenth of the data and test on the rest.[1] Usually, annotated data is so scarce that you minimize the test set as much as possible.

But the experiment suggests that the way OOV words are handled is flawed; they are not just a new type, they are an entirely different kind of predictable item. The goal of transforming unknown words into special tokens is so that we don't end up with impossible sentences in our test data just because we've never seen a particular unigram before. But the way this is handled in the given library, we end up trying to predict the unknown-ness of tokens in certain contexts. Instead, we should be penalized for any unknown unigram or bigram. Another side effect of this model is that the unknown-ness of some token is used to predict the preceding or following tokens, which seems incorrect.

This model's approach to smoothing is better than nothing; it doesn't completely break on OOV words. However, these results suggest that the evaluation step (measure of perplexity) is skewed when we use a disproportionate train/test split. This will probably become less of a problem as we use more data, but we predict that evaluation will be especially skewed for small data sets, particularly if there is a high ratio of word types to word tokens.

4 A note on my Scala code

You can run it with SBT using something like these commands, depending on your directory structure:

```
$ sbt
> run-main nlp.cb.NgramEvaluator
                        ../data/penn-treebank3/tagged/pos/brown/ 0.1
```

I rewrote the BigramModel class as an exercise for my own benefit (and it just so happened that having this code fresh in my mind was very useful for one of my Google internship technical interviews), but the forward bigram model produces input identical to Ray's code.

The whole project (without data) in the proper SBT project directory stucture can be found in the following github repository:

```
git clone https://github.com/chbrown/nlp.git
```

[1] I don't think I've ever seen that in a research paper.